Cross-Building Linux

the Little Blue Linux build process

Brett Neumeier

Version , 2021-01-09

Table of Contents

Introduction and Overview

Little Blue Linux (variously referred to here as *LB Linux*, *LBL,* or just *Little Blue*) is a basic but usable GNU/Linux operating system distribution that includes all the programs needed to rebuild itself from source code. It is most suitable for use on servers rather than desktop or laptop computers, because it lacks a graphical user interface, but it is as flexible and extensible as any other basic GNU/Linux system with C and C++ development tools installed.

Cross-Building Linux, or *CBL,* is this book. It outlines a step-by-step process you can follow to build the Little Blue Linux system entirely from source code. If you follow it precisely, what you wind up with is a Little Blue Linux system. If you modify the process, what you wind up with will be a variant or derivative of Little Blue Linux — perhaps you'll like the result enough to give it a name and make it available to other people!

The aspect of all this that I focus the most on — the thing that is most important to me — is the *process:* the narrative that describes every necessary component of a GNU/Linux system, how they fit together, and how to bootstrap them all, starting from an existing GNU/Linux system, to create a new complete, minimal, self-hosted GNU/Linux system, entirely from source code.

The most important design goal is that the entire process should be as clear and transparent as possible. Ideally, it should be difficult to read through CBL *without* understanding how the resulting system works and how it was put together. (I realize that's a pretty lofty objective, but you've got to have dreams.)

A secondary goal, almost as important as the first, is that I want to be sure that *every piece of the final system was built from source code*. That is, I want to be confident that none of the binary programs or libraries from the initial system — where you start the build — winds up simply being copied to the resulting system. I want to be certain that everything has been rebuilt from the ground up. I'll talk more about why that matters to me later.

It's also desirable for the resulting system, Little Blue Linux, to be useful. But that utility is only a tertiary consideration! Mostly what I'm interested in is telling a story about how *you* can create a GNU/Linux system — the programs and libraries and configuration elements that comprise it — and how all those pieces fit together.

As it turns out, I do find that Little Blue Linux is an outstanding server system: since it has only the software packages that I really need, it has a minimal attack surface, and it's trivial to rebuild everything with specific compiler optimizations for whatever hardware platform I want to run it on, because the CBL process is explicitly designed to be automated.

It would be strange if I *didn't* find Little Blue Linux to be ideal for my purposes — after all, I'm making all the policy decisions about what components to use and how to configure them! That leads to yet another design goal for CBL: I would like to ease the path of anyone who wants to build *their* ideal system to do that — perhaps by starting with CBL and modifying it to suit their own tastes, or perhaps by doing something entirely different.

If you have ideas about what your own ideal computer system should look like, how you want it to work, maybe CBL will serve as a starting point for that. Even if it does not, the most important thing

about CBL is that it is a demonstration that there is no deep mystery or ancient magical lore involved in how GNU/Linux systems work — this book is not exactly short, but there is nothing really *hard* in it. Anyone who wants to build a custom system exactly to their own taste can do it! You just have to do some work.

Chapter 1. About Little Blue Linux

Before we get into the details of how we'll build the system, let's talk a little bit about what you'll wind up with once it's complete. All GNU/Linux systems have a fair amount in common with each other, as well as a handful of differentiating factors; this is a brief overview of what those factors are in LB Linux. All of these are discussed more fully later on in the narrative.

1.1. S6-Based Init System

The init framework — which bootstraps the userspace environment and manages all the background "daemon" processes that are needed in a healthy GNU/Linux system — is based on the `s6` suite of programs by Laurent Bercot. This uses a bunch of tiny little programs to manage the basic userspace of the system, rather than a few really big ones — looking at the process table on a basic LBL system, there are 38 s6-related processes running, adding up to a total of 6725 pages of RAM.[1] In contrast, looking at a systemd-managed system (booted into single-user mode so that an absolutely minimal set of programs is running), there are only seven processes running — but they occupy a total of 23,480 pages of memory.

1.2. Package Users

Rather than maintaining a central database of installed packages, and providing specialized package-management tools to query that database to discover things like what files are part of a package or what package a specific file belongs to, LBL simply creates a separate user account for each package. The files installed by each package are owned by the package-specific user, so standard system utilities can be used to determine package information. To find out what package is responsible for a file, for example, you can just use `ls -l`; or you can `find / -user ⋯` to list all the files and directories owned by a package.

1.3. Version Control For Configuration Files

I have, for years, had the habit of maintaining a version control repository of configuration files for the programs I use a lot — especially programs like `bash` and `vim` and `tmux`, whose configurations I have customized, in some cases extensively. A lot of people I know do this — it's very handy to be able to start with a fresh-out-of-the-box operating system installation, and configure everything the way you like it just by checking out your normal configuration files from version control.

After I'd been doing that for a while, it occurred to me that exactly the same considerations apply for *system* configuration files — things like the configuration files for `sshd` and `sudo` and other such programs. Why not put all of those configuration files in a version control repository as well, so you have a record of what files changed, and when, and for what reason? On servers managed by more than one person, it can also be helpful to know who made those changes.

That's part of what all Little Blue Linux systems have: a git repository for tracking changes to system configuration and policy files, accessed using a simple wrapper script.

1.4. Modern Versions of Everything

Sometimes, when I've been interested in using a program but am using a distribution like Debian or CentOS, I have run up against version constraints on dependencies. The current version of QEMU, say, has a feature I'd like to use, so I try to install it; but then it turns out it needs four or five libraries with a version later than what is provided in the distribution package repositories, and building modern versions of those libraries reveals other updates that *they* need — it's frustrating!

Little Blue Linux is not always completely up-to-date, since new versions of packages are released all the time, but it's reasonably close. Every few months I go through the packages that make up the base LBL system and update the blueprints to use the latest stable version of everything. So I've never run into a situation where I have to upgrade a bunch of other packages to be able to use a modern version of something else.

(To be completely honest, this isn't as much of a selling point as it could be — because, although there are not ancient or obsolete versions of any package in LBL, there are a huge number of packages for which CBL simply has no blueprints, and you'll have some work to do to use them at all. But it suits *my* purposes just fine!)

1.5. Cross-Architecture Build Process

As I mentioned in the Introduction and Overview, I want to be sure that every part of the final system was built from source code, and that no binary code was simply copied from the original host system to the final system. The best way I've thought of to make sure of this is to make sure that none of the code from the host system will actually *work* on the eventual Little Blue Linux system that we are building here. We do that by building everything for a different kind of computer than the one we start the build on. The very first part of the CBL process constructs a *cross-toolchain:* a compiler and related tools that run on one type of computer — conventionally, this is referred to as the "host" system — but create programs and libraries that will work on some completely different type of computer architecture: the "target" system.

The CBL process itself breaks down naturally into two different parts: the part that you run on the *host* system, and the part that you run on the *target* system. I sometimes refer to those as the two "sides" of the CBL process, the "host side" and the "target side."

The idea of using a cross-toolchain, to start with one kind of computer and use it to build a system that works on a different kind of computer, is so fundamental to CBL that I named the process after it. Within that constraint, though, there is a lot of flexibility in how to perform the CBL build. The host can be a physical machine, like an Intel x86-architecture notebook computer or ARM-architecture chromebook, or it can be a virtual machine emulated by a program like QEMU. There are advantages and disadvantages to both options. The target, similarly, can be a physical computer or a QEMU virtual machine; and, again, there are benefits and drawbacks to both. Any of those combinations can work — you can use a physical computer as the host system, then move all the pieces you built there to a different physical computer to finish the build, or you can use a virtual machine as the host system and a different virtual machine as the target system, or anything else you can think of.

The main benefits of using QEMU — for the host or target or both — are, first, that you don't need to

have a real computer with the architecture you want to use for that side of the CBL process; and, second, that it's possible to automate the entire build process — since you don't have to move a physical storage device from one computer to another, or press any actual power buttons, or anything like that.

The main disadvantages of using QEMU are, first, that emulated systems are generally a lot slower than real computer systems, so the build process takes a long time; and, second, QEMU is sometimes not as stable as a real computer. When running an ARM64-target emulator on a 64-bit Intel notebook computer, QEMU sometimes crashes during the final system glib build, for example. In many cases this appears to be caused by the limited system resources (especially memory) available on the emulated system. When using QEMU to emulate a computer (as opposed to using it to run a virtual machine of the same architecture as the host system), I primarily emulate ARM systems, because QEMU for ARM can emulate a computer with any amount of RAM by using the `virt` machine type.

In theory, you can follow the CBL process with any kind of computer as the host or target platform, as long as they are supported by the GNU toolchain programs and the Linux kernel, but it seems as though every different architecture presents idiosyncracies that require additional work to support. That means that if you go outside the host/target pairings that we use for developing and testing CBL, you will probably need to do some additional work.

The physical computer systems that I use are 64-bit x86 (aka Intel- or AMD-architecture) computers, 32- and 64-bit ARM systems, and 32-bit MIPS systems — because those are the types of computers I have handy.

1.6. About This Specific Build

The canonical form of *Cross-Building Linux* is a set of "blueprint" files that are available in a publicly-accessible git repository.[2] If you're reading this as a book or web page, it was produced from those blueprint files by the litbuild program.

Every time litbuild produces the CBL book, it configures it for a specific type of build, with a particular kind of host system and target system. It also includes instructions on how to launch the target-side build in QEMU if those are relevant, and so on. If this book does not describe the type of build you want to do, all you need to do is obtain the CBL blueprint files and the litbuild program, set some environment variables as described in the Configuration Parameters and Default Values section, and use litbuild to generate a new version of the book.

For this CBL build, the host system is `x86_64-unknown-linux-gnu` and the target system is `aarch64-cbl-linux-gnu`. The final system will be called `cblviton.freesa.org`. Most of the work will be done in the `/tmp/cblwork/build` directory — so the storage device where that directory lives should have a few dozen GB of free space.

If any of that looks wrong — or if any of the other parameters in Configuration Parameters and Default Values are not set the way you want them to be — change the configuration and generate a new book!

[1] I think pages are 4096 bytes, but I haven't looked it up so I might be wrong.

[2] currently at https://gitlab.com/freesa/cross-building-linux

Chapter 2. Configuration Parameters and Default Values

2.1. How configuration parameters work

The CBL build can be adjusted and tuned in a variety of ways using the configuration parameters described in this section. To override any parameter from its default value, you can set an environment variable with its name to a new value (or simply modify this blueprint).

The *default* values here are appropriate for a build taking place on an Intel or AMD 64-bit computer, building a 64-bit ARM target system, and using a QEMU-emulated virtual machine for the target system. (The defaults work regardless of whether the host system is a physical computer or QEMU virtual machine.)

Each sub-section below discusses a related set of configuration parameters.

2.2. Setting The Host And Target Architectures

These parameters determine what type of build is done and what options are used to control aspects of that build.

Parameter: HOST

Value: `x86_64-unknown-linux-gnu` *(default)*

As described in the overview, the CBL build process *always* uses a cross-toolchain, and produces a version of Little Blue Linux that will run on a different kind of computer than the initial host system. The `HOST` parameter should be set to the "triplet" of the system where the host side of CBL (that is, the cross-toolchain itself and the target-system "scaffolding" programs built using it) will be constructed. You can read more about "triplets" in the Constructing a GNU Cross-Toolchain section or in the documentation of GNU Autoconf (as of Autoconf 2.71, triplets are described in section 14).

The easiest way to get the correct value here is simply to run the script `config.guess`, found in the sources for GCC or Autoconf, on whatever computer or virtual machine you're going to use for the host side of the CBL build.

Parameter: TARGET

Value: `aarch64-cbl-linux-gnu` *(default)*

`TARGET` is the other parameter used to control cross-compilation. It should be set to the triplet of the *target* system, whatever computer or virtual machine will be booted into using the scaffolding. The sample configuration files provided as part of CBL may be helpful in figuring out what you should set this to. The second component of the triplet is conventionally set to `cbl` when following the CBL process.

Parameter: TARGET_GCC_CONFIG

Value: `--enable-fix-cortex-a53-835769 --enable-fix-cortex-a53-843419 --with-cpu=cortex-a72.cortex-a53 --enable-standard-branch-protection` *(default)*

When configuring GCC for the target system, it may be useful or necessary to specify some set of configuration flags. You can consult the GCC installation instructions and the gcc section for more details about the options available to you. The default shown here is suitable for the Rockchip RK3399 SOC, which has two Cortex-A72 CPU cores and four Cortex-A53 CPU cores in what's called a "big.LITTLE" architecture. (That default value is used simply because I happen to have such a system.)

Parameter: HOST_GCC_CONFIG

Value: not set *(default)*

It's sometimes useful or necessary to specify some set of configuration flags when configuring GCC for the *host* system (that is, for the GCC build done in Trustworthy Host-System Programs, if those are being built). This works just like `TARGET_GCC_CONFIG`, but for the initial native GCC.

Parameter: TARGET_GMP_CONFIG

Value: not set *(default)*

Similarly to GCC, the GMP library may need to have some extra configuration flags specified — so that it knows what ABI to build for, for example.

Parameter: KERNEL_ARCH

Value: `arm64` *(default)*

Different packages or programs have different ways of referring to CPU architectures. As mentioned earlier, the GNU toolchain refers to "triplets," which you can read about in Constructing a GNU Cross-Toolchain; the CPU architecture is the first component of the triplet. The Linux kernel has its own naming convention for CPU architectures, which in many (but not *all*) cases is the same as the CPU field in the triplet.

The default value for the `KERNEL_ARCH` parameter is an example where the naming convention differs. The GNU toolchain refers to the 64-bit ARM architecture as `aarch64` (for "ARM Architecture, 64-bit words"), but the Linux kernel calls it `arm64`.

Another example is MIPS. The Linux kernel has a single architecture, `mips`, that is used for all MIPS variants (big-endian and little-endian, with both 32-bit and 64-bit word lengths), but each variant has a different value for the CPU component of the triplet: "mips," "mipsel", "mips64," and "mips64el."

Parameter: TARGET_EXPECTED_MACHINE_NAME

Value: `AArch64` *(default)*

To verify that the cross-toolchain is working as expected, CBL compiles a simple program and then inspects the resulting binary to see whether it is built for the correct kind of target machine. If the binary doesn't indicate that it's built for the machine type specified by this parameter, the build will halt so you can inspect the situation and see what's going on.

Parameter: KERNEL_CONFIG

Value: `defconfig` *(default)*

The Linux kernel has approximately a jillion [1] different configuration elements. These determine the hardware devices and features that will be supported by linux kernel produced by the build. Starting from scratch isn't necessarily a good idea; luckily, we don't have to do that, because the kernel is distributed with a set of default configuration files for every supported CPU architecture. This parameter sets the default configuration file that will be used as a starting point for the CBL build.

This parameter has some relation to the QEMU machine type, when targeting a QEMU emulated machine.

Parameter: KERNEL_TARGET

Value: `Image.gz` *(default)*

Another element where different CPU architectures are inconsistent with each other is the name of the kernel file that is produced by the build. It is sometimes vmlinux, sometimes vmlinuz, sometimes bzImage, sometimes Image.gz... As far as I can tell, it's completely at the discretion of whoever is maintaining that architecture within the kernel, and of course everyone has their own preferences.

Parameter: TARGET_SWAP_DEVICE

Value: `/dev/vdb` *(default)*

If the target-side build has a storage device available for virtual memory, it can be specified as `TARGET_SWAP_DEVICE` and will be used if such a device is found. If the device doesn't exist, it won't cause any problems, though; and if the target *does* exist but has a filesystem or partition table or anything like that on it, it won't be touched.

Parameter: TARGET_SYSTEM_CFLAGS

Value: `-O2 -fno-omit-frame-pointer -mcpu=native` *(default)*

When building the programs and libraries that will comprise the final system, it is generally desirable to set the `CFLAGS` (for C) and `CXXFLAGS` (for C++) environment variables to a common value so that optimization flags are used consistently across the entire system. `TARGET_SYSTEM_CFLAGS` provides a way to do that.

The presence of the `-fno-omit-frame-pointer` flag deserves some additional comment. A frame pointer is a pointer to a stack frame; if GCC is told to store frame pointers, it uses a specific CPU register to store a pointer to the stack frame when making function calls. That register is then unavailable for other purposes, which can make code larger and less efficient, but facilitates "unwinding" the stack; this can be useful when trying to diagnose exceptions. The default behavior of GCC was to include frame pointers until GCC 8, and then changed to omit frame pointers.

Since the default build style for CBL targets 64-bit ARM architecture, it is important to set the default `CFLAGS` to include frame pointers: The AArch64 ABI was designed with the presumption that frame pointers would always be present. See https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84521 for some detail on this.

If you're doing a CBL build for a *different* target, and you don't plan to debug programs using `gdb` or something similar, you might want to omit `-fno-omit-frame-pointer` (and possibly add `-fomit-frame`

`-pointer`) so that the resulting programs and libraries are a little bit smaller and faster.

Parameter: TARGET_SYSTEM_MAKEFLAGS

Value: `-j8` *(default)*

The `make` program is used by most projects to automate their build and installation processes. Among many other things, `make` supports parallelism in builds: if you are using a computer with multiple CPUs and sufficient memory to support several simultaneous compiler processes, you can run `make` with a `-jN` command line option, or set the environment variable `MAKEFLAGS` to include such an option, and `make` will run up to `N` processes concurrently. This can speed up builds enormously!

For CBL, the degree of parallelism in build processes should not necessarily be the same on the host system and the target systems, because they may have very different hardware resources available. So to configure the number of concurrent build processes on the *host* system, simply set `MAKEFLAGS` as normal when running the host-side scripts. To configure the number of concurrent build processes on the *target* system, use this parameter!

For target builds that will run in a QEMU emulator, it may not make sense to specify any parallelism higher than `-j1`, because in many cases QEMU doesn't actually have the ability to run multiple emulated CPUs simultaneously. I've observed this to be the case, for example, when running an `x86_64` target computer on an `arch` host.

This parameter should generally align to the `TARGET_QEMU_CPUCOUNT` parameter, if the target will be a QEMU virtual machine.

Parameter: ENABLE_TARGET_NETWORK

Value: `true` *(default)*

The default presumptions made by CBL are that the target system has a wired ethernet interface, that there is a DHCP server available, and that networking should be enabled when the target system is booted. If any of those isn't the case, set `ENABLE_TARGET_NETWORK` to any value *other than* `true` — in that case, you'll need to set up networking yourself after the build completes.

Parameter: TARGET_BRIDGE

Value: `qemu` *(default)*

There's more than one way to get from the host side of the CBL build process to the target side. Each of these is defined in a blueprint named `target-bridge-NAME` (where `NAME` can be whatever you like); the one that is used for a particular CBL build is the one named in this parameter.

The default CBL process uses the real host computer system for the first part of the build, and an emulated QEMU virtual machine for the target. That's what the `qemu` target bridge does. Another option is to use QEMU virtual machines for both the host *and* target systems; the `automated-qemu-to-qemu-build` blueprint describes one way to do that.

Parameter: TARGET_MACHINE

Value: `normal` *(default)*

Under most circumstances, nothing extraordinary or unusual needs to be done to support particular target devices. However, some single-board computers like the Raspberry Pi require

extensive modifications to the Linux kernel to work properly. To support those computers without unnecessarily modifying the kernel for *other* systems, the blueprint for the Linux kernel has blocks that can be enabled based on the `TARGET_MACHINE` parameter.

If you want to use the CBL process to build an operating system for one of those devices, set this parameter appropriately.

2.3. Target System QEMU-related Parameters

Different parameters are required depending on whether the target system is a real computer or a virtual machine. The parameters in this section are used whenever the target system is a QEMU virtual machine.

If you're not using a QEMU virtual machine for the target, most of these are irrelevant — with one exception, `TARGET_QEMU_ARCH`, as noted in its description.

Parameter: TARGET_QEMU_ARCH

Value: `aarch64` *(default)*

The most fundamental of the QEMU-related parameters is QEMU's name for the target CPU architecture. Like the Linux kernel, the way that QEMU refers to machine architectures is *often* the same as the CPU field in the triplet — this is the case for 64-bit ARM machines, which both the GNU toolchain and QEMU call "aarch64" (for "ARM Architecture, 64-bit").

Even when the target system is going to be a physical computer rather than an emulated one, it's important to specify the correct value for `TARGET_QEMU_ARCH` — QEMU is used to validate that the cross-toolchain works properly, even if it's not used for the target-side build.

Parameter: TARGET_QEMU_MACHINE

Value: `virt` *(default)*

For most architectures, QEMU can emulate a variety of different machines. This parameter lets you select from those. This selection relates to the kernel configuration you start with, which is specified with the `KERNEL_CONFIG` parameter.

The best documentation for what machine types are supported for different architectures is on the QEMU wiki: https://wiki.qemu.org/Documentation/Platforms is the top-level URL as of October 2023. You can also run the QEMU full-system-emulator program (like `qemu-system-x86_64` or `qemu-system-aarch64` or whatever) with the argument `-machine help` to get a list of the machines it supports.

Parameter: TARGET_QEMU_CPU

Value: `cortex-a57` *(default)*

Similarly to the `machine` argument, QEMU can emulate a variety of CPUs; and you can get a list of the options here with `-cpu help`. In many cases you don't really need to specify a CPU because there will be a default value that works fine, but this is not the case with the `virt` machine that is used in the default configuration.

If the target QEMU system will be run as a native virtual machine rather than an emulator — that

is, if the actual computer is an x86_64 machine, and you're doing a build in an x86_64 virtual machine so you can use the computer for other things while the build is running — you can specify `-cpu host` to tell QEMU not to emulate a processor at all, and simply act as a hypervisor.

Parameter: TARGET_QEMU_CPUCOUNT

Value: `8` *(default)*

The QEMU full-system emulators can provide multiple CPU cores to the guest virtual machine. This may or may not actually be helpful in terms of performance — as mentioned earlier, when talking about `TARGET_SYSTEM_MAKEFLAGS`, this has historically only been helpful when QEMU is running as a hypervisor, not emulating a different machine architecture, because the TCG code generator that converts guest CPU instructions into host system instructions only ever operated in a single thread. Recent versions of QEMU have supported multi-threaded code generation (the "MTTCG" feature) for some architectures, which provides real parallelism within emulated machines. This speeds up builds dramatically for some packages, up to the limit of parallelism that QEMU emulated machines will accept. As of QEMU 4.1.0, the `virt` machine supported by the ARM emulator will accept up to eight CPU cores, so that's the value used by default.

You should set the level of parallelism used by `make` on the target system — that's the `TARGET_SYSTEM_MAKEFLAGS` parameter — to be the same as the number of CPU cores provided to the target virtual machine here.

Obviously, it's a bad idea to set this parameter higher than the number of CPU cores that the host system actually has!

Parameter: TARGET_QEMU_RAM_MB

Value: `8192` *(default)*

QEMU allows you to define the amount of RAM that will be made available to a virtual machine. The CBL process puts a fair amount of stress on the target system, and the amount of memory available to the compiler — especially for C++ builds — has a huge impact on the reliability of the process as a whole. The default value of 8 GiB works pretty well, but when the host system has more memory I always give the target as much as I can, up to about 24 GiB.

The `virt` machine type, available when using the ARM emulators, allows as much memory as you'd like to allocate.

Parameter: TARGET_QEMU_DRIVE_PREFIX

Value: `vd` *(default)*

The emulated hardware in QEMU virtual machines is not the same for all architectures and machines. Depending on what hardware is emulated, storage devices might show up as `sd` (SCSI) devices, `hd` (IDE or ATAPI) devices, `vd` (virtual) devices, or possibly something else entirely. This driver-defined prefix must be used when launching QEMU so that the Linux kernel can find the root filesystem, and is also used in QEMU builds when creating partition tables and filesystems.

The way drive prefixes are used in CBL correspond to a historical convention for the way that device files have been named: for SCSI disks, for example, the disks are referred to as `/dev/sda`, `/dev/sdb`, and so on; partitions on the disks are referenced as `/dev/sda1`, `/dev/sda2`, etc. This

convention is not always used, though; the convention for NVMe storage is for the devices to be named `/dev/nvme0n1`, `/dev/nvme1n1`, and so on; and for partitions on those devices to be `/dev/nvme0n1p1`, `/dev/nvem0n1p2`, and so on.

That means that if you're doing a CBL build using NVMe storage, or some other type of storage that uses a different naming convention than the historical one, you'll need to tweak the blueprints that refer to the `QEMU_DRIVE_PREFIX` parameters.

Parameter: TARGET_SERIAL_DEV

Value: `ttyAMA0` *(default)*

When building the target system in a QEMU virtual machine, the normal graphics console provided by QEMU is not used. Instead, we take advantage of QEMU's ability to map the standard input and standard output of the virtual machine process to a simulated serial device. The first serial device on most Linux systems is `/dev/ttyS0`, but for ARM computers it might show up as `/dev/ttyAMA0` instead.

2.4. Target Boot Configuration

Similarly to `TARGET_BRIDGE`, there's more than one way to make a GNU/Linux system bootable, and so there are multiple blueprints for doing that. These parameters are used to select which blueprint to use, and (for those that need additional configuration parameters) configure how it should work.

Parameter: BOOTLOADER

Value: `manual` *(default)*

The `BOOTLOADER` parameter selects the blueprint that will be used to set up the boot loader for the target system. The actual blueprint that will be used for this is named `setup-bootloader-` followed by this parameter; in the case of this specific build, for example, the blueprint is `setup-bootloader-manual`.

As with the target bridge, the `manual` option means you're on your own when it comes to making the target system bootable — the `setup-bootloader-manual` blueprint does not do anything.

Parameter: BOOT_DEVICE

Value: not set *(default)*

If a boot loader is being installed, it's a good idea to set BOOT_DEVICE to the name of the block special device that will be used by the boot loader. For example, the GRUB boot loader is typically installed on the first sector (also known as the "boot sector") of a storage device, where it can be found and loaded by the BIOS.

2.5. Target system name and login details

A non-root user account is always created on a CBL system. These parameters control the details that will be used for that account. It's almost certainly a good idea to override these parameters with values you prefer!

Parameter: LOGIN

Value: `lbl` *(default)*

This parameter controls the login name for the non-root user. It's a good idea to change this to your preferred login name. (The parameter name `USER` would be more idiomatic, but litbuild uses environment variables to override the default value for configuration parameters, and the `bash` shell always sets `USER` to the current user name — so using `USER` here would conflict with that behavior of the `bash` shell.)

Parameter: LOGIN_FULL_NAME

Value: `A Little Blue User` *(default)*

The UNIX user database has a "comment" field that, for accounts used by actual human users, is conventionally used for the full name of the user. Again, it's a good idea to change this to your own full name.

Parameter: DOMAIN_NAME

Value: `freesa.org` *(default:* `example.org`*)*

A domain name — preferably one that you control, defaulted here to `example.org` — will be used in various places throughout system setup and configuration.

Parameter: HOST_NAME

Value: `cblviton` *(default:* `cbl`*)*

The final target system will set its hostname to whatever is specified by this parameter (in the `DOMAIN_NAME` domain).

2.6. Directories Used For the Build Process

The rest of the parameters govern where different parts of the build artifacts will reside. You can set these however you like.

Parameter: QEMU_IMG_DIR

Value: `/tmp/cblwork` *(default)*

When targeting a virtual machine, the disk image files used by the QEMU emulator will be created in this directory.

Parameter: CROSSTOOLS

Value: `/tmp/cblwork/cross-tools` *(default)*

This sets the directory into which the cross-toolchain will be installed.

Parameter: HOSTTOOLS

Value: `/home/random/cbltools` *(default:* `/usr`*)*

This sets the directory into which the Trustworthy Host-System Programs will be installed, if they are being built. If these are not needed, this can be left at the default value of `/usr` — or, if the host

system has QEMU installed in a different location, whatever location that is.

Parameter: SYSROOT

Value: `/tmp/cblwork/sysroot` *(default)*

The "sysroot" framework is used for the cross-toolchain, and will contain the root filesystem that will be used by the target system. You can read much more about this in the various GCC sections. Initially, during the host stage of the CBL build, this will only contain a single subdirectory, `/scaffolding`. Absolutely everything else will be created in the target stage of the build.

Parameter: TARFILE_DIR

Value: `/home/random/materials` *(default:* `/tmp/cbl-materials`*)*

The source code for the software packages that make up the CBL system is distributed in files created by the `tar` program. `tar` stands for "Tape ARchive" — a term left over from bygone days, when magnetic tapes were the primary storage format used to move large amounts of data from one system to another. Even though tapes aren't commonly used any more, this is still used as the primary distribution format for source code on UNIX-ish systems.

This parameter sets the location where CBL will look for all of the source packages needed during the build.

Parameter: PATCH_DIR

Value: `/home/random/materials` *(default:* `/tmp/cbl-materials`*)*

Sometimes, the source code package that is distributed by a project needs to be modified or adjusted before it is built. This is generally done using the `patch` utility, and the files that contain descriptions of the modifications that need to be made are called "patch files."

This parameter sets the location where CBL will look for all of the patch files needed during the build.

Parameter: WORK_SITE

Value: `/tmp/cblwork/build` *(default)*

When building software from source code, you need to unpack the source code somewhere and then configure, build, and (sometimes) test it before installing it to its final destination directory. The `WORK_SITE` parameter specifies where all that activity will be done; the name comes from the construction metaphor that litbuild uses. It should be considered a transient or temporary directory, and can be deleted after the build is complete. The full CBL process can use twenty or thirty gigabytes of storage space, so to be on the safe side, define `WORK_SITE` as some location with at least that much free space available.

Parameter: LOGFILE_DIR

Value: `/tmp/cblwork/logs` *(default)*

Everything printed to standard output and standard error throughout the CBL build process will be written to log files; if something goes wrong, the main way to figure out what happened is to look at the log files.

This parameter specifies the location where log files are written during the host side of the CBL process. Log files written during the target side of the build will be on the target system, of course.

Parameter: SCRIPT_DIR

Value: `/tmp/cblwork/scripts` *(default)*

`SCRIPT_DIR` specifies the location where litbuild will write the bash scripts that automate the build story — the "tangle" side of the literate build system.

Parameter: DOCUMENT_DIR

Value: `/tmp/cblwork/docs` *(default)*

`DOCUMENT_DIR` specifies the location where litbuild will write an AsciiDoc document that tells the build story — the "weave" side of the literate build system.

[1] I didn't count them

Chapter 3. Packages and How To Build Them

The GNU system is a collection of *packages*. Each package includes some set of programs, libraries, configuration files, and other data files of innumerable types; collectively, all those files make up the system.

This is a pretty basic concept and you should feel free to skip ahead if you are familiar with these topics!

When people talk about a Linux system (or, equivalently, a GNU/Linux sytem), they're talking about a collection of software packages that have been assembled in a particular way, guided by some policy decisions about how to fit those things together. There are some common elements among all of these systems: they use the Linux kernel to manage hardware resources and provide services to userspace programs; there is some `init` program that runs those userspace programs (and, in many cases, keeps system programs running); there is a core set of userspace programs that you can reasonably expect to find on the system, like the `bash` shell and the core GNU utilities... aside from those basic elements, though, there is considerable variation in how different systems are set up, which packages are available, the mechanism used to set up additional programs, which `init` program is used, how the filesystem is arranged, all can vary widely from one system to another.

Many people and organizations provide easy-to-install *distributions* (sometimes called "distros") of those packages, policy decisions, and so on, and this is how the vast preponderance of GNU/Linux systems are set up — so much so that people generally talk about these systems in terms of which distribution was installed: RedHat, or Debian, or one of the hundreds of derivatives of those systems, or one of the newer independent distributions like Arch or Void Linux. There are a surprisingly large number of others! You can find a timeline of many of the distributions and the relationships between them on the Internet, perhaps here; if that link doesn't work, try doing an Internet search for "GNU/Linux distribution timeline." Alternatively, spend some time looking at the https://distrowatch.com/ site — they track releases and other activity on a lot of distributions.

The CBL process describes a GNU/Linux distribution, as well: if you follow the process described in this book, you will wind up with a Little Blue Linux system. If you modify the CBL process, then you'll wind up with your very own distribution — a derivative of Little Blue, just as Ubuntu is a derivative of Debian GNU/Linux.

But I digress. All of these systems are, primarily, a collection of software packages, each of which is maintained and released by a person or project team. Generally, these packages are released by their respective project teams as `tar` archive files containing the package source code and other files. Most of the work that goes into making a GNU/Linux distribution is in taking those release files and setting them up as part of the system, then repackaging the result so that it's easy for users of the distribution to set it up as well.

That process — setting up a new package so you can use it on your system — generally consists of four steps:

1. You *configure* the package for your specific system, with some set of configuration settings to control how it will be built and where its files will eventually wind up;
2. then you *compile* the package source code into executable programs and libraries;

3. then you run a suite of automated *tests* to verify that the program was built successfully and is working as expected; and, finally,
4. you *install* the package files into the system so that it is available for use.

Sometimes one or more of these stages is missing for a package — for example, a program may not have a test suite, or a package might be written in a language that is primarily interpreted, like perl or python, so there may not be a compile step — but that sequence of build stages is common enough that the instructions you'll find here always frame the process of setting up packages in terms of those steps.

3.1. The GNU Build System

Many of the packages that constitute the basic Little Blue system — including almost all of the most fundamental components, like the C standard library and software construction tools — are part of the GNU system created and maintained by the Free Software Foundation. These packages are generally designed to be constructed using *the GNU Build System*, which includes the autotools for configuration and `make` for running all the commands that actually compile, test, and install the package.

So many of the packages built during the CBL process use this build system that it forms the default sequence of steps used to set up software packages. If you look at the source blueprints that define the CBL process, you may notice that package blueprints often omit some or all of the commands used to set up the package. That's because the default sequence of steps can be used for them:

1. The package is configured using `./configure --prefix=/usr`, to use default configuration settings for everything except the location where the package files will be installed (the default for this is usually `/usr/local`, for reasons we won't go into here);
2. the package is compiled by simply running `make`, which causes the default target to be built;
3. the test suite is run with `make check`; and
4. the package files are installed with `make install`.

It's common for packages to be set up using one or two commands that differ from the defaults, so sometimes you'll see that there's an explicit definition for the configuration commands, or the test commands, or something like that.

3.2. Executing the CBL Process Manually, Part One

CBL is designed and intended for automation — if you take the source blueprints for CBL, set environment variables for all the configuration parameters you want to override, and run the `litbuild` program on them, you'll wind up with a shell script you can run to kick off at least the host side of the build; depending on which "target bridge" you use, the conclusion of that process might automatically kick off the entire second half of the build as well.

The reason for the focus on automated builds is simple: people are, generally, bad at being consistent. That means that any time you want to be able to repeat a process several times consistently and without mistakes, automation is a practical necessity — and I definitely want to be able to repeat the CBL process many times! I sometimes run through the CBL process multiple times

in a week.

On the other hand… you may have a different preference. If, for whatever reason, you prefer to type all the commands yourself, or copy and paste them from a PDF or web browser, you can do that. This section has a few tips to help with this.

Several parts of CBL set *environment variables* that control or influence how things are built. Those enviroment variables are scoped by the section structure of the process, so (for example) when you start the Constructing a GNU Cross-Toolchain section, you should set all the environment variables defined in that section (and explicitly `unset` any variables that say they should not be defined at all) before you start building any of the packages in that section. But when you've *finished* that part of the process, you should start a new shell process *without* those variables set.

Aside from those environment variables, every section in CBL either has some commands to run — which are hopefully pretty straightforward — or sets up a package. If you're following the process manually, here's how to set up each package:

1. Unpack the source tarfile. The archive file should always unpack into a new directory; `cd` to that directory.
2. If the blueprint specifies any *In-Tree Packages*, you should unpack the source tarfile for each of those packages, and move the resulting directory to the location specified in the blueprint.
3. If the blueprint specifies any *Patches*, you should apply those to the source tree using `patch -p1`.
4. If the blueprint specifies a *Build Directory*, you should create it and `cd` to it before proceeding.
5. Supposing all of that worked without errors, proceed by running all the Configuration commands, then the Compilation commands, then (optionally) the Test commands, and finally the Installation commands.

If anything goes wrong at any stage… well, that's a demonstration of why it's a lot of work to create and maintain a GNU/Linux distribution. Operating systems are complicated! Things break all the time, and you have to spend time and effort figuring out whether it's a real issue that has to be addressed or an ephemeral problem that will go away if you just restart whatever process failed.

In some cases, sections in CBL define files that should be written to the system at some specific location. These are sometimes written piecemeal throughout the text of a section, but there is always a subsection that provides the complete text of these files, at the end of that section. These should just be copied verbatim to the system, before running any of the commands defined in that section.

This process is also what the litbuild-generated scripts do automatically, up until the package-users framework is installed. At that point, the process you should use to build packages manually will also change — and we'll talk more about that when we get there!

Chapter 4. Patches in Cross-Building Linux

In CBL, we strongly prefer to stay as close as possible to the latest stable released version of every package. Sometimes that's not feasible or practical, though, for various reasons; in case you're not familiar with the idea of "patch" files, we'll talk a bit about what we do in those circumstances.

If you know all about patches, you might want to skip ahead!

UNIX systems have, since time immemorial,[1] included a userspace program called `diff`, whose purpose is to find differences between two files or two directory trees. This is really handy in all kinds of circumstances, as you can probably imagine! Any time you want to know what changed in a file, as long as you have a copy of the original version, you can use `diff` to find exactly what lines are different between the old and new versions. `diff` also provides options to include lines of context around the changes, and... really, lots of other things; you can read all about its capabilities with `man diff` or `info diff`.

Larry Wall, best known for creating the Perl programming language, wrote a program that is sort of the reciprocal or inverse of `diff`: the `patch` program. If you use `diff` to find all the differences between one set of files and another, and save that output to a file, you can then take that output file and feed it to the `patch` program, and `patch` will take the first set of files and transform it into the second set of files.

This is really handy when you want to distribute modifications to source code efficiently! Rather than creating an archive file with the entire modified source code directory, you can use `diff` to capture all of the differences between the original and modified versions of the source code; then you can distribute the output of the `diff` program however you need to. Anyone who has both the original version of the source code and the `diff` output can use `patch` to reproduce your modified version of the code.

Since you use the `patch` program you use to apply these changes, it's common to refer to the output of `diff` as "patches" or "patch files," and it's common to refer to the process of applying those files as "patching."

In CBL, we use a few different types of patches, described below. In most cases, we consider these patches to be a part of the CBL project itself, so they are maintained in a git repository you can find at `http://git.freesa.org/freesa/cbl-patches`, as well as being available in the CBL file repository at `http://files.freesa.org`.

4.1. Miscellaneous tweaks or fixes

Sometimes, packages just don't work the way that we would like them to. For example, when cross-compiling the GNU binutils for some host/target pairs, GCC issues a string truncation warning when compiling `gas/config/tc-i386.c`. This causes the binutils build to crash, since it's set to treat all warnings as errors. Modifying a `snprintf` call to use the correct formatting spec (`%hhx` rather than `%x`) resolves the problem, but the binutils maintainers don't want to simply make that change because of concerns that it might have a negative impact on some of the (many) architectures that binutils supports.

In this kind of situation, CBL applies a patch to make the necessary change.

4.2. Kernel configurations

The Linux kernel has to be configured to include support for whatever features and hardware device drivers are needed, and omit support for features and device drivers that are not desired. This is a pretty complicated process and can be hard to automate.

The configuration system provides a way to start with a named configuration rather than the default settings for an architecture; if this feature is used, a file from `arch/*/configs` is used to override the default settings.

CBL provides kernel configurations for some specific cases — for example, a configuration for kernels intended to run on EC2 instances in the Amazon Web Services cloud. These default configuration files are added to the Linux source tree via patches.

All of this is described more thoroughly in the Linux sections, so you can consult those for more information.

4.3. Gnulib updates

The GNU project maintains a repository of source code that is intended to be used in other software packages. This repository is called "Gnulib." However, unlike other packages referred to as "libraries," like `libxml2` or `libgcrypt`, Gnulib is *not* compiled into a library of functions that are then dynamically linked into other programs at runtime. Rather, the expectation is that files from the Gnulib repository will be *copied into* the source tree of other projects.

This sometimes presents challenges.

An example of this is release 2.28 of the `glibc` package. This release of `glibc` removes some obsolete and deprecated header files — an example is `libio.h` — that are not a part of the C standard library but were part of earlier `glibc` releases. These header files are not referenced by Gnulib — but until sometime in 2018, they *were*. Since other packages, like `m4` and `gzip`, still include those old versions of the Gnulib files, those packages won't build on systems that use glibc 2.28 or later. At least, not until new versions of *those* packages, using modern versions of the Gnulib components, are released! And some packages are not released very often: as I write this, for example, it's October of 2023, and the most recent version of `m4` is 1.4.19, released in May of 2021 — over two years ago.

When I encounter this situation, my practice is simply to compose a patch by copying in the newest version of whatever Gnulib source files have compilation issues.

4.4. Branch-update patches

The most common — and least objectionable — type of patch used in CBL is the "branch-update" patch.

All large and complicated software packages, like GCC and the GNU C library, have bugs. Some of those bugs are, inevitably, severe. A common practice for project teams is to maintain bugfix

branches in their source repositories for at least the most recent few releases of their package(s); as bugs are found and fixed, these changes are back-ported from the current main line of development to these bugfix branches.

It's my practice to pull in all the updates from these bugfix branches from time to time, and apply those as patches so that the CBL system is as stable and bug-free as possible.

Any time you see a package with a patch called `branch-update-` along with a year-month-day datestamp in its name, that's what it is: a compilation of all changes from the upstream project's bugfix branch, as of whatever date is indicated by the patch file name.

Unlike other CBL patches, these branch-update patches are not tracked in the `cbl-patches` git repository, although they are present in the CBL file repository. The rationale for this is that it's trivial to reproduce these patches from the upstream project's version control repository; all you have to do is obtain that repository and then use a command like `git diff glibc-2.28..remotes/origin/glibc-2.28/master` to produce a current branch-update patch for glibc 2.28.

[1] In other words, I don't personally remember ever seeing a UNIX system without it.

The Host-Side Build

Chapter 5. Preparing For The Build

Before we can start the CBL build process, there are a couple of things we need to make sure of.

5.1. Required Host-System Programs

First, and most importantly, it's important to make sure you have all the tools you need on the host system. Many GNU/Linux systems are missing one or more of the programs necessary for CBL, or provide a version of those programs that won't work properly for one reason or another. So, although it's not an intrinsic part of the build process *per se,* CBL includes the Trustworthy Host-System Programs appendix for building the programs that we've found to cause problems later on.

The basic set of requirements from the host system include modern versions of the GNU toolchain (GCC, binutils) and build system (`make`, the autotools, and so on), the QEMU emulator, and the `lzip` compression program. The `file` program is also necessary, and must be the same version that is used in the CBL process. If you're unsure of whether you have everything you need, please do check the Trustworthy Host-System Programs appendix and make sure you have the things built there.

If you're following the CBL process on a LB Linux system, this is probably unnecessary — all of the packages built in the host-prerequisites appendix are also built in the CBL process and are part of the basic LB Linux system. The only gotcha in that case is the file package, which can only be cross-compiled on a system that has the *same version of* `file` *installed already*. So if you're using an LB Linux system but the version of `file` installed there is older than the one that the CBL process currently uses, you'll need to upgrade that package before you can proceed.

5.2. Final Preparations For The Build

The whole first part of the CBL process — the part that runs on the host system — will need to find the trusted host system programs (if they were built) and the cross-toolchain programs, so we should make sure they are on the `PATH`. We also need to ensure that shared libraries installed as part of those packages can be found by those programs — if you're not clear on what that last part means and don't feel like being patient, you can skip ahead to A Word About The Dynamic Linker, where shared libraries are discussed.

Environment variable: PATH

```
/tmp/cblwork/cross-tools/bin:/home/random/cbltools/bin:$PATH
```

Environment variable: LD_LIBRARY_PATH

```
/home/random/cbltools/lib:$LD_LIBRARY_PATH
```

Litbuild provides a feature that allows the scripts it generates to be re-run if the build crashes partway through, by recording which scripts have completed successfully and skipping them in future runs. To activate this feature, we define an environment variable `LITBUILDDBDIR`.

Environment variable: LITBUILDDBDIR

```
/tmp/cblwork/cross-tools/litbuilddb
```

Now, at last, we're ready to start the CBL build process.

Chapter 6. Constructing a GNU Cross-Toolchain

This section describes how to build a GNU cross-toolchain that runs on computers with one CPU architecture — for example, x86_64 — and constructs programs that will run on a different CPU architecture, like MIPS or ARM. As usual in the CBL process, we're going to use modern versions of all the toolchain components; in this case, that means:

Package Versions:

- binutils 2.41
- gcc 13.2.0
- glibc 2.38
- gmp 6.3.0
- isl 0.26
- linux 6.6.1
- mpc 1.3.1
- mpfr 4.2.1

6.1. Toolchain Basics

A "toolchain" is the set of all of the programs and libraries required to transform source code into executable programs that you can actually run. It's called a "chain" because there are multiple programs involved: each program takes some kind of input file and produces some kind of output file, which then becomes the input for the next program in the chain. Each program is like a link in the chain. (When you think about it that way, it's not really the best metaphor. It's really a lot more like an assembly line! But "toolassemblyline" sounds awful.)

In this case, we are building a C and C++ toolchain: it will be able to construct executable programs from C and C++ source code.

The toolchain being built here consists of: a *preprocessor*, which handles include directives and macro calls and things like that; a *compiler*, which takes C or C++ source code and translates it into assembly language code; an *assembler*, which takes that assembly code and translates it into binary object code; and a *linker*, which combines object code produced by the assembler with additional object code contained in libraries and produces executable programs. The process also requires an implementation of the C and C++ standard libraries, which contain a large collection of functions that the linker uses when producing programs. In many cases, the implementation of those functions involves making *system calls*, which are basically functions provided by the operating system kernel; because the standard libraries make use of system calls, the process of building a toolchain also requires the kernel header files that specify what system calls are available and how to invoke them.

In the GNU toolchain, the preprocessor is `cpp` and the compilers are `gcc` (for C source code) and `g++` (for C++ source code). All of these — along with the standard C++ library — are contained in the

GNU Compiler Collection (gcc) distribution. The assembler and linker are `as` and `ld`, and are provided by the GNU binutils ("binary utilities") distribution. The C standard library is distributed separately as the glibc (short for "GNU libc") package, and the kernel header files are part of the Linux kernel distribution.

Since glibc is a rather large library — over a hundred megabytes of source code, producing shared library files that are in the megabytes — it is not very well-suited to building programs that must fit in a compact space such as the flash chip that holds the firmware in wireless routers. For those programs, an alternative C library, such as musl or uClibc, may be more appropriate. Toolchains using those C libraries can be produced using a variant of these instructions.

6.2. Cross-Toolchains

A "cross-toolchain" is much like a normal, or "native," toolchain. The difference is that a cross-toolchain runs on a computer of one type (the "host" system) but builds programs that will run on a different type (the "target" system). For the CBL project, for example, we use common Intel-processor computers to build software that will run on an ARM-architecture CPU. That way we can be reasonably certain that the final system really is entirely built from source, and no binary code was simply copied from the original build system to the target system: code from the build system simply can't run on the target system, so if any binary code from the host system winds up on the target system, it won't work at all.

Faux Cross-Toolchains

This same process can be used to build a "faux" cross-toolchain: a set of tools that runs on (for example) Intel computers and builds programs that will also run on Intel systems, but using the cross-toolchain build process instead of building a native toolchain.

That may seem pointless, but this can be a useful technique if you want to be *pretty sure* that you are building all system components from source code and that those components are entirely independent of the original build system. This is also a handy technique if you're starting with a non-multilib 64-bit system and want to build 32-bit programs, or something of the sort.

You can also use this approach simply to test that the cross-toolchain build process works properly without needing a second computer or an emulator to execute programs built using it.

CBL is not designed to use "faux" cross-toolchains because I prefer to be *really* sure that everything is getting built entirely from source code.

To build a GNU cross-toolchain, we pass `--host`, `--target`, and `--build` options to the `configure` scripts (produced using the autotools programs of the GNU build system) of the toolchain components.

As you may recall from Configuration Parameters and Default Values, the values assigned to these options are called "target triplets," a term that also comes from the GNU build system. A triplet is a string with multiple components that are separated by hyphens. Historically, the triplet has had

fields for CPU, manufacturer, and operating system (e.g., `mipsel-pc-gnu` for a little-endian MIPS CPU, PC hardware, and the GNU operating system); more commonly, these days, the OS field is subdivided into two fields, "kernel" and "system," so for all intents and purposes the target triplet winds up having four components: `cpu-manufacturer-kernel-os` (e.g., `mipsel-pc-linux-gnu`). That means that the term "target triplet" is kind of obsolete and misleading, and it would make more sense to refer to them as "target quadruplets." But sometimes history wins over accuracy and clarity.

The manufacturer field is basically freeform; in many cases it's just set as `pc` for IBM PC-architecture systems, or left as `unknown` or `none`. In CBL we set the manufacturer to `cbl`, because it's shorter than `unknown` and is distinctive: any time you see a triplet like `arm-cbl-linux-gnu`, you can be fairly confident it was produced using the CBL procedure.

The manufacturer field can be omitted entirely, but that makes the whole situation much more complex and ambiguous: for example, in the triplet `arm-linux-gnu`, is `linux` the manufacturer or is it part of the `os` field? The GNU build system includes a script called `config.sub` specifically to take triplet strings and figure out what they mean. My advice is: *always* specify triplets as cpu-manufacturer-kernel-os.

The other components of the triplet — cpu, kernel, and OS — sometimes trigger specific behavior, especially during GCC builds. It's a good idea to review the "Host/Target specific installation notes for GCC" section of the GCC build instructions when choosing a triplet for your system.

The GNU build system includes a script, `config.guess`, that tries to figure out what the host triplet is. Generally, this should be used as the `HOST` configuration parameter in CBL. You can always find an up-to-date copy of `config.guess`, and the related script `config.sub` mentioned above, in the GCC source code distribution.

Different combinations of those configure directives, `host`, `target`, and `build`, are used for different toolchain components and at different points in the CBL process. What they mean is:

`build`
: the system where the toolchain components are built

`host`
: the system where the toolchain components will run

`target`
: the system where the resulting artifacts will run

When `build`, `host`, and `target` are all different, it's called a "Canadian Cross." We're not sure why. In CBL, we don't build any Canadian Crosses. During the CBL process, we:

1. build a native compiler unless we already have one that we can definitely trust; for this one, of course, we don't need to specify `host`, `build`, or `target`;
2. use the trusted native compiler to build a cross-toolchain (at this point, `build` and `host` are both that of the initial system, and `target` is the target system type);
3. use that cross-toolchain to build a target-native toolchain as well as a collection of programs that will be needed on the target system (at this point, `build` is the initial system, and `host` and

`target` are both the target system type); and finally

4. boot into the minimal target system userspace we constructed in the previous step, and use the target-native toolchain built there to construct a new, testable, native toolchain (for which we again don't need to specify `build`, `host`, or `target`).

Once the cross-toolchain is built, most packages built using it will be configured with `build` set to the host computer and `host` set to the target computer.

6.3. The CBL Sysroot Toolchain

The cross-toolchain built here uses the *sysroot* framework. The idea behind a sysroot toolchain is simple: a directory on the host system (the "sysroot" directory) is set up to contain a subset of what will eventually become the root filesystem of the target system: `/bin`, `/lib`, `/usr/lib`, `/etc`, that sort of thing. Header files and libraries will be used only from the sysroot location, not from the normal host system locations.

This is fine for most cross-compiling purposes, but it's not quite perfect for our purposes in CBL.

Remember that the whole point of this cross-toolchain is to build a bootable GNU/Linux system — the kernel, and a minimal set of userspace programs that we'll then use to construct the final system. Those components, the "scaffolding," won't be used any longer than is necessary. This is because we don't *want* to rely on the programs produced by the cross-toolchain. By preference, for any program that is distributed with a test suite, we want to run the test suite before we assume it works properly! When you're cross-compiling programs, you can't easily run the tests.

So once we have the target system booted, we only use the scaffolding to build the first parts of the final system. As we build those, we install them into the canonical filesystem locations where they belong: the standard directories `/bin`, `/lib`, `/usr`, and so on. To make that process as straightforward as possible, and avoid any interference between the scaffolding and the final system components, we want all the scaffolding to wind up in a `/scaffolding` subdirectory of the root filesystem. If all of the ephemeral stuff is self-contained within `/scaffolding`, then obviously it won't conflict with the final system programs as we build and install them.

Setting up everything so that it's self-contained within a non-standard location also makes it easier to ensure that our final system doesn't still rely on any of the components there: once the full system build is complete, we can simply delete the `/scaffolding` directory and we'll be left with just the Little Blue Linux system.

When building a toolchain, it's important to simplify the execution environment as much as possible; any unncessary compiler or linker flags can cause things to break. Don't worry about optimizing anything for this stage, either: remember, everything we're building at this point is ephemeral, and we'll be throwing it away as soon as we can.

Environment variable: PATH

```
/tmp/cblwork/cross-tools/bin:$PATH
```

Environment variable: LC_ALL

```
POSIX
```

Environment variable: CFLAGS

(should not be set)

Environment variable: CXXFLAGS

(should not be set)

Environment variable: LDFLAGS

(should not be set)

Environment variable: LD_LIBRARY_PATH

`/home/random/cbltools/lib`

This is as good a point as any to discuss the dynamic linker and the way it works.

6.4. A Word About The Dynamic Linker

In most cases, executable programs on Linux systems are linked against *shared libraries* rather than *static libraries*. That means that programs don't contain a copy of the binary code for library functions they invoke; instead, programs have references to those library funtions, and have a list of the shared libraries that are expected to contain the implementation for those functions.

Whenever a program is executed, those references to library functions obviously must be resolved — that is, the shared libraries are searched for all the functions needed by the program (and, since those functions might call other library functions as well, libraries are searched for *those* functions as well, and so on recursively), and linked together so that all of the necessary functions are available. This resolution is done by the *dynamic linker*, also known as the *dynamic loader* and, sometimes, as the *program interpreter*. This is a program included with the GNU C library — or whatever other C library is being used, like musl — and is conventionally installed at `/lib/ld.so` or `/lib/ld-linux.so`.

The way that the dynamic linker finds shared library files is a bit complicated, and that complexity is at the root of a lot of the issues that can come up during the CBL build process, so let's talk about it a bit!

6.4.1. The tl;dr

Here's a summary of the basics, for anyone who doesn't need all the grueling details:

- There are a bunch of standard system directories that are always used when looking for shared library files — directories like `/lib` and `/usr/lib`. If you add a directory to `/etc/ld.so.conf`, it basically becomes one of those system directories.
- If you have library files in a different directory, you can get ldconfig to look there by setting an environment variable, `LD_LIBRARY_PATH`.
- If, when you're building a program, you know that the program will need some shared libraries and those libraries will not be in one of the standard system locations, you can make those directories a part of the library search path for that program by giving `ld` the argument `-rpath /whatever/dir:/another/dir`. This sets an `RPATH` for the program, which overrides `LD_LIBRARY_PATH`: the `RPATH` will be used before the directories in `LD_LIBRARY_PATH` are checked.

- If you want to do something *like* `RPATH`, but you want to make it easy for people who run the program to override the library search path, you can add the `ld` option `--enable-new-dtags` in addition to the `-rpath` option. This will cause `ld` to set a `RUNPATH` rather than an `RPATH`; `RUNPATH` is checked for shared library files *after* `LD_LIBRARY_PATH` is checked.

This all matters for CBL because the build process for some of the necessary packages will result in programs that can't find their shared libraries unless we use an `RPATH` or `RUNPATH`; and the build process for other packages sets an incorrect `RPATH` or `RUNPATH` that can cause problems unless we remove it.

Usually, `ld` isn't executed directly by build processes, but is instead invoked by the `gcc` driver program. To get `gcc` to pass an option along to `ld`, you can give it the option `-Wl`, which should be followed by additional words separated by commas; `gcc` will replace the commas with spaces and pass the resulting arguments on to `ld`. So to set an `RPATH` of `/some/dir` and `/another/dir`, you can give `gcc` the argument `-Wl,-rpath,/some/dir:/another/dir`.

An alternative that might work is to set the environment variable `LD_RUN_PATH` to the desired `RPATH` before linking — the `ld` documentation suggests this will work, but I haven't tried it.

6.4.2. The grueling details

This might make your eyes glaze over a bit.

A lot of this discussion pertains only to programs in the Executable and Linkable Format, commonly abbreviated as ELF. This is the only program format used in modern GNU/Linux systems, but you should be aware that there *are* other executable program formats, like a.out and COFF, and a lot of the details here don't apply to those formats.

ELF programs consist of an ELF header followed by some number of segments of various types. If you'd like to examine the full structure of an ELF program, you can use the program `readelf`, which is part of the GNU binutils package.

When a program is executed, the Linux kernel looks in its `.interp` section to find the path for its interpreter — in our case, this will be the full path of `ld.so`. That interpreter then looks in dynamic segments for the names of shared library files that are needed by the program and tries to find them. If any shared library can't be found, the dynamic linker will terminate with an error.

There's a bit of additional complexity: shared library files are *also* in ELF format and can *also* declare that they are dependent on other shared libraries. This can result in a chain of library dependencies, potentially a lengthy one! When the dynamic linker is trying to find a library, its behavior is partly determined by which object is having its references resolved: the original program, or one of the libraries it depends on, or one of *their* libraries, and so on. Whichever ELF file is having its references resolved is called the "loading object" here.

The rest of this section describes the procedure used by the dynamic linker to locate shared library files.

Shared library names *can* contain slash characters, although they usually do not. (I don't even know how to get `ld` to create a program with library dependencies that specify a full path.) If a shared library name *does* contain any slash characters, it is treated as a relative or absolute path and the

dynamic linker will only look for the library at that path.

In the common case, when a shared library is specified without any slash characters, the dynamic linker looks for it in a variety of locations. The algorithm it uses is poorly documented and there's a lot of contradictory information about it on the web. After looking around for quite a while, I found a helpful blog post on `qt.io` that asserted that the relevant code is `_dl_map_object` in the glibc file `elf/dl-load.c`, and a review of that code revealed the following:

1. If the loading object has a `RUNPATH`, skip ahead to the `LD_LIBRARY_PATH` step.
    a. Look in the `RPATH` of the loading object, if any.
    b. Consider the thing that loaded the loading object. If *it* has a `RUNPATH`, skip ahead to the `LD_LIBRARY_PATH` step. If not, look in its `RPATH`.
    c. Continue doing this recursively up the loading chain (skipping ahead if you find a `RUNPATH`, looking for libraries in `RPATH`) until you reach the end of the loading chain. This will normally be the program being executed, but could also be a shared library loaded using the `dlopen` function.
2. Look in the `LD_LIBRARY_PATH` environment variable.
3. Look in the `RUNPATH` of the loading object, and once again look up through the loading chain until you reach the end.
4. Look in the locations found in `/etc/ld.so.cache`, which is generated from `/etc/ld.so.conf` using the `ldconfig` program.
5. Finally, look in the default directories defined when glibc was compiled; when using the standard build process, this means `/lib` and `/usr/lib`.

As soon as the shared library is found in any of those locations, that's the version that will be used. If it's not found in any of those locations, the dynamic linker will give up and crash with an error message. And, in case this was not obvious, in all the things that look like `PATH` — `LD_LIBRARY_PATH`, `RPATH`, `RUNPATH` — you can specify any number of directories separated by colons, just like the `PATH` environment variable.

A historical note: `RPATH` has been around longer than `RUNPATH`. `RUNPATH` was implemented, along with all the logic about skipping `RPATH` when a `RUNPATH` is present, because people realized that when someone specifies an `LD_LIBRARY_PATH` it's usually because they really do know what they want the dynamic linker to do, and it's rude to override that desire at program compilation time.

When you give `ld` the `-rpath` option by itself, it just creates an `RPATH` in the resulting program or library. When you add the option `--enable-new-dtags`, it still creates an `RPATH` (in case you run the program with an old dynamic linker that doesn't understand `RUNPATH`), but it *also* creates a `RUNPATH` so that modern dynamic linkers will ignore the `RPATH`.

As a reward for reading this far, here's one last option you can use if none of the above suits your purpose: before it does anything else, the dynamic linker loads any `.o` object files, or `.so` or `.a` library files, that are named in the environment variable `LD_PRELOAD` or in the `/etc/ld.so.preload` file. Any functions defined in those files are used in preference to any other function definitions found later in the process. That lets you override individual function definitions, if you want to override some part of the program without replacing an entire library with a different version.

You can read more about all this stuff in the `ld` and `ld.so` info and man pages, and in the "Program Library HOWTO" in the Linux Documentation Project.

6.5. binutils

Name	GNU binary utilities
Version	2.41
Project URL	http://www.gnu.org/software/binutils/
SCM URL	git://sourceware.org/git/binutils-gdb.git
Download URL	https://ftp.gnu.org/gnu/binutils/
Patches	• `binutils-2.41-branch-updates-20231103.patch`

6.5.1. Overview

The GNU binary utilities package contains a plethora of programs and libraries that can be used to produce, manipulate, and otherwise operate on (compiled and assembled) object files. I'll briefly describe them all here, but don't worry! Not only will there not be a test on any of this, you won't usually need to invoke any of these programs manually. The `gcc` driver program will invoke `as` and `ld` as necessary to do its work, and several of the other utility programs are similary used internally during the build process of other system components but you won't need to use them yourself.

The most important binutils programs are `as`, the *assembler*, which transforms assembly source code (`.s` files) into binary or "object" code (`.o` files); and `ld`, the *loader* or *link editor* (`ld` is usually just called the *linker*, but it's hard to see why it's called `ld` without knowing the other terms), which combines multiple object files into an executable program.

There are actually two different `ld` programs in current GNU binutils: the original version uses the binary file descriptor (BFD) library and can always be found at `ld.bfd`. There's also a newer program, "gold," that doesn't use BFD and only works for binaries that are in the "Executable and Linkable Format," aka "ELF"; it's always available at `ld.gold`. A hard link to one or the other of those programs is created by the installation process so it can be executed simply as `ld`.

The other programs in this package are:

- `addr2line` translates program address locations to filenames and line numbers, which can be helpful during debugging.
- `ar` can be used to create, modify, and extract files from archives.
- `c++filt` converts the mangled function names found in compiled C++ programs back to the original un-mangled names.
- `gprof` lets you run programs with instrumentation that tells you how much time is spent in different parts of the code, which can be helpful when optimizing programs.
- `nm` lists symbols found in object files.
- `objcopy` can translate object files to various alternative formats.

- `objdump` is a disassembler; it can convert binary files into a canonical assembly language.
- `ranlib` generates indexes for archive files.
- `readelf` shows information about ELF-format object files.
- `size` displays sections of an object or archive, along with their sizes.
- `strings` prints out printable character sequences found in binary files.
- `strip` discards symbols or other unnecessary data from object, library, or program files, which reduces their size but makes them much harder to debug.

There are a couple of shared libraries used by those programs and available to others, as well: `libbfd` and `libopcodes`. All of the utility programs are documented much more thoroughly in man pages and the binutils info file.

As I mentioned earlier, you don't need to run any of those programs directly because the `gcc` driver program streamlines the simple case of building executables — to compile a "Hello, World" program, you just need to run `gcc hello.c`, and let the gcc driver program run `as` to assemble compiled source into object files, `ld` to link multiple object files together to produce an executable, and maybe other programs if it needs to for some reason.

The downside of using a driver program is that it can make complex builds (when, for example, specific options need to be passed to the assembler, compiler, and linker) a lot more complicated and fussy — as you'll see, from time to time, during the CBL process.

Building From The Source Code Repository

The source code for this package is in the same repository as the GNU Debugger, `gdb`. I believe this is a historical artifact of the olden times, when the primary source code management system used for those and other packages was CVS, but for whatever reason it is still the case today.

If you want to build binutils from the git repository, it's therefore a good idea to extract just the source code corresponding to that package from the combined repository. Conveniently, the repository contains a script that will do that for you.

After you've checked out the specific revision you want to build, simply run `./src-release.sh binutils` and you'll wind up with a tar archive file `binutils-$VERSION.tar` in the top directory of the source tree.

You can similarly construct tarfiles for the `gas`, `gdb`, or `sim` packages by passing those arguments to the `src-release.sh` script. (I don't know why you'd want to build a standalone `gas` package, since `gas` is the GNU assembler and is part of the binutils package. And I don't know what `sim` is at all. But the options are there, regardless.)

Patch:

- binutils-2.41-branch-updates-20231103.patch

The combined-source-repository situation described a moment ago also complicates the

construction of branch-update patches a little bit: we always produce branch-update patches by running `git diff` commands against the source repository, and sometimes the resulting patch for binutils includes changes to files that are not part of the binutils distribution — which means they don't apply cleanly. We adjust for this by applying the patch, fixing any issues we find as a result, and then using the corrected patch here.

6.5.2. binutils (gnu-cross-toolchain phase)

Build Directory	`../build-binutils-2`

Binutils, like some other parts of the toolchain, should be built in a separate directory from the source code. As with other components that we build several times, CBL puts each distinct build in a separate location so that each build is entirely independent of, and unaffected by, the other builds.

Build Directory

`../build-binutils-2`

We build a cross-binutils using the "sysroot" framework (which you'll read more about shortly). That framework isn't particularly well-documented, but the important thing at this point is that we need to specify the configure options `--with-sysroot` and `--with-build-sysroot` to inform the build machinery of the desired sysroot directory.

All of the toolchain components should be installed in the same filesystem location. The `CROSSTOOLS` parameter lets you specify that location — in this case, `/tmp/cblwork/cross-tools`.

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/tmp/cblwork/cross-tools \
  --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=aarch64
-cbl-linux-gnu \
  --with-sysroot=/tmp/cblwork/sysroot --with-build-sysroot=/tmp/cblwork/sysroot \
  --disable-nls --enable-shared --disable-multilib \
  --with-lib-path=/tmp/cblwork/sysroot/scaffolding/lib \
  --enable-64-bit-bfd
make configure-host
```

Some of the warning messages present in GCC 8 and later present problems when compiling the binutils. We can tweak the generated Makefiles so those warnings won't be converted to errors.

Configuration commands:

```
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=stringop-truncation@' bfd/Makefile
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=stringop-truncation@' gas/Makefile
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=format-overflow@' binutils/Makefile
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

6.6. gmp

Name	GNU Multiple Precision arithmetic library
Version	6.3.0
Project URL	https://gmplib.org/
SCM URL	*(unknown)*
Download URL	*(unknown)*

6.6.1. Overview

GMP, the GNU Multi-Precision arithmetic library, is a component in — or perhaps it is more properly called a dependency of — the GNU toolchain. It allows arithmetic operations to be performed with levels of precision other than the standard integer and floating-point types. Applications can use GMP to provide arithmetic with thousands or millions of digits of precision if that's what they need. GMP also provides support for rational-number arithmetic, as well as integer and floating-point.

If you're really interested in high-precision floating-point arithmetic, you might want to look into MPFR rather than GMP! The GMP people say it's much more complete.

GMP has been needed by the Fortran GCC front-end for some time, but starting with release 4.3.0 of GCC it is needed for C (and C++) as well.

MPC, another dependency of GCC, requires a GMP built with C++ support, so we need to specify that at configure time.

This package is often built in-tree as part of GCC, rather than separately — that's especially true when the only reason you're buiilding GMP is because GCC requires it. When using an in-tree build, this blueprint is pretty much irrelevant. However, as of 2015-09-27, there's an issue with in-tree builds of GMP in some cross-toolchain builds, so for CBL we build it separately.

6.6.2. gmp (gnu-cross-toolchain phase)

This is not necessary, because the system or host-prerequisites version of GMP can be used just fine by the cross-compiler. If things break down here and you haven't built the Trustworthy Host-System Programs, you should probably do that.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
(none)
```

6.7. mpfr

Name	MPFR library
Version	4.2.1
Project URL	https://www.mpfr.org/
SCM URL	*(unknown)*
Download URL	https://www.mpfr.org/mpfr-current/#download
Dependencies	gmp

6.7.1. Overview

MPFR is a library for arbitrary-precision floating-point arithmetic. It stands for "Multiple-Precision Floating-point Rounding," I think, but that's not really clear from their site so I might be wrong. It uses GMP internally, but provides any level of precision (including very small precision) and provides the four rounding modes from the IEEE 754-1985 standard.

Like GMP, MPFR is a component in, or dependency of, the GNU toolchain. It has been needed by the Fortran GCC front-end for some time, but starting with release 4.3.0 of GCC, MPFR is needed for C and C++ as well. GCC uses MPFR to pre-calculate the result of some mathematic functions when those functions have constant arguments, and produces the same results regardless of the math library or floating point engine used on the runtime system. This occurs in what is called the GCC "middle-end," which is kind of a silly name since it's not an end.

This package is often built in-tree as part of GCC, rather than separately. However, as of September 2015, in-tree builds of the dependencies have some issues in certain circumstances, so in CBL we step away from the in-tree build facility altogether.

6.7.2. mpfr (gnu-cross-toolchain phase)

As with GMP, this is not necessary because the system or host-prerequisites version of MPFR can be used.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
(none)
```

6.8. mpc

Name	GNU Multiple Precision Complex library
Version	1.3.1
Project URL	https://www.multiprecision.org/
SCM URL	*(unknown)*
Download URL	https://www.multiprecision.org/mpc/download.html
Dependencies	gmp, mpfr

6.8.1. Overview

MPC is a C library for arbitrary-precision arithmetic on complex numbers providing correct rounding. It can be thought of an extension to the MPFR library.

Like GMP and MPFR, MPC is a component in, or dependency of, the GNU toolchain. I haven't been able to find any description of what features of MPC are actually needed by the GNU toolchain, but MPC is a hard build-time dependency of GCC.

If your system already has MPC installed, this step can be skipped.

Like the other GCC library dependencies, this package is often built in-tree as part of GCC. As mentioned earlier, though, this sometimes introduces problems, so CBL doesn't take advantage of the in-tree build machinery provided by GCC.

6.8.2. mpc (gnu-cross-toolchain phase)

As with GMP, this is not necessary if there is a system or host-prerequisites version of MPC available.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
(none)
```

6.9. isl

Name	Integer Set Library
Version	0.26
Project URL	http://isl.gforge.inria.fr/
SCM URL	git://repo.or.cz/isl.git
Download URL	https://libisl.sourceforge.io/
Dependencies	gmp

6.9.1. Overview

The project homepage describes ISL as a library for manipulating sets and relations of integer points bounded by linear constraints. I'm not sure what that means. It sounds like math.

Unless you have some specific reason to want ISL for one of your own projects, the main benefit it provides is as an optional dependency of GCC: the "graphite loop optimizations" in GCC (whatever those are) require ISL to be available.

6.9.2. isl (gnu-cross-toolchain phase)

As with GMP, this is not necessary if there is a system or host-prerequisites version of ISL available.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
(none)
```

6.10. linux

Name	Linux kernel
Version	6.6.1
Project URL	http://www.kernel.org/
SCM URL	https://git.kernel.org/
Download URL	http://www.kernel.org/
Patches	• `linux-6.6.1-aws-ami-config-1.patch`

6.10.1. Overview

The kernel is Linux *per se* — the foundation of the operating system. Often, when people say "Linux," they mean the entire operating system that lets them use a computer; properly, though, Linux is just the kernel. Many — perhaps most — of the other components that make up the full operating system are pieces of the Free Software Foundation's GNU project, which stands for "GNU's Not UNIX"; almost all of the program-construction tools that form the foundation of the system are GNU components.

Linux or GNU/Linux?

This fact — that the foundation of the system consists of components from the GNU system — is the basis of the Free Software Foundation's suggestion that the operating system as a whole should be called "GNU/Linux": it's a combination of the software that makes up the

GNU system and the Linux kernel.

That's not a bad point, and so generally when I am speaking of the whole system I tend to refer to it either as GNU/Linux or sometimes as Linux/GNU (because, you know, symmetry). But *this* process is called "Cross-Building Linux" rather than "Cross-Building GNU/Linux" simply because I think it sounds nicer.

The job of the kernel — this is a critically important point and I repeat it a lot — is to initialize and manage all of the hardware on the computer (including CPUs, memory, and I/O devices) and provide services to userspace processes. That's a big job, but it's also a limited one! Everything you *do* with your computer is the responsibility of userspace processes.

The way that Linux performs its job — more generally, the way that UNIX kernels work — is conceptually very simple: when it is executed on a computer, Linux does some hardware initialization, mounts the root filesystem, and then starts a single userspace process. That process has process ID (PID) 1, and is conventionally called `init`.

The `init` process is responsible for starting all other userspace programs and getting the machine into a usable state; after the kernel starts `init`, it gets out of the way and waits for that process, or for other userspace programs, to request its services.

The complexity in the Linux kernel emerges mostly from the huge variety of hardware that it supports and the many layers of functionality that can be built into it. If you unpack the Linux source code, you'll find that the whole thing adds up to about (as of the 6.6 kernel) 1477 megabytes in total. Of that, almost ten percent (144 megabytes) is specific to the twenty-six different CPU architectures that Linux supports, and about two-thirds (971 megabytes) is device drivers. That means that for any given system, a large majority of the code that makes up the Linux kernel won't *ever* be used.

In CBL, the kernel is kept as pristine as possible. In most cases, the only patches we apply to it are intended to add new default configuration settings — which is more convenient than setting dozens of configuration options manually.

Patch:

- linux-6.6.1-aws-ami-config-1.patch

6.10.2. linux (gnu-cross-toolchain phase)

The way that programs make requests of the kernel is by invoking kernel functions known as "system calls." The Linux kernel sources include header files that define all of the system calls it makes available to userspace programs.

Userspace programs don't *usually* invoke system calls themselves (although they *can*, and some do). Instead, they invoke library functions — particularly the ones defined in the C standard library, which on GNU/Linux systems is usually the GNU libc, glibc, but might be musl or uClibc or something else altogether. Those library functions invoke system calls as needed to accomplish their work.

In this step, we're not actually building the Linux kernel; all we're doing is installing the header

files that specify those system calls. These header files are then used by the C library and other userspace libraries and programs to invoke system calls as needed.

The Linux source tree also includes a number of other, private, header files — these define data structures and functions that should be used only within the kernel itself, and are not intended to be visible to userspace programs. The Makefile target we use here, `headers_install`, installs only the public header files, not these private internal headers.

The makefile target `mrproper` puts the source tree into a completely pristine state. The name is a reference to the Proctor & Gamble cleaning product that people in the United States know as "Mr. Clean"; in many parts of Europe, it is marketed under the brand name "Mr. Proper."

Configuration commands:

```
make mrproper
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

The `install_headers` target deletes the entire target directory before it does the installation. This is inconvenient, because we might have header files there that we want to retain (such as from the binutils build). To avoid any issues, we're going to install the headers to a temporary location and copy them to the real target location from there.

Notice that we're actually installing the headers into the scaffolding directory *underneath* the sysroot directory. That's common to the entire host-side portion of CBL: *Everything* that will be conveyed to the target system is under the scaffolding location. That way, when the final system is completed, we can get rid of `/scaffolding` and be fairly confident that everything remaining has been built entirely from source and (if it continues to work) has no dependencies on the scaffolding programs and libraries.

Installation commands:

```
make ARCH=arm64 INSTALL_HDR_PATH=_dest headers_install
install -dv /tmp/cblwork/sysroot/scaffolding/include
cp -rv _dest/include/* /tmp/cblwork/sysroot/scaffolding/include
rm -rf _dest
```

6.11. gcc

Name	GNU Compiler Collection

Version	13.2.0
Project URL	http://gcc.gnu.org/
SCM URL	git://gcc.gnu.org/git/gcc.git
Download URL	https://ftp.gnu.org/gnu/gcc/
Patches	• `gcc-13.2.0-branch-updates-20231103.patch` • `gcc-13.2.0-fix-relocation-headers-1.patch` • `gcc-13.2.0-fix-missing-rpath-1.patch` • `gcc-13.2.0-remove-crypt-interceptors-1.patch`
Dependencies	gmp (gnu-cross-toolchain phase), mpfr (gnu-cross-toolchain phase), mpc (gnu-cross-toolchain phase), isl (gnu-cross-toolchain phase), binutils (gnu-cross-toolchain phase)

6.11.1. Overview

GCC is the GNU Compiler Collection. This is the single most important package in CBL, even including the Linux kernel itself: without a compiler, you can't build any software. We use GCC to bootstrap everything, including itself.

Unfortunately, GCC is also probably the most complex package we need to build, and it has a very complex configuration and build process because it is tied so intricately to other packages. This is particularly true of glibc, the C standard library we use in CBL, which also has a complex build process of its own!

On the plus side, if you can get GCC built and working properly, you're past the biggest hurdle of building a complete GNU/Linux system entirely from source.

The GCC installation process is documented in the "Installation" manual, found in the source distribution in the `INSTALL` directory. If you have problems getting GCC built, that's an excellent resource for figuring out what's going wrong. If you want to get a better understanding of the GCC configuration and build process — how it actually *works* — read the "Source Tree Structure and Build System" section of the GCC Internals document, found in the source distribution in `gcc/doc/gccint.info`.

The most important compilers in the collection, for purposes of bootstrapping a system, are the C and C++ compilers; but GCC also includes compilers for D, Fortran, Go, Ada, Objective-C, and probably other languages as well. And it supports a huge number of machine architectures! Since CBL relies absolutely on having support for at least two types of computer for every build, that support for many architectures is probably the most important aspect of GCC for our purposes.

6.11.2. `gcc`: The driver program

The program you usually invoke to build C programs, `gcc`, is not actually a compiler. It's just a driver that knows how to invoke other programs, using rules called "spec strings" that tell it exactly what other programs it needs to invoke, and what command-line arguments it should provide, to

turn source code into an executable program. (We'll talk more specifically about spec strings in Adjusting the GCC specs.)

I find it really important to keep that in mind throughout toolchain construction, so I'll expand on that point: `gcc` just invokes other programs. Some of those programs — the C preprocessor `cpp`, `cc1` (the C compiler itself), and internal utility programs like `collect2` (which is sort of a first-stage linker, used to set up calls to constructors and other initialization routines as a program starts to run) — are part of the GCC package. Others, like the `as` and `ld` programs, are distributed separately (in the case of `as` and `ld`, that package is GNU binutils).

To get from source code to an executable C program, `gcc` actually performs a series of steps:

1. First, it invokes `cpp` to pre-process the source, include header files, resolve macros, and things like that;[1]
2. then it invokes `cc1` to transform the pre-processed source into assembly language code;
3. then it invokes `as` to transform the assembly code into object code;
4. and finally it invokes `ld` to combine all the separate bits of object code, along with code from libraries, into an executable program. (At least, conceptually, that's what it does. In reality, it invokes `collect2`, which does some other stuff and then executes the actual `ld` program).

You can find out exactly what commands `gcc` is running by giving it the `-v` (for "verbose") command line argument. That's a handy trick when things are going wrong and you're not sure why!

The term "compilation" — which is the job of the compiler — actually refers only to the second of those steps: source code is compiled into assembly code. It's not precisely correct to say that you're "compiling" source code into an executable program! That's a common conversational shorthand, but it masks the complete story about what's going on. It would be more correct to say that you're "pre-processing, compiling, assembling, and linking" source code to produce an executable program.

On the other hand, that's a lot of words, and the complete story is seldom really something you need to keep in mind. This is probably why the shorthand term is so popular.

6.11.3. How C and C++ Programming Languages Work in GNU/Linux

NOTE

In *Cross-Building Linux*, when we first discuss the package that provides a programming language, we include a section like this one that describes some high-level details about how that language works in GNU/Linux systems generally, and in Little Blue Linux in particular. That seems a little strange in the case of C and C++, since those are so intrinsic to the system: the packages that make up the whole foundation of the system are implemented in C or C++, and a great deal of the base LB Linux system is present entirely to support them. But I think it's a good idea to establish the convention now and maintain it consistently throughout.

Here, I'm focusing mainly on the C programming language. The two languages are very closely related, to the point that the earliest C++ compiler didn't produce binary object code at all; it compiled C++ programs into C source code, which was then compiled into binary programs by a C compiler. A lot of the details about how C works apply exactly the same in C++; the differences are

mostly things like the naming convention used for source files (C language source code files have the extension `.c`. There is no single convention that is used consistently for C++ source files! According to the `gcc` manual page, extensions for C++ source files include `.cc`, `.cp`, `.cpp`, `.cxx`, `.CPP`, `.C`, and `.c++`) and the program used to invoke the compiler (`g++` rather than `gcc`).

I'm not discussing other languages, like Objective-C or Fortran, at all, because I don't know anything about how they work!

C source code files generally have the extension `.c`. Along with these, you'll often find *header files*, typically with the extension `.h`; these contain function declarations, macro definitions, constant definitions, and other things like that. These files can contain *preprocessor directives*, which are evaluated and executed by the C preprocessor before the code is compiled. Commonly, C source files contain `#include` directives that cause the preprocessor to copy the content of header files into the C file as part of this evaluation.

In addition to the header files that are included as part of the source code for C packages, UNIX systems have *system header files* that can be included by any program that needs them. These files can typically be found in the `/usr/include` directory and its subdirectories.

In addition to executable programs, packages can install *libraries*. These are basically collections of related functions that can then be used in other programs. These libraries are commonly installed in the `/lib` and `/usr/lib` directories (and their subdirectories). Sometimes, libraries are installed into variants of those directories, which can be set up on systems that support more than one kind of CPU instructions. For example, the AMD64 CPUs that are used in modern Intel and AMD systems support both 64-bit and 32-bit instruction sets, and GCC can produce both 64-bit and 32-bit object code. If you have one of these computer systems, you may want to have both 64-bit and 32-bit programs. In that cases, 64-bit programs will need to be linked against 64-bit verisons of libraries, and 32-bit programs will have to be linked against 32-bit versions, but both versions of the libraries will have the same filename. To allow these "multilib" systems, libraries will be installed in directories like `/usr/lib32` and `/usr/lib64` rather than simply `/usr/lib`.

In most cases, when a package installs a library in any of those directories, it also installs header files (into `/usr/include`) that specify how to call the functions in that library.

There are two different kinds of library files. Both are used when a program (or sometimes another library) are linked. *Static libraries* have the extension `.a`; when a program is linked against a static library, the binary code for methods invoked by that program is actually copied from the library into the program. *Shared libraries*, with the file extension `.so`, are the other kind. When a program is linked against a shared library, a reference to the library is written into the program, but the code for library functions is not. The benefit of this is that, no matter how many programs you have that use a common function like `memcpy` or `printf`, only one copy of the function code needs to be present on the system, in the shared library that defines them. Programs with such library references are called "dynamically linked," and when they are run, the program interpreter is responsible for tracking down the shared libraries it uses and finding the code for functions within them. We'll talk more about how it does that later on!

One library present in the Little Blue Linux system is simply called "the C standard library." The functions in that library are defined as a part of the C language, and almost every C program uses some of those functions. (For example, the "Hello, World" program that is traditionally the first C

program written by those learning the language uses the `printf` function defined in the standard library.)

This package does not include an implementation of the standard library.[2] In the GNU system (and in Little Blue Linux), the standard library is provided by the glibc package — which also provides the program interpreter that resolves shared library references in dynamically-linked programs. Although glibc includes both static and shared versions of libraries, it does not really work well with statically-linked programs. So the vast majority of executable programs found on GNU/Linux systems are dynamically linked.

After being compiled and linked, executable C programs are typically installed into directories like `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin`. The same directories also contain all of the other programs that are part of the system, regardless of whether their source code is written in C.

Many of these topics are discussed in more detail elsewhere in CBL. The others are discussed in more detail *somewhere,* but you might have to do some searching! In any case, that's a basic summary of how C programs work on LBL — and other GNU/Linux — systems.

Dependencies

gmp, mpfr, mpc, isl.

A few external dependencies were added to gcc in release 4.3: the GMP, MPFR, and MPC libraries are required for all compiler builds, and a few optimizations — the graphite loop optimizations — are only available if the Integer Set Library (ISL) is available. The graphite loop optimizations are not critically important, but there's no reason not to include them if it's not difficult to do.

Sources for the required — and optional — libraries can be included in the GCC sources in directories named gmp, isl, mpc, and mpfr; if they are, then they'll be built automatically along with gcc. There's actually a fairly large number of packages that the build machinery for GCC detects and incorporates into builds automatically. This is convenient, but restrictive: in-tree builds are only reliable when specific versions of the dependency libraries are present, and for CBL I prefer to use the latest stable release of everything.

There are also some cross-compilation scenarios in which the in-tree library builds actually do the wrong thing and produce a compiler that will not work right; that's another reason that we avoid them here.

Patch:

- gcc-13.2.0-fix-relocation-headers-1.patch

There's a problem introduced in GCC 6.1 involving hard-coded paths to C++ header files. This prevents them from being found by the scaffolding compiler (which lives at a different filesystem location when the target system is booted). We can apply a patch to work around that.

Patch:

- gcc-13.2.0-fix-missing-rpath-1.patch

When specifying locations for the dependency libraries (GMP, MPFR, etc), the specified library location should be used both at build time (with an `-L` linker directive) and at run time (with an

`-rpath` link directive). This is not always done by the normal GCC build process, but we can patch it easily to add that behavior.

CAUTION	The next few paragraphs discuss issues related to the way the dynamic linker works. If you're not familiar with its operation, it might be a good idea to review the section A Word About The Dynamic Linker — or just skip ahead a bit, if you're not interested in the details of a linking problem and the reason we apply this patch to work around it.

Even though the math libraries needed by GCC (GMP, ISL, MPC, and MPFR) have just been built here and GCC is configured to find them in their correct locations, some of the programs that make up GCC are built without an `RPATH` or `RUNPATH`, so the dynamic loader will look for those libraries at runtime in the normal host system library directories. This was filed as GCC bug 84153, but as of the last time I looked at it, the GCC maintainers don't consider it a problem.

To be fair, this really is *not* usually much of a problem! The issue appears when the version of the library used in CBL is a major version later than the version installed on the host system. In that case, the GCC we're building in this section won't work because it won't be able to find the version of the libraries it depends on.

This is the sort of situation that `LD_LIBRARY_PATH` is intended to resolve, but there's a gotcha: in some of the host-scaffolding builds, the *native* toolchain is used to build some programs that are run as part of the build process itself, and the `LD_LIBRARY_PATH` set in the environment for the cross-build is unset when building those native programs.

That means that to get the this native gcc to be reliable, we need to set a `RUNPATH` or `RPATH` to tell the dynamic loader where to look for shared libraries. The way you normally do this is to set an `LDFLAGS` environment variable with a value like `-Wl,-rpath,/home/random/cbltools/lib` when running the `configure` script, so that `ld` would be told to build programs with that `RPATH`, but it turns out the GCC build doesn't use the `LDFLAGS` linker arguments when constructing some of the programs that need the dependency libraries, like `cc1`, `cc1plus`, and `lto1`.

We can work around that by patching the `configure` script so that any time it's given arguments like `--with-gmp` or `--with-gmp-lib`, the flags it collects will include a `-Wl,-rpath` option along with the `-L` option. That's what this patch does.

Patch:

- gcc-13.2.0-remove-crypt-interceptors-1.patch

GCC 13.2.0 sometimes does not build properly when using glibc 2.38 because libsanitizer expects to do things related to the `libcrypt` library provided by glibc, but starting in glibc 2.38, that library is no longer built by default. I found a patch in GCC bug 111057 that fixes this.

6.11.4. gcc (gnu-cross-toolchain-minimal phase)

Environment	• `CPPFLAGS_FOR_TARGET`: `--sysroot=/tmp/cblwork/sysroot` • `LDFLAGS_FOR_TARGET`: `--sysroot=/tmp/cblwork/sysroot`

Build Directory	`../build-gcc-2`

Dependencies

gmp (gnu-cross-toolchain phase), mpfr (gnu-cross-toolchain phase), mpc (gnu-cross-toolchain phase), isl (gnu-cross-toolchain phase), binutils (gnu-cross-toolchain phase).

Bootstrapping GCC as part of a cross-toolchain is tricky. You can build the compiler *per se* without a working standard library (libc), but that compiler won't be able to produce executable programs: GCC can only create programs if it has access to a set of C runtime object files that it expects to be provided by the C library — and, realistically, it also needs a C library to compile programs because essentially all C programs invoke standard library functions.

Obviously, we can't compile the C library without a compiler! So there's a chicken-and-egg bootstrapping problem.

To work around that cycle of dependencies, we're going to start by building just the parts of the compiler we really need at this point: a plain cross-compiler, and a minimal version of the `libgcc` support library that the compiler needs in order to function. (GCC needs `libgcc` because sometimes, while processing C code, the compiler generates references to functions defined in `libgcc` rather than generating assembler code.)

GCC can most easily be built as part of a cross-toolchain by using the "sysroot" framework — and, in fact, the GCC developers don't support any other method for creating cross-toolchains. To perform a sysroot build, you specify the configure options `--with-sysroot` and `--with-build-sysroot`; and when building GCC, set the environment variables LDFLAGS_FOR_TARGET and CPPFLAGS_FOR_TARGET to specify `--sysroot` as well.

Environment variable: CPPFLAGS_FOR_TARGET

```
--sysroot=/tmp/cblwork/sysroot
```

Environment variable: LDFLAGS_FOR_TARGET

```
--sysroot=/tmp/cblwork/sysroot
```

...At least, that's what Carlos O'Donell said in a comment on GCC bug 35532. The documentation on sysroot builds is not particularly easy to find — or at least, it wasn't when this was written. (If you know where sysroot builds are documented, please tell me!)

The sysroot concept is pretty clever, in fact. The basic idea is, when you build a cross-toolchain, you give it a local directory path — a directory that exists on the *build* system — and tell the cross-toolchain that that local directory will eventually become the root filesystem directory on the *target* system. The cross-toolchain then knows to look for system header files and shared library files and so on in the sysroot location.

CBL uses the standard sysroot approach, in a slightly-nonstandard way: everything is installed not into the sysroot directory *per se*, but into a *subdirectory* of the sysroot called `/scaffolding`. That way, when we finally get booted into the target system, the root filesystem will be empty, except for the `/scaffolding` directory: everything we create after that, and install outside of `/scaffolding`, will become part of the final system.

Depending on the target, GCC has a variety of options that control how it operates. Generally, these can all be specified with command-line arguments beginning with `-m`. For many of these options, default values can also be specified when GCC is being configured.

For example, for many targets, GCC can build programs with a variety of application binary interfaces (ABIs) — this is the machine-code-level interface between programs, libraries, and the operating system; it defines things like the way that registers are used when invoking functions.

An example of a CPU that supports multiple ABIs is the 64-bit x86 architecture, which is called `x86_64` or `amd64`. As mentioned earlier, these processors can run programs in 64 bit mode (with 64-bit pointers and all AMD64 processor features enabled), 32 bit mode (32-bit pointers and only i686 processor features enabled — in this mode, fewer CPU registers are availble to programs, for example), or a hybrid "x32" mode (32-bit pointers but all AMD64 processor features enabled). You can specify the ABI to which GCC will compile by using an `-m32`, `-m64`, or `-mx32` command-line argument.

You can also override the default ABI when configuring GCC by specifying a `--with-abi` configure directive; or, for some target architectures, other options, like `--with-multilib-list` or `--enable-targets` or probably a combination of those things. Confusingly, even though the way you set an x86 GCC to generate code for the 64-bit ABI is to use the configuration directive `--with-abi`. Although there is a runtime command line option `-mabi`, it doesn't override that selection; it only selects between the `sysv` calling convention used by UNIX-ish systems and the `ms` convention used by Microsoft Windows.

This confusion is characteristic of the GCC configuration and usage options: a lot of the options that are available, and their meaning, depends on what architecture is targeted by the compiler, and the same options or terms can mean very different things for different targets.

There are a few ways to figure out what options are available for a specific target architecture: the installation manual lists some of them, mostly in the "configuration" section. The GCC manual also has information in section 3.18 ("Machine-Dependent Options" — if you've built a set of trusted host tools specifically for the CBL build, the correct info file can be found at `/home/random/cbltools/share/info/gcc.info`). You can also look at the configuration script, `gcc/config.gcc`, to see what options are accepted for each type of target. And, finally, after building GCC for the target architecture, you can run `gcc --help=target` to see what options are available and what values they can have; and you can also find the actual compiler, `cc1` (for C) or `cc1plus` (for C++) under the `libexec/gcc/aarch64-cbl-linux-gnu/$VERSION` directory, and run it with the `--help` command-line argument to see what options are enabled and disabled.

In addition to the selection of ABI, GCC can be instructed to optimize for specific CPUs or CPU families for many target types. The command-line arguments that control this behavior, like the ABI arguments, are target-specific — look for `-mtune`, `-march`, `-mcpu`, and things like that.

In CBL, any set of these configure directives can be specified for the cross-toolchain in the `TARGET_GCC_CONFIG` parameter. It's generally a good idea to specify default values for all of the options supported by the target platform. (If you don't want to set any options at all, you can set the parameter to an empty string.)

GCC is generally built with a large number of libraries included. Some of those fail in some

circumstances — for example, x86 CPUs can't build the libquadmath library when the C library being used is uClibc (or, at least, that was the case some time ago, the last time I tried); and the libsanitizer library fails to build when compiling for 64-bit Sparc machines. The easiest way to work around those problems at this stage is simply to disable those libraries, and, considering that the cross-toolchain we're building here is just going to be used to build the ephemeral scaffolding programs, that seems like a reasonable approach here. So if any libraries cause the build to fail, try adding an appropriate `--disable` directive to `TARGET_GCC_CONFIG`.

Build Directory

```
../build-gcc-2
```

You might notice that, in this build, we're using the same host-system builds of the various arithmetic dependency libraries as we used for the host-prerequisite GCC (or the same ones that are used for the native system GCC, if you've skipped the host-prerequisites). It's totally unnecessary to build them again for this GCC — the cross-compiler we're building here will only run on the host system, so the target architecture is irrelevant to it.

To build the minimal libgcc, we specify the configuration options `--without-headers` and `--with-newlib`. This is a bit sloppy — the first of those options is the only one that should be necessary — but the last time I tried building a minimal compiler without the newlib directive, it didn't work.

Once this minimal compiler is finished, we'll use it to build the C library; and then we can use the files provided by the C library to go back and build a full, functional GCC compiler.

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/tmp/cblwork/cross-tools \
  --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=aarch64
-cbl-linux-gnu \
  --with-sysroot=/tmp/cblwork/sysroot --with-build-sysroot=/tmp/cblwork/sysroot \
  --disable-decimal-float --disable-libgomp --disable-libmudflap \
  --disable-libssp --disable-multilib --disable-nls --disable-shared \
  --disable-threads --enable-languages=c,c++ --with-newlib \
  --without-headers \
  --with-gmp=/home/random/cbltools --with-mpfr=/home/random/cbltools \
  --with-mpc=/home/random/cbltools --with-isl=/home/random/cbltools \
  --enable-fix-cortex-a53-835769 --enable-fix-cortex-a53-843419 --with-cpu=cortex
-a72.cortex-a53 --enable-standard-branch-protection
```

The only targets we build at this point are `all-gcc`, which produces the plain compiler, and `all-target-libgcc`, which produces the minimal libgcc needed by that compiler.

Compilation commands:

```
make all-gcc all-target-libgcc
```

Test commands:

```
(none)
```

Installation commands:

```
make install-gcc install-target-libgcc
```

6.12. glibc

Name	GNU C standard library
Version	2.38
Project URL	https://www.gnu.org/software/libc/
SCM URL	git://sourceware.org/git/glibc.git
Download URL	https://ftp.gnu.org/gnu/libc/
Patches	• `glibc-2.38-branch-updates-20231103.patch`
Dependencies	linux, gcc (gnu-cross-toolchain-minimal phase)

6.12.1. Overview

glibc is the C standard library produced as part of the GNU project. It contains the implementation for all of the functions that are assumed to be available in C programs — like `printf` and so on. It also provides a dynamic loader, which almost all programs use to find shared libraries at runtime, and a few miscellaneous utility programs.

Since all C and C++ userspace programs (except the Linux kernel itself) link against the C library, glibc is by far the most deeply-embedded component of a GNU/Linux installation. Upgrading the Linux kernel is a relatively trivial operation compared to upgrading the C library installed on a computer.

There are alternative C libraries that can be used instead of glibc. However, glibc is the C standard library used by the vast majority of GNU/Linux systems; using an alternative library like musl or uClibc-ng as the primary C library may cause problems somewhere down the line.

6.12.2. glibc (gnu-cross-toolchain phase)

Build Directory	`../build-glibc-1`

Dependencies

linux, gcc (gnu-cross-toolchain-minimal phase).

This builds a sysroot libc (for the target architecture), configured to install into the `/scaffolding`

subdirectory of the sysroot.

That might be a little bit opaque, so let's break it down a little bit. As mentioned earlier, the sysroot framework is all about setting up a path on the host system that will eventually become the root filesystem on the target system. And, also as mentioned earlier, the goal in the first stage of the CBL build process is to set up a kernel and minimal userspace for the target system, with the entirety of that userspace contained in a `/scaffolding` subdirectory rather than using the conventional filesystem paths (like `/bin` and `/lib` and the `/usr` directory structure and all the rest of that stuff you're used to seeing).

What we're building here is the libc that's going to be used to build all the scaffolding programs and libraries — the stuff that we're cross-compiling — so we want it to be contained in the scaffolding directory, and found by the scaffolding programs at runtime in that location. Hence, it's a sysroot libc, with an extra prefix to move it from its usual normal `/lib` and `/usr/lib` directories to the `/scaffolding/lib` directory.

That means we configure it with a prefix of `/scaffolding`, but then when we install it we tell it that the root of the installation location is the sysroot directory `/tmp/cblwork/sysroot`.

Simple, right?

If you ever want to set up a completely standard sysroot toolchain, by the way, it works *pretty much* the same way as this but you specify `--prefix=/usr` and `--with-headers=/tmp/cblwork/sysroot/usr/include`. There is some magic in the glibc configuration or build machinery related to the `--prefix` directive: if you specify a prefix of `/usr`, the bits of glibc that are conventionally installed in `/lib` will be put there, rather than in `/usr/lib`.

Build Directory

```
../build-glibc-1
```

Since this glibc is built for the target machine architecture, a number of tests run by the configure script won't work right. The way we work around that, for glibc as with everything else that uses the GNU build system, is by setting the correct values in a `config.cache` ahead of time.

An option that we're *not* using here is `--enable-kernel`, which can limit the amount of compatibility code built into glibc to support old Linux kernel versions. Since the glibc being built here is temporary and will be discarded in its entirety, saving a little bit of space here is kind of pointless. That option also makes the build more fragile, since the kernel version that will be checked by the code is the host system kernel; we don't want to make any more presumptions about the host system than we must.

Configuration commands:

```
echo "libc_cv_forced_unwind=yes" > config.cache
echo "libc_cv_c_cleanup=yes" >> config.cache
echo "libc_cv_gnu89_inline=yes" >> config.cache
echo "libc_cv_ctors_header=yes" >> config.cache
echo "libc_cv_ssp=no" >> config.cache
echo "libc_cv_ssp_strong=no" >> config.cache
BUILD_CC="gcc" CC="aarch64-cbl-linux-gnu-gcc" AR="aarch64-cbl-linux-gnu-ar" \
```

```
  RANLIB="aarch64-cbl-linux-gnu-ranlib" CFLAGS="-g -O2" \
  ${LB_SOURCE_DIR}/configure --prefix=/scaffolding \
  --host=aarch64-cbl-linux-gnu --build=x86_64-unknown-linux-gnu \
  --disable-profile --enable-add-ons --with-tls --with-__thread \
  --with-binutils=/tmp/cblwork/cross-tools/bin \
  --with-headers=/tmp/cblwork/sysroot/scaffolding/include \
  --cache-file=config.cache
```

The glibc build process uses the `makeinfo` program to create the documentation, and the texinfo source file specifies a document encoding of UTF-8. When using some versions of perl, this leads to problems — it's unclear why; perhaps this is because `makeinfo` only works right with UTF-8 documents when being run with UTF-8 localization settings, or maybe something else is going wrong.

Regardless of exactly what triggers the issue, there's an easy way to work around it: just remove the `@documentencoding` directive from the libc manual source file.

Configuration commands:

```
sed -i -e '/^@documentencoding UTF-8$/d' \
  ${LB_SOURCE_DIR}/manual/libc.texinfo
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install_root=/tmp/cblwork/sysroot install
```

6.13. gcc (gnu-cross-toolchain phase)

For an overview of gcc, see gcc.

Environment	• `CPPFLAGS_FOR_TARGET`: `--sysroot=/tmp/cblwork/sysroot` • `LDFLAGS_FOR_TARGET`: `--sysroot=/tmp/cblwork/sysroot`
Build Directory	`../build-gcc-3`

Dependencies

glibc.

Now that we have a C library installed, we can finally do a full GCC build. So now we'll enable multi-threaded code and some of the runtime libraries we turned off previously.

Build Directory

`../build-gcc-3`

Environment variable: CPPFLAGS_FOR_TARGET

`--sysroot=/tmp/cblwork/sysroot`

Environment variable: LDFLAGS_FOR_TARGET

`--sysroot=/tmp/cblwork/sysroot`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/tmp/cblwork/cross-tools \
  --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=aarch64
-cbl-linux-gnu \
  --with-sysroot=/tmp/cblwork/sysroot --with-build-sysroot=/tmp/cblwork/sysroot \
  --disable-multilib --disable-nls --enable-languages=c,c++ \
  --enable-__cxa_atexit --enable-shared --enable-c99 \
  --enable-long-long --enable-threads=posix \
  --with-native-system-header-dir=/scaffolding/include \
  --with-gmp=/home/random/cbltools --with-mpfr=/home/random/cbltools \
  --with-mpc=/home/random/cbltools --with-isl=/home/random/cbltools \
  --enable-fix-cortex-a53-835769 --enable-fix-cortex-a53-843419 --with-cpu=cortex
-a72.cortex-a53 --enable-standard-branch-protection
```

Fairly early in the build process, a cross-compiler version of `gcc` is built and installed as `xgcc`, for use in later build steps. Then later on, when building `libgcc.so` and various other libraries that are part of GCC, `xgcc` runs the cross-binutils `ld`. Unforunately, `xgcc` doesn't know to tell `ld` to look in the `/scaffolding` directory to find the startup files (`crti.o` and so on). `xgcc` also insists on looking for libraries and startup files in the variant `lib` directories used by the multilib scheme, even though we configure GCC with `--disable-multilib`; it doesn't appear that there's any way to coerce the GCC build system not to use those multilib directories.

A workaround that sometimes helps is to add the correct directory using the `LDFLAGS_FOR_TARGET` and `CFLAGS_FOR_TARGET` options. That's no good in this case, though: many of the libraries in gcc use the `libtool` script to do all their compilation and linking, and libtool ignores the `LDFLAGS` and `CFLAGS` we set.

So to get this build completed, we use a kludge: we create symlinks from `/tmp/cblwork/sysroot/scaffolding/lib` to all the locations where `xgcc` might expect to find the libraries.

If the build crashes and gets restarted, the `ln` commands will fail. That doesn't matter, so we temporarily tell the shell to proceed rather than terminating if an error occurs.

Compilation commands:

```
set +e
```

```
ln -s /tmp/cblwork/sysroot/scaffolding/lib /tmp/cblwork/sysroot/lib
ln -s /tmp/cblwork/sysroot/scaffolding/lib /tmp/cblwork/sysroot/lib32
ln -s /tmp/cblwork/sysroot/scaffolding/lib /tmp/cblwork/sysroot/lib64
ln -s /tmp/cblwork/sysroot/scaffolding/lib /tmp/cblwork/sysroot/libx32
set -e
make AS_FOR_TARGET="aarch64-cbl-linux-gnu-as" LD_FOR_TARGET="aarch64-cbl-linux-gnu-ld"
```

Test commands:

```
(none)
```

Installation commands:

```
make install
rm -f /tmp/cblwork/sysroot/lib*
```

6.14. Adjusting the GCC specs

6.14.1. Overview

As mentioned earlier, in `gcc`: The driver program, the `gcc` program we think of as a compiler really just runs other programs, and it uses a bunch of directives called "spec strings" to determine what programs to run and what options to give them. The format of specs strings is documented in the GCC documentation in section 3.15, "Specifying Subprocesses and the Switches to Pass to Them." Spec strings don't have the most readable structure — I find it helpful to think of them as being written in a domain-specific language, because to me they look as much like line noise as complicated regular expressions do — but sometimes there's no better way to figure out what is going on than to read the specs, and there is often no better way to adjust the behavior of `gcc` than to modify the specs it is using.

We have to do this — modify the specs — a few times throughout the CBL process, primarily because we need to control how `gcc` runs `ld` to link programs.

You can see the spec strings that gcc will use by running `gcc` with the `-dumpspecs` option. The default specs are built in to `gcc`, but you can provide your own specs to override the default behavior by using the `-specs=` command-line argument or by creating a specs file and putting it in a specific location in the filesystem.

We're going to do the latter. The basic process is to first dump the specs file to the location where `gcc` will look for it:

```
gcc -dumpspecs > $(dirname $(gcc -print-libgcc-file-name))/specs
```

Then we can modify it however we need to, and `gcc` will use the modified version.

6.14.2. Adjusting the GCC specs (gnu-cross-toolchain phase)

Dependencies

gcc (gnu-cross-toolchain phase).

CAUTION

If you look at the structure of the toolchain directory structure, you'll see that there are a couple of different ways you can refer to the programs in it. First, in the `bin` directory, there are a bunch of programs prefixed with the target-triple (e.g. `aarch64-cbl-linux-gnu-gcc`); second, in `aarch64-cbl-linux-gnu/bin`, the same set of programs exist without a prefix. You can use either one, but there's a gotcha: if you put `/tmp/cblwork/cross-tools/aarch64-cbl-linux-gnu/bin` in your PATH, driver programs like `gcc` and `g++` sometimes won't be able to find some of the other programs they rely on, like `cc1` and `cc1plus`. Those programs are in `libexec/gcc/aarch64-cbl-linux-gnu/VERSION`, and for some reason they can be found when the driver executable is in `bin` but sometimes not when it's in `aarch64-cbl-linux-gnu/bin`.

That means that unless you put the path to the directory containing `cc1` in your `PATH` as well, you might wind up getting mysterious error messages like `gcc: error trying to exec 'cc1': execvp: No such file or directory` when you try to compile programs. If that happens to you, just add the `cc1` directory to your `PATH`, or use the full `aarch64-cbl-linux-gnu-gcc` program name to compile programs.

Not to belabor the point, but remember that the whole purpose of this cross-toolchain is to let us build the scaffolding that will then allow us to build the final target system entirely from source code.

While we're using the scaffolding tools to build the final system, we want to do a native build of everything, including glibc, so we want the scaffolding to be independent of any filesystem locations that will still be present in the final system. In particular, this includes `/lib` and `/usr/lib`. That way, once we're done doing the target system build, we can delete the scaffolding directory altogether and be confident that there are no lingering host-system artifacts or ephemera on it.

When the standard GNU toolchain builds an executable, it almost always links it against the dynamic link library (sometimes called the "dynamic loader" or "program interpreter"; this is something like `ld-linux.so.2` or `ld.so.1`, and is conventionally found in a top-level directory called `/lib` or a multilib variant like `/lib32` or `/lib64`).[3] That's normally fine, but we want the scaffolding to be entirely independent of `/lib`. So we need to adjust our cross-toolchain so that the programs it builds look in the `/lib` directory *under the scaffolding location* for their libraries, including the dynamic link library. This is done by modifying the GCC specs file.

Commands:

```
aarch64-cbl-linux-gnu-gcc -dumpspecs | \
  sed -e 's@/lib/ld@/scaffolding/lib/ld@g' \
  -e 's@/lib32/ld@/scaffolding/lib/ld@g' \
  -e 's@/libx32/ld@/scaffolding/lib/ld@g' \
  -e 's@/lib64/ld@/scaffolding/lib/ld@g' > \
```

```
  $(dirname $(aarch64-cbl-linux-gnu-gcc --print-libgcc-file-name))/specs
```

There's another problem we need to work around, as well: `gcc` doesn't provide any good way to tell it where to find some object files that need to be linked into every program: `crti.o`, `crti1.o`, and `crtn.o`. These are provided as a part of glibc, so like the rest of glibc they were installed into `/tmp/cblwork/sysroot/scaffolding`, specifically into its `/lib` subdirectory. But, of course, that's not a location where gcc normally expects to find them. So at this point if you were to try compiling a "Hello World" program with your cross-toolchain, it would complain that the cross-ld can't find `crt1.o` or `crti.o`.

You can tell exactly what is going wrong by repeating the compile with `-v`, to get gcc to print out all the commands it's running: `cc1` to compile the code, then `as` to assemble it, then `collect2` to link it. And apparently `collect2` is running `ld`, which is producing the error message. (That's weird, but you get used to weird stuff when you're trying to figure out toolchain problems. There's also no explicit execution of `cpp`; that's because the C pre-processor is actually implemented in the `libcpp` library and is invoked as function calls to that library by the `cc1` compiler program.)

Once you have the actual command line that's failing, you can try adjusting it to see if there's an easy way to get it to work. For example, the command line just names the startup files without any path components at all. After fussing around for a while looking for pleasant alternatives, the best solution I found was to specify the filenames with absolute paths. So that's what we're going to do in the specs file: replace every occurrence of the bare filename with absolute paths, for all the object files that appear in the `scaffolding/lib` directory.

Commands:

```
for FILE in crt1 crti crtn gcrt1 Mcrt1 Scrt1; do \
  sed -i -e "s@\\b$FILE.o\\b@/tmp/cblwork/sysroot/scaffolding/lib/$FILE.o@g" \
  $(dirname $(aarch64-cbl-linux-gnu-gcc --print-libgcc-file-name))/specs; \
  done
```

We can now verify that the cross-toolchain is able to build programs successfully, and is set up to link against the sysroot glibc in the `/scaffolding` directory, by compiling any simple program (like "Hello World") and then running `readelf` on it — there will be a line in the program headers section that says "Requesting program interpreter:" and should contain the path to the dynamic link library in the scaffolding location.

6.15. Verify that a toolchain works properly

6.15.1. Overview

After building a significant toolchain component, it's a good idea to make sure that it works as intended. This is a simple smoke-test: it just compiles a "Hello, World" program and then inspects it to make sure it was built as expected and runs properly.

6.15.2. Verify that a toolchain works properly (gnu-cross-toolchain phase)

Environment variable: LD_LIBRARY_PATH

`/tmp/cblwork/cross-tools/lib:$LD_LIBRARY_PATH`

Environment variable: PATH

`/tmp/cblwork/cross-tools/bin:$PATH`

Dependencies

gcc.

This verifies that the cross-toolchain and emulator work properly: look at the machine type and dynamic linker location for a compiled program, and then make sure that it runs in the userspace emulator. This proves that the cross-toolchain and QEMU were properly built with a compatible target architecture and so on.

File /tmp/cblwork/build/hello.c:

```
#include <stdio.h>
int main(void)
{
    printf("Hello, QEMU Emulated aarch64-cbl-linux-gnu World!\n");
    return 0;
}
```

This is compiled with:

Commands:

```
aarch64-cbl-linux-gnu-gcc /tmp/cblwork/build/hello.c -o /tmp/cblwork/build/hello
```

To verify that it's linked properly, use readelf.

Commands:

```
aarch64-cbl-linux-gnu-readelf -a /tmp/cblwork/build/hello | tee \
  /tmp/cblwork/build/program_info
grep 'Machine:' /tmp/cblwork/build/program_info | grep \
  'AArch64'
grep 'interpreter: /scaffolding/lib' /tmp/cblwork/build/program_info
```

The `Machine` line should indicate the target architecture, rather than the host architecture, and the program interpreter requested by the program should be under the `/scaffolding/lib` directory (which is where the dynamic loader will be found once we're booted into the target system). If either of those is not the case, the `grep` commands will fail, causing the CBL build process to abort.

One last thing we can usefully do at this point is verify that the user-mode QEMU emulator can actually run programs for the target architecture.

This is a bit tricky when running the dynamically-linked program we just built, because it is expecting to find the program interpreter at `/scaffolding/lib` but in fact it's actually at that location *under the sysroot directory*. Luckily, the user-mode QEMU emulator can be told where to find library files, using the `-L` command line argument or the `QEMU_LD_PREFIX` environment variable.

Commands:

```
qemu-aarch64 -L /tmp/cblwork/sysroot /tmp/cblwork/build/hello | \
  grep 'Hello, QEMU Emulated aarch64-cbl-linux-gnu World'
```

This will produce a friendly greeting, or — if something goes wrong — will, again, cause the build process to abort.

(If you'd like, you can try running the `hello` program outside of QEMU as well; that should produce an error message like "cannot execute binary file." And, again, if it doesn't, that means something is horribly wrong!)

6.15.3. Complete text of files

6.15.3.1. /tmp/cblwork/build/hello.c

```
#include <stdio.h>
int main(void)
{
    printf("Hello, QEMU Emulated aarch64-cbl-linux-gnu World!\n");
    return 0;
}
```

Building a GNU/Linux system from the ground up is like constructing a building. At least, that's the analogy I had in mind when I was designing the CBL process and writing this book.

When constructing a building, it's sometimes useful to start by assembling a scaffolding around the build site. Then you can climb onto the scaffolding and use it as a support and framework while you construct the building you actually want. Once the final building is complete, you can tear down and discard the scaffolding — it's not important in and of itself, only as a means to an end.

That's what we do in the CBL process: we use the cross-toolchain to construct a set of programs and libraries that we can boot into, as an ephemeral "scaffolding" framework from which we can build the actual target system. We build that scaffolding in such a way that it sits alongside the final system as we build it; the scaffolding components won't conflict with anything that will eventually form part of the final Little Blue Linux system, because everything related to those components will be self-contained within a top-level `/scaffolding` directory. After the build is complete, we'll delete `/scaffolding`.

[1] This is not *technically* true, because the C pre-processor is implemented in a library called `cpplib`. The `cc1` compiler is linked against `cpplib` and invokes library functions directly to perform the pre-processing step, rather than executing a separate program. But from a *logical* perspective, this is the first step.

[2] Oddly, the GCC package *does* include an implementation of the C++ standard library!

[3] The exception to this is when the program is linked statically, e.g. with the `-static` linker directive. When using the GNU libc,

though, this can be problematic — you should not try to build statically-linked programs unless you are sure it is the right thing to do.

Chapter 7. Ensuring isolation from the host system

It's possible for the build process of some of the scaffolding programs to find things on the host system and try to compile or link against them. This doesn't work, of course, because the scaffolding programs are for the target machine architecture, and that's incompatible with the host system architecture. Unfortunately, it still causes the build to fail. We can prevent that by setting up some programs that will isolate us from the host system.

This really should not be needed! Everything we're building in the scaffolding is being cross-compiled, and packages should never try to compile or link against any host-system libraries when they're being cross-compiled. But I ran into an issue with this at least once, and setting up a small guard against this kind of build-system bug is not hard to do.

7.1. pkgconf

Name	Improved pkg-config
Version	2.0.3
Project URL	https://github.com/pkgconf/pkgconf
SCM URL	*(unknown)*
Download URL	https://github.com/pkgconf/pkgconf/releases
Patches	• `pkgconf-2.0.3-run-autogen-1.patch`

7.1.1. Overview

pkg-config is a program that makes it easy to find installed libraries and header files and things. Many programs use pkg-config in their build processes to find out if the libraries they depend on are present on the build system, and to find out what linker and include directives should be used to compile and link against them.

The original pkg-config program can be found at `pkg-config.freedesktop.org`. At some point in its development, its developers decided to use functions from the glib library; unfortunately, glib uses pkg-config to find its own dependencies (really, just zlib — glib doesn't have a lot of dependencies), which introduced a cyclic dependency. That doesn't cause any really intractable problems — a couple of environment variables can be defined when building glib so that it doesn't have to use pkg-config to find zlib — but cyclic dependencies are always kind of horrifying: not to sound like a broken record, but the idea with CBL is to start with a *minimal* set of binaries and a pile of source code and turn that into a whole system. (Even worse than the glib dependency, pkg-config requires *itself*, so that it can find and link against the glib library, unless additional environment variables are provided.)

The situation has since been resolved, but it was resolved by bundling glib along with pkg-config. This isn't a particularly elegant solution: the pkg-config distribution is about 11.5 megabytes of

code, unpacked, and about 9.5 megabytes of that is glib.

A different approach was taken in a fork of pkg-config, "pkg-config-lite," which includes just the snippet of glib that is needed by pkg-config: that's better, but still not ideal.

This brings us to pkgconf, a completely separate implementation of the pkg-config program. It has no external dependencies, doesn't bundle any third-party code within its own distribution, and also has a design that I find preferable to the original pkg-config (it internally builds a directed acyclic graph of dependencies, rather than building an in-memory database of all known pkg-config files at runtime and then resolving dependencies from that database).

Patch:

- pkgconf-2.0.3-run-autogen-1.patch

As distributed, the pkgconf program is missing a lot of files that are normally produced by the GNU autotools; the conventional way to build it is to start by running the `autogen.sh` script provided with the package distribution. However, one of the places the pkgconf program is built is as part of the host prerequisites; if that's being done, it's probably not a great idea to rely on the autotools being present at prerequisite build time. We can instead simply patch the source distribution to create the files that would normally be created via `autogen.sh`.

7.1.2. pkgconf (host-isolation phase)

This is built here because pkg-config is already present on the host system (as a prerequisite, if nothing else), and it probably knows about a lot of host system libraries. While building the scaffolding, we need to ensure that their build processes don't find (and try to link against) any of the host system libraries. Since the host system libraries are for a different machine architecture, this would cause build failures. We can do that by putting a new version of pkg-config, configured only to look in the scaffolding directory, at the head of the PATH while building them.

We could probably get by with a simple shell script that always just says "I couldn't find anything!" when asked for dependencies. On the other hand, it takes around ten seconds to build pkgconf and set it up, so why not just do that?

Configuration commands:

```
find . -exec touch -r README.md {} \;
./configure --prefix=/tmp/cblwork/cross-tools \
  --with-pkg-config-dir=/tmp/cblwork/sysroot/scaffolding/lib/pkgconfig \
  --with-system-libdir=/tmp/cblwork/sysroot/scaffolding/lib \
  --with-system-includedir=/tmp/cblwork/sysroot/scaffolding/include
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
ln -sf pkgconf /tmp/cblwork/cross-tools/bin/pkg-config
```

In addition to the pkg-config symlink, we create a symbolic link aarch64-cbl-linux-gnu-pkg-config so that any build process that tries to find dependencies using a target-system pkg-config program will find ours.

Installation commands:

```
ln -sf pkgconf /tmp/cblwork/cross-tools/bin/aarch64-cbl-linux-gnu-pkg-config
```

Chapter 8. Construction of a minimal bootable userspace

In this section, we're going to use our shiny new cross-toolchain to build all of the programs and libraries we'll need to get to a working target-architecture userspace. We call these components, collectively, the "scaffolding" because we're going to use them as a kind of staging area and framework from which we can construct the final CBL system.

It's useful to keep that purpose in mind! These programs will provide just enough of a userspace environment that, once we've got the target system booted, we'll be able to build the final system components using these as a foundation. That means we'll need, basically, all the stuff that we've already been using from the host system — the programs that let us build programs — and we also need programs that will let us work with partition tables, filesystems, and other low-level operating system concerns.

It's also useful to keep in mind that everything here is *ephemeral*. As soon as we get the target system booted, we'll use these scaffolding programs to build all of the stuff that make up the final CBL system, and then we're going to throw them away.

Dependencies

Ensuring isolation from the host system.

8.1. About the Scaffolding

To maintain a hard line of separation between the scaffolding and the final system components, we're building all of this stuff so that it installs into a directory called `/scaffolding`. When we set up the root filesystem for the target, it's only going to have that one top-level directory! Then, as the scaffolding programs are used to construct the final system components, those final-system programs will be installed to normal system directories — `/bin`, `/usr`, and so on — and will be used in preference to the scaffolding programs that they replace.

In most cases, the scaffolding components use the GNU build system. That means they can be configured to expect that they will live in `/scaffolding`, but then be installed with a `DESTDIR` of the sysroot directory. That way, they actually get installed to `/tmp/cblwork/sysroot/scaffolding` (which is exactly where they need to be on the host system), but *think* they're installed to a top-level `/scaffolding` directory — which is where they actually will live once we boot into the target device.

Building everything with a very simple environment is still a good idea.

Environment variable: PATH

`/tmp/cblwork/cross-tools/bin:$PATH`

Environment variable: LC_ALL

`POSIX`

Some of the scaffolding pieces install libraries and headers. We want those to be visible to the rest of the scaffolding, so CFLAGS and LDFLAGS are not as empty as they have previously been.

Environment variable: CFLAGS

```
-I/tmp/cblwork/sysroot/scaffolding/include
```

Environment variable: CXXFLAGS

```
-I/tmp/cblwork/sysroot/scaffolding/include
```

Environment variable: LDFLAGS

```
-L/tmp/cblwork/sysroot/scaffolding/lib
```

To use the cross-toolchain for these builds, we need to define a bunch of additional environment variables. Many of the scaffolding programs use the GNU build system, and therefore consult these environment variables to determine how to invoke toolchain programs.

Environment variable: CC

```
aarch64-cbl-linux-gnu-gcc
```

Environment variable: CXX

```
aarch64-cbl-linux-gnu-g++
```

Environment variable: AR

```
aarch64-cbl-linux-gnu-ar
```

Environment variable: AS

```
aarch64-cbl-linux-gnu-as
```

Environment variable: RANLIB

```
aarch64-cbl-linux-gnu-ranlib
```

Environment variable: LD

```
aarch64-cbl-linux-gnu-ld
```

Environment variable: STRIP

```
aarch64-cbl-linux-gnu-strip
```

We're also going to use the cross-toolchain options `build`, `host`, and `target` for most of the components we're building in this section. This time, `--build` is going to be the host system; `--host` and `--target` are going to refer to the target system.

8.2. Create Symbolic Links For Scaffolding Lib Directories

Some target architectures have a "multilib" feature, and the installation process for many packages insists on installing library files into a variety of different directories (like `lib32` and `lib64`) to support this feature — even when multilib is disabled, as we try to do throughout the CBL process. The issue with this is, not all of the multilib directories are on the default library path known by the dynamic loader; this leads to errors in some package builds.

To ensure that all library files wind up in the `lib` directory *per se* rather than a multilib variant

directory, we use a ugly but simple and effective kludge: we create symbolic links to ensure that all library files are placed directly into `/scaffolding/lib`. (In the target-side build, we'll do something similar for the `/lib` and `/usr/lib` directories: this is done in Write the Scaffolding Init Script.)

The set of symbolic links needed here might expand as additional targets are added to the set that CBL can handle.

Commands:

```
mkdir -p /tmp/cblwork/sysroot/scaffolding/lib
ln -s lib /tmp/cblwork/sysroot/scaffolding/lib32
ln -s lib /tmp/cblwork/sysroot/scaffolding/lib64
ln -s lib /tmp/cblwork/sysroot/scaffolding/libx32
```

8.3. attr

Name	Filesystem Extended Attributes programs
Version	2.5.1
Project URL	http://savannah.nongnu.org/projects/attr
SCM URL	*(unknown)*
Download URL	http://download.savannah.nongnu.org/releases/attr/

8.3.1. Overview

Most Linux filesystems support "extended attributes" — arbitrary name/value pairs that can be associated with files or directories. These can be used for any purpose; for example, you might attach a "user.file_encoding" extended attribute to a text file, if it's encoded unusually.

One of the primary uses of extended attributes is to implement access control lists or capabilities.

The attr package provides programs that allow extended attributes to be viewed and modified.

8.3.2. attr (host-scaffolding-components phase)

Since the configuration repository preserves extended attribute metadata as well as basic owner and mode, and since the package-users build script sets a `user.package_owner` attribute on all files that are installed by packages, we need the `getfattr` and `setfattr` commands early in the target system build.

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.4. bash

Name	GNU Bourne Again SHell
Version	5.2.15
Project URL	http://www.gnu.org/software/bash/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/bash/

8.4.1. Overview

Bash is a shell — it provides a command line prompt, parses commands and pipelines entered on the command line, executes programs based on those commands and pipelines, and then does it all again. If you're interacting with a Linux system, and you're not using a graphical environment like Xorg, you're using a shell like bash. And, probably, you're using bash — there are other shells, but they are not nearly as common.

The bash maintainers don't provide all patchlevels for convenient download; you may have to download the tarfiles for the main version (like 4.3) and then separately download and apply all the update patches. The files.freesa.org repository, for convenience, has a tarfile that includes all of the patches for the version specified in this blueprint.

8.4.2. bash (host-scaffolding-components phase)

Several of the tests run by the configure script don't work right when bash is being cross-compiled. As with some other programs that use the GNU build system, we can short-circuit those tests by pretending that the configure script was run previously and knows the results from those tests.

Bash includes its own memory allocation routine, but historically it has not always been reliable. We use `--without-bash-malloc` to disable it so that the C standard library's malloc is used instead. For the scaffolding version of bash, we also configure it to be linked statically — that reduces the number of pieces that are needed to get the minimal scaffolding-based system to boot.

Configuration commands:

```
echo "ac_cv_func_mmap_fixed_mapped=yes" > config.cache
echo "ac_cv_func_strcoll_works=yes" >> config.cache
echo "ac_cv_func_working_mktime=yes" >> config.cache
echo "bash_cv_func_sigsetjmp=present" >> config.cache
echo "bash_cv_getcwd_malloc=yes" >> config.cache
echo "bash_cv_job_control_missing=present" >> config.cache
echo "bash_cv_printf_a_format=yes" >> config.cache
echo "bash_cv_sys_named_pipes=present" >> config.cache
echo "bash_cv_ulimit_maxfds=yes" >> config.cache
echo "bash_cv_under_sys_siglist=yes" >> config.cache
echo "bash_cv_unusable_rtsigs=no" >> config.cache
echo "gt_cv_int_divbyzero_sigfpe=yes" >> config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --enable-static-link --without-bash-malloc --cache-file=config.cache
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.5. binutils (host-scaffolding-components phase)

For an overview of binutils, see binutils.

Build Directory	`../build-binutils-3`

This is a new native set of the binutils, built using the cross-toolchain: they will run on the target architecture, and produce binaries that will also run on the target architecture. (But they're being built on the initial system, so we specify `build` as such.)

Build Directory

`../build-binutils-3`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --target=aarch64-cbl
```

```
-linux-gnu \
  --disable-nls --enable-shared --disable-multilib \
  --with-lib-path=/tmp/cblwork/sysroot/scaffolding/lib \
  --enable-64-bit-bfd
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.6. m4

Name	GNU M4
Version	1.4.19
Project URL	http://www.gnu.org/software/m4/m4.html
SCM URL	git://git.sv.gnu.org/m4
Download URL	https://ftp.gnu.org/gnu/m4/

8.6.1. Overview

M4 is a macro processor: it copies its standard input to standard output, expanding macros as it goes. It has a set of built-in macros that it understands, and it's possible to add user-defined ones as well. M4 is used extensively in the GNU build system, primarily by Autoconf.

8.6.2. m4 (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.7. bison

Name	GNU Bison
Version	3.8.2
Project URL	https://www.gnu.org/software/bison/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/bison/
Dependencies	m4

8.7.1. Overview

Bison is a parser generator. That means it takes a description of a language (in a specialized format called a "context-free grammar") and converts it into a parser for that language. This is mostly useful when writing compilers: the parser is the part of a compiler that takes a stream of tokens and figures out what syntax elements they represent. Parsers are fiddly and difficult to get correct, so it's common to generate the code for parsers using tools like Bison.

The name "Bison" is kind of a joke. The original parser generator that was commonly available on UNIX systems is called "Yacc," which is an acronym for "Yet Another Compiler-Compiler." When the GNU project implemented their own parser generator, they called it "Bison" because "Yacc" sounds like "Yak."

Bison's installation process also creates a program called `yacc` that simply runs `bison` in yacc emulation mode.

8.7.2. bison (host-scaffolding-components phase)

The absolute path to the `m4` program is written into `bison`, so we need to make sure that the scaffolding `bison` will expect to find `m4` in the scaffolding directory. This adds a little complexity, though: the bison configuration script checks to see whether `lex` is available and, if it is, whether it is `flex`. `flex` uses `m4`, so if we tell the bison configuration that `m4` can be found in the scaffolding, flex will try to use that version of `m4`, which was built for the target system rather than the host system. As with other packages that use the GNU Build System, we can use config.cache to tell bison that flex is simply not available.

Configuration commands:

```
echo "ac_cv_prog_lex_is_flex=no" > config.cache
echo "ac_cv_prog_lex_root=no" >> config.cache
echo "ac_cv_lib_lex='none needed'" >> config.cache
M4=/scaffolding/bin/m4 ./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --cache-file=config.cache
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.8. bzip2

Name	Block-sorting compression utility
Version	1.0.8
Project URL	https://sourceware.org/bzip2/
SCM URL	git://sourceware.org/git/bzip2.git
Download URL	https://www.sourceware.org/pub/bzip2

8.8.1. Overview

Bzip2 is a compression program like gzip, but uses the Burroughs-Wheeler block sorting algorithm and Huffman coding. It is a lot slower than gzip for both compressing and decompressing, but compresses data much better in most cases.

Bzip2 doesn't use the GNU build system, so there isn't a configure script.

8.8.2. bzip2 (host-scaffolding-components phase)

There isn't really a configuration step for bzip2, but we'll use the configuration stage to modify the build process so it won't try to run the tests — they won't work when building with a cross-toolchain.

The regular Makefile only creates a static version of the library, which is fine; however, it doesn't compile that library with `-fPIC`, which causes problems later on when other programs try to use it.

We can add that directive directly in the Makefile.

Configuration commands:

```
mv Makefile Makefile.orig
sed -e 's@^\(all:.*\) test@\1@g' -e 's@^CFLAGS=@CFLAGS=-fPIC @' \
  Makefile.orig > Makefile
```

Because bzip2 doesn't use the GNU build system, we need to specify the cross-tools as arguments to `make`.

Compilation commands:

```
make CC="${CC}" AR=${AR} RANLIB=${RANLIB}
```

Test commands:

```
(none)
```

Installation commands:

```
make PREFIX=/tmp/cblwork/sysroot/scaffolding install
```

8.9. coreutils

Name	GNU Core Utilities
Version	9.4
Project URL	http://www.gnu.org/software/coreutils/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/coreutils/

8.9.1. Overview

The Core Utilities are basic file, shell, and text manipulation utility programs: things like cat, chmod, chown, cp, … stuff like that. You use these all the time.

8.9.2. coreutils (host-scaffolding-components phase)

A couple of the tests run by the configure script don't work right when the coreutils are being cross-compiled. As with other programs that use the GNU build system, we can short-circuit those tests by pretending that the configure script was run previously and knows the results from those tests.

Configuration commands:

```
echo "fu_cv_sys_stat_statfs2_bsize=yes" > config.cache
```

```
echo "gl_cv_func_working_mkstemp=yes" >> config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --enable-install-program=hostname --cache-file=config.cache
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
sed -i 's@^cu_install_program =.*@cu_install_program = install@' Makefile
make DESTDIR=/tmp/cblwork/sysroot install
```

8.10. diffutils

Name	GNU Diffutils
Version	3.10
Project URL	http://www.gnu.org/software/diffutils/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/diffutils/

8.10.1. Overview

Diffutils is a bundle of programs that find differences between files and help you to merge them together: `cmp`, `diff`, `diff3`, `sdiff`.

The most important program here is `diff`, which finds differences between files. It can find differences between two files, in the simplest case; but, more commonly, `diff` is used to find all the differences between all the files in two entire directory trees.

You can save the output from `diff` when used this way as a "patch file", and then use the `patch` program to take one of the directory trees and make it look just like the other one.

This is handy when distributing modified versions of software packages: if you've made a minor change to GCC, for example, and you want to submit that change to the GCC mailing list for consideration by the GCC maintainers, you can use `diff` between the original GCC source tree and your modified GCC source tree, and then include the output from `diff` in your email.

8.10.2. diffutils (host-scaffolding-components phase)

When cross-compiling, the configure script guesses that the `getopt` function provided by GNU glibc is not available and tries to use an internal one instead. This doesn't work, for some reason (possibly because of behavior changes in GCC 7). Since the glibc getopt *is* available, we can just tell the configure script not to guess about it.

Configuration commands:

```
echo "gl_cv_func_getopt_gnu='yes'" > config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --cache-file=config.cache
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.11. util-linux

Name	miscellaneous linux utilities
Version	2.39.2
Project URL	https://www.kernel.org/pub/linux/utils/util-linux/
SCM URL	*(unknown)*
Download URL	*(unknown)*

8.11.1. Overview

util-linux is a grab-bag of miscellaneous Linux utility programs, for all kinds of things: disk partitioning, filesystem creation and validation and mounting... all kinds of stuff like that. (I'd expect a lot of this stuff, like `more`, `mount`, and `umount` — to be a part of the GNU system, perhaps in the coreutils package, but that's not the only thing I find surprising about the GNU/Linux world.)

8.11.2. util-linux (host-scaffolding-components phase)

We configure with an option that tells the build machinery not to chgrp the `wall` program to the tty group. This will probably make `wall` non-functional, but that's perfectly okay — `wall` can be used to

send messages simultaenously to all users logged on to the system, as for example if the system is going to be shut down or something of the sort, which is not very useful on single-user computers and certainly not necessary in the minimal `scaffolding` userspace.

You'll notice that we're configuring this package with a prefix of `/tmp/cblwork/sysroot/scaffolding` instead of just `/scaffolding` (and installing without a `DESTDIR`). That's because, when we configure and install util-linux the way we're doing most of the scaffolding components, we've sometimes encountered problems where some components can't find the `libuuid` library that is provided by this package. There are probably other ways to address that, and it's not really clear whether those components are strictly necessary, but this non-standard configuration scheme doesn't cause any problems and works fine.

The installation routine for util-linux changes the ownership and mode of some programs (like `mount` and `umount`) to be setuid to root. This doesn't work when running the installation as a normal user, as is being done in this section, so we disable that.

Configuration commands:

```
./configure --prefix=/tmp/cblwork/sysroot/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --disable-use-tty
-group \
  --without-ncurses --without-ncursesw --disable-makeinstall-chown \
  --disable-makeinstall-setuid
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

8.12. e2fsprogs

Name	Ext2/3/4 Filesystem Utilities
Version	1.47.0
Project URL	http://e2fsprogs.sourceforge.net/
SCM URL	git://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git
Download URL	https://mirrors.edge.kernel.org/pub/linux/kernel/people/tytso/e2fsprogs/
Dependencies	util-linux

8.12.1. Overview

The ext2fsprogs are utilities for managing the ext2 (and later) filesystems. ext2 was the first popular filesystem for Linux systems and its latest incarnation, ext4, is the most common. This package contains programs for creating filesystems, reconfiguring them, checking them for problems, that sort of thing.

There's some overlap between e2fsprogs and util-linux: both provide `libuuid` and `libblkid` libraries, a `uuidd` daemon, and a `fsck` script. We skip those from e2fsprogs.

Build Directory

`build`

8.12.2. e2fsprogs (host-scaffolding-components phase)

Build Directory	`build`

There is a glitch in the configuration logic that creates a file outside of the `DESTDIR`-specified path when there is no `/etc/cron.d` directory. This can be avoided by explicitly specifying that there is no crond directory at all.

Configuration commands:

```
../configure --prefix=/scaffolding \
  --enable-elf-shlibs --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --disable-libblkid --disable-libuuid --disable-fsck --disable-uuidd \
  --without-crond-dir
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
make DESTDIR=/tmp/cblwork/sysroot install-libs
```

8.13. expat

Name	Expat XML parser
Version	2.5.0
Project URL	https://libexpat.github.io/

SCM URL	*(unknown)*
Download URL	https://github.com/libexpat/libexpat/releases

8.13.1. Overview

Expat is a library that provides a stream-oriented XML parser. It's used by all sorts of other programs.

8.13.2. expat (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --without-docbook
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.14. file

Name	file type guesser
Version	5.45
Project URL	http://www.darwinsys.com/file/
SCM URL	https://github.com/file/file
Download URL	http://ftp.astron.com/pub/file/

8.14.1. Overview

`file` guesses file types by looking for particular characteristic byte values within them. If you've got a file and you're not sure what it actually is, you can run `file` on it and be told things like "that's a text file," or "that's a JPEG image."

On Microsoft Windows, file types are inferred based on the extension of the file — `exe` files are executable programs, `jpg` programs are JPEG images, and so on. `file` does things differently: it looks at the content of the file and tries to determine the file type by looking for characteristic patterns

within it, using a compiled database of these patterns called the "magic file," which you can read more about with `man 5 magic`. The magic file is produced during the build process by running `file -C` (with the freshly-built `file` program) to compile a bunch of "magic fragment" files into the database that it normally uses.

CAUTION	This use of `file` to compile the magic database at build time causes some complexity in cross-compilation scenarios. Since a cross-compiled `file` cannot run on the build server, the magic database is *copied from the build system* when the package is cross-compiled, rather than being recompiled. The build process checks to make sure that that magic database was produced by the exact same version of `file` that is being cross-compiled, and crashes if it was not. If the host-system `file` is not the same version being built here, you will need to fix that before starting the CBL process. This can be done by building the Trustworthy Host-System Programs, or by upgrading the host-system `file` so it matches the version being built here.[1]

8.14.2. file (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.15. findutils

Name	GNU Find Utilities
Version	4.9.0
Project URL	http://www.gnu.org/software/findutils/
SCM URL	https://git.savannah.gnu.org/git/findutils.git
Download URL	https://ftp.gnu.org/pub/gnu/findutils/

Patches	• findutils-4.9.0-add-xattr-predicate-1.patch

8.15.1. Overview

The Find Utilities are basic directory searching programs: mostly `find` and `locate`. This package also includes `updatedb`, which creates the file name database that `locate` uses; and `xargs`, which takes lists of file names produced by `find` and generates command lines that operate on all the files in those lists.

Patch:

- findutils-4.9.0-add-xattr-predicate-1.patch

The GNU version of `find` has no support for extended attributes, but since Little Blue Linux uses extended attributes to preserve the name of the package owning files even when they are `chown`ed or `chgrp`ed, we need that feature for use in some of the package user scripts. I found a patch from Nicolas Iooss that adds this functionality.

8.15.2. findutils (host-scaffolding-components phase)

A couple of the tests run by the configure script don't work right when the findutils are being cross-compiled. As with some other programs that use the GNU build system, we can short-circuit those tests by pretending that the configure script was run previously and knows the results from those tests.

Configuration commands:

```
echo "gl_cv_func_wcwidth_works=yes" > config.cache
echo "ac_cv_func_fnmatch_gnu=yes" >> config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --cache-file=config.cache
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.16. gawk

Name	GNU awk
Version	5.3.0
Project URL	http://www.gnu.org/software/gawk/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/gawk/

8.16.1. Overview

AWK is a special-purpose programming language that makes it easy to do simple text manipulations. (The name is an acronym of the three people who designed the language: Alfred Aho, Peter Weinberger, and Brian Kernighan.) Gawk is the GNU implementation of the AWK language.

If you're comfortable in perl or python or ruby, you probably won't have much use for awk; on the other hand, if you're working in a bash script, awk might turn out to be very handy.

8.16.2. gawk (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.17. gmp (host-scaffolding-components phase)

For an overview of gmp, see gmp.

Build Directory	`../build-gmp-3`

Although this is a cross-build of GMP, GMP doesn't like having a `target` specified, just `build` and `host`.

The manual explains that this is because `target` is for toolchain programs, and specifies the kind of machine where programs they produce will run; GMP doesn't produce programs, it's just a library, so `target` is meaningless for it.

Depending on the target architecture and ABI, it may be necessary to specify additional configure variables or arguments. (You'll know that you need one if the build fails immediately.) The `TARGET_GMP_CONFIG` parameter is available for that purpose.

The configure for this and the other GCC dependencies doesn't use the `--prefix` and `DESTDIR` trick to install to a different directory than the prefix, because that doesn't work well with the layered dependencies of mpfr and mpc later. That means that some of the files installed by these dependencies refer to host-system directories that won't exist on the target system; we'll fix those up later.

Build Directory

`../build-gmp-3`

Configuration commands:

```
${LB_SOURCE_DIR}/configure \
  --prefix=/tmp/cblwork/sysroot/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --enable-cxx
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

8.18. mpfr (host-scaffolding-components phase)

For an overview of mpfr, see mpfr.

Build Directory	`../build-mpfr-3`

Build Directory

`../build-mpfr-3`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/tmp/cblwork/sysroot/scaffolding \
```

```
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --target=aarch64-cbl
-linux-gnu \
  --with-gmp=/tmp/cblwork/sysroot/scaffolding
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

8.19. mpc (host-scaffolding-components phase)

For an overview of mpc, see mpc.

Build Directory	`../build-mpc-3`

Build Directory

`../build-mpc-3`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/tmp/cblwork/sysroot/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --target=aarch64-cbl
-linux-gnu \
  --with-gmp=/tmp/cblwork/sysroot/scaffolding \
  --with-mpfr=/tmp/cblwork/sysroot/scaffolding
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

8.20. isl (host-scaffolding-components phase)

For an overview of isl, see isl.

Build Directory	`../build-isl-3`

Build Directory

`../build-isl-3`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/tmp/cblwork/sysroot/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --target=aarch64-cbl
-linux-gnu \
  --with-gmp-prefix=/tmp/cblwork/sysroot/scaffolding
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

8.21. zlib

Name	zlib compression library
Version	1.3
Project URL	http://www.zlib.net/
SCM URL	*(unknown)*
Download URL	*(unknown)*

8.21.1. Overview

Zlib is a compression library. It implements the same Lempel-Ziv compression algorithm as gzip and info-zip, which means it doesn't compress data as effectively as most other algorithms (like the Burrows-Wheeler algorithm used by bzip2 or the LZMA algorithm used by xz-utils), but on the other hand it doesn't use very much memory to compress, and it's pretty fast.

There are a lot of programs that link against zlib to get basic compression capabilities. We're not

totally sure which components will fail to build if it's not available, but zlib itself has no dependencies and is really fast and small to build, so making it available is no big deal.

8.21.2. zlib (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.22. gcc (host-scaffolding-components phase)

For an overview of gcc, see gcc.

Environment	• unset `CFLAGS` • unset `CXXFLAGS` • unset `LDFLAGS`
Build Directory	`../build-gcc-4`

Dependencies

zlib.

As with binutils, this is going to produce a native compiler that will be used on the target system to let us build the pieces of scaffolding that are tricky to cross-compile, and the first few components of the final CBL system. The final system binutils and GCC are among those first components; as soon as we have those built and installed, we won't be using this compiler any more.

IMPORTANT	This is one of the checkpoint steps. If there's any problem with the cross-toolchain or the way it's set up, this target-native GCC build will probably fail. If you can get past this step, you can relax a little bit!

For this build, we need to ensure that some of the `FLAGS` environment variables are empty; otherwise, the build system can get confused and try to use target-system header files to compile

programs to be run on the host system. That doesn't work!

(You might be wondering why GCC builds programs for the host system when doing a target-native build. It's because the build process constructs some programs that it then immediately runs; those programs produce more source code as output, which then gets compiled for the target environment. Compiler build systems are complicated.)

Environment variable: CFLAGS

(should not be set)

Environment variable: CXXFLAGS

(should not be set)

Environment variable: LDFLAGS

(should not be set)

In-tree compilation of all the various dependencies is problematic for this build: for some architectures, like ARM and Sparc32, when GCC does the in-tree configuration of GMP, it configures it with a `host` and `target` starting with `none-`. That apparently was a working configuration in old versions of GMP, but current versions break when this is done and — again, only on certain CPU architectures! — result in errors when trying to link ISL a few steps later on.

As of this writing, this is an open issue. There are several ways this could be fixed or worked around: for example, GCC could be patched (specifically, the configure-gmp targets in `Makefile.def` and the `Makefile.in` generated from it, where they specify the invalid `host` and `target` parameters — probably, there is a better way to tell GMP to disable asm optimizations). For CBL, we sidestep this and other issues simply by avoiding in-tree builds for the dependency libraries.

Build Directory

`../build-gcc-4`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --target=aarch64-cbl
-linux-gnu \
  --with-local-prefix=/tmp/cblwork/sysroot/scaffolding --with-system-zlib \
  --with-native-system-header-dir=/tmp/cblwork/sysroot/scaffolding/include \
  --enable-languages=c,c++ --enable-checking=release \
  --disable-multilib --disable-nls --disable-libssp \
  --with-gmp=/tmp/cblwork/sysroot/scaffolding \
  --with-mpfr=/tmp/cblwork/sysroot/scaffolding \
  --with-mpc=/tmp/cblwork/sysroot/scaffolding \
  --with-isl=/tmp/cblwork/sysroot/scaffolding \
  --enable-fix-cortex-a53-835769 --enable-fix-cortex-a53-843419 --with-cpu=cortex
-a72.cortex-a53 --enable-standard-branch-protection
```

Compilation commands:

```
make GCC_FOR_TARGET="aarch64-cbl-linux-gnu-gcc" \
  CC_FOR_TARGET="aarch64-cbl-linux-gnu-gcc" CXX_FOR_TARGET="aarch64-cbl-linux-gnu-g++"
\
  AS_FOR_TARGET="aarch64-cbl-linux-gnu-as" LD_FOR_TARGET="aarch64-cbl-linux-gnu-ld"
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.23. gettext

Name	GNU Gettext
Version	0.22.3
Project URL	http://www.gnu.org/software/gettext/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/pub/gnu/gettext/

8.23.1. Overview

Gettext is a set of tools that can be used by other programs to provide internationalization and localization capabilities. This lets a single program provide user interfaces and user-facing messages in a variety of languages without requiring it to be rebuilt.

8.23.2. gettext (host-scaffolding-components phase)

One of the tests run by the configure script doesn't work right when the coreutils are being cross-compiled. As with some other programs that use the GNU build system, we can short-circuit those tests by pretending that the configure script was run previously and knows the results from those tests.

If the emacs editor is available on the system, the gettext build will do something with it. I don't really understand what it is, but it generates a couple of gigabytes worth of log output and makes the build process take orders of magnitude longer than it otherwise would. Luckily, it is easy to disable that with a configuration flag.

Configuration commands:

```
echo "gl_cv_func_wcwidth_works=yes" > config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
```

```
  --cache-file=config.cache --without-emacs
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.24. grep

Name	GNU grep
Version	3.11
Project URL	http://www.gnu.org/software/grep/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/grep/

8.24.1. Overview

Grep searches input files for lines that match patterns called "regular expressions." It often prints those lines out as well.

We have heard that the program name "grep" was originally derived from the `ed` command "g/re/p", which does basically the same thing as the command-line `grep` program, and wikipedia makes this claim as well. Maybe it's true!

8.24.2. grep (host-scaffolding-components phase)

The GNU libc library provides regular expression functions, but grep ignores them by default and uses its own bundled regex library. We use a configuration flag to override that behavior.

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --without-included-regex
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.25. gzip

Name	GNU zip compression utility
Version	1.13
Project URL	http://www.gnu.org/software/gzip/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/gzip/

8.25.1. Overview

Gzip is a compression program that basically does the same thing as the zlib library, but with a command-line interface.

8.25.2. gzip (host-scaffolding-components phase)

Environment	• `CFLAGS`: `$CFLAGS -Wno-error=format-truncation`

The default `-Werror` setting for gzip causes a format-truncation message produced by GCC 8 to abort the build. I am not all that concerned about format-truncation problems when printing error messages, so I just disable that setting.

Environment variable: CFLAGS

```
$CFLAGS -Wno-error=format-truncation
```

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

The generated Makefiles for gzip include a `-Wabi` directive that GCC 8 complains about. We can just remove that directive.

Configuration commands:

```
find . -name Makefile | while read filename; \
  do \
  sed -i -e 's@-Wabi@@g' $filename; \
  done
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.26. kbd

Name	Keyboard Utilities
Version	2.2.0
Project URL	http://ftp.altlinux.org/pub/people/legion/kbd/
SCM URL	*(unknown)*
Download URL	*(unknown)*

8.26.1. Overview

The Kbd package contains key-table files and keyboard utilities.

The `vlock` program requires Pluggable Authentication Modules (Linux-PAM), which is not part of the basic CBL system, so we always disable it. Conversely, other programs that aren't built by default might be helpful, so we enable those.

8.26.2. kbd (host-scaffolding-components phase)

We need some of the programs provided by kbd (notably, `openvt`, which will let us run interactive shells on virtual terminals in the minimal target system) in the minimal userspace.

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --disable-vlock --enable-optional-progs
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.27. libffi

Name	Foreign Function Interface library
Version	3.4.4
Project URL	https://sourceware.org/libffi/
SCM URL	git://github.com/libffi/libffi
Download URL	https://github.com/libffi/libffi/releases

8.27.1. Overview

Libffi is a portable library that helps programs written in one language to invoke functions that were written in a different language.

8.27.2. libffi (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.28. libressl

Name	LibreSSL
Version	3.8.1
Project URL	http://www.libressl.org/
SCM URL	git://github.com/libressl-portable/
Download URL	https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/

8.28.1. Overview

LibreSSL is a fork of OpenSSL with the goals of modernizing the codebase, improving the reliability and security of the code, and applying better development practices.

As an added bonus, the LibreSSL package includes a program called `nc` (short for "netcat") that can be used as a simple TCP and UDP client or server program — this can be enormously handy if you just want to shove data over a network connection, or test network client or server programs, or something like that. You can `man 1 nc` if you'd like to see more about it.

8.28.2. libressl (host-scaffolding-components phase)

LibreSSL should not be needed in the scaffolding, but the current version of rubygems has issues when used in a ruby installation without openssl support.

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.29. libxcrypt

Name	libxcrypt
Version	4.4.36

Project URL	https://github.com/besser82/libxcrypt
SCM URL	https://github.com/besser82/libxcrypt
Download URL	https://github.com/besser82/libxcrypt/releases

8.29.1. Overview

Generally, when you log in to a UNIX system, you provide a username and a password. Obviously, the password needs to be kept secret; I don't think there's any need to belabor that point. When you log in, the system — in this case, the `login` program — has to validate that the password you entered is correct. It does this by consulting an authentication database file.

If some nefarious person were to want to compromise the security of a system (I know it's shocking, but there *are* some nefarious people out there), they might try to do this by obtaining a copy of the database file. System designers and administrators try to keep that file locked down so that it's not feasible for nefarious people to make a copy of it, but some bad actors are very clever and find ways to do so. To make that situation less problematic, the database file does not actually contain the passwords that users enter. Instead, when users enter a password, the `passwd` program mangles the password using a *hashing function* that cannot be reversed, and stores the mangled, or "hashed," string rather than the actual password. When a user logs in, the `login` program uses the same hashing function on the password that they provide, and compares the hashed value to the hashed value in the password database. If they are the same, then `login` can conclude that the user entered the correct password.

libxcrypt is an implementation of these hashing functions.

Historically, these hashing functions — primarily `crypt` and `crypt_r` — have been provided by glibc. Rather than being part of the main shared library `libc.so`, these functions were put in a separate library `libcrypt.so`. However, in glibc release 2.38, this library is no longer built by default; it can be constructed by setting the `--enable-crypt` configure flag, but the recommendation in the glibc release notes is to switch to an alternative implementation like this one. Conveniently, `libxcrypt` builds a `libcrypt.so` library that is compatible with the one that has historically been provided by glibc.

8.29.2. libxcrypt (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.30. libyaml

Name	LibYAML
Version	0.2.5
Project URL	https://pyyaml.org/wiki/LibYAML
SCM URL	https://github.com/yaml/libyaml
Download URL	https://pyyaml.org/download/libyaml/

8.30.1. Overview

YAML — the name stands for "YAML Ain't Markup Language" — is a human-readable data serialization format. You can read all about it on https://yaml.org/. (Perhaps worthy of note, directives in litbuild blueprints are written in a very YAML-ish syntax.)

LibYAML is a library, originally written as part of the PyYAML project, for parsing YAML into data structures and emitting YAML from data structures. It's a convenient serialization format for cases where someone might need to look at the serialized data and understand what it is without any arcane hackery or tools.

The Ruby standard library includes a YAML implementation that depends on LibYAML.

8.30.2. libyaml (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.31. linux (host-scaffolding-components phase)

For an overview of linux, see linux.

Now we're going to build a kernel that will let us use the minimal userspace we constructed in `/scaffolding`.

Although the Linux kernel *per se* is entirely self-contained and doesn't use any functions defined in shared libraries, the kernel build process compiles a program `extract-cert` that requires a TLS library. Both LibreSSL and OpenSSL work fine for this. Presumably, the host system already provides this! If not, build and install LibreSSL first.

Kernel configuration is generally a manual business, and it is tricky to automate. The normal process I follow is to start with the configuration that produced the current kernel, then use `make olddefconfig` to define settings for any newly-added configuration keys, and then use one of the more friendly user interfaces (I usually use `nconfig`, a text-mode configuration program written by Nir Tzachar, because I am often working on a system that doesn't have a GUI set up and have gotten used to the text interfaces) to set individual options that I know I care about. Then, in this blueprint, I use the `config` script to set those configuration keys non-interactively.

It's always best to start with *some* kind of base configuration, regardless of what starting point makes the most sense for you, because there are thousands of configuration options in total. These fall into a few basic categories:

- *Device drivers,* which provide support for all the various types of I/O peripherals that might be part of your system — basically, this is everything other than the CPU *per se*;
- *Linux kernel features,* like support for various types of filesystem, cryptographic algorithms, I/O schedulers, and stuff like that; and
- *Other arbitrary settings* that let you specify things like the default computer name or command-line or a string to append to the kernel version.

The exact number of configuration options varies by architecture; for MIPS, there are about 9600 configuration options, while for 64-bit x86 computers there are almost eleven thousand, as of the 5.1 kernel. Regardless of architecture, there's enough complexity that starting from an empty configuration and enabling everything you need is not very much fun.

The most typical approach is to start with the configuration of the kernel that's currently running, which is often found in `/boot` and, even when not, can often be found in the `/proc` pseudo-filesystem as `/proc/config.gz`.

In this stage of CBL, though, the kernel on the host system is totally irrelevant for the scaffolding kernel we're building — it's not even for the correct machine architecture! So we're going to start with one of the default config files. You can look in `arch/arm64/config` to see all the default kernel config files you can use — or, just as easily, you can run `make ARCH=arm64 help` to get a summary of

the make targets that are available for the target architecture. In CBL, we use a litbuild configuration parameter to set the specific platform; the value that's set for this specific build is `defconfig`.

If there aren't any default configurations that are close to what you want for the target system, you can create a new one! To do this, just configure the kernel manually (using `make nconfig` or whatever you prefer); when you've got it set up the way you want it, use `make savedefconfig` to produce a new `defconfig` file for your configuration. To use this new configuration in CBL, move that `defconfig` file to `arch/arm64/configs/your_machine_defconfig` and generate a patch that adds just that file to the kernel source tree. Now you can add that patch to the Linux blueprint, so it will be applied automatically from the litbuild-produced script, and use `make your_machine_defconfig` to configure the kernel.

The standard CBL blueprint for Linux has an example of that whole process, in the [linux::aws-ami-bootable] phase.

Configuration commands:

```
make mrproper
make ARCH=arm64 defconfig
```

Starting from the default configuration, we now set a bunch of options that we definitely want for our minimal scaffolding environment. Notice that at this point we're just enabling options to be built directly into the kernel, rather than setting any features to be built as modules — once we get to the final kernel build, the stuff that we might or might not want to enable at runtime will be built as modules. In CBL, though, we're always going to build everything necessary to get userspace started directly into the kernel, instead of building an entirely-modular kernel like most of the binary distributions do. That means we won't have to set up an initramfs to provide an early userspace, which simplifies the boot process substantially.

This technique, of manually setting configure options that we want, is not really as robust as I'd like it to be. In many cases, config options are dependent on other config options — for example, you can't select POSIX access control lists for the JFFS2 filesystem unless you've enabled the JFFS2 filesystem. Ideally, manually enabling an option like `JFFS2_FS_POSIX_ACL` would cause any other required options, like `JFFS2_FS`, to be enabled as well. But, in fact, what happens is that any manually-configured options with unmet dependencies simply get disabled again automatically.

So the process I follow to obtain the set of options that are enabled here is: First, I run `make ARCH=arm64 nconfig` and manually set all the options that I care about. Then I compare the new version of `.config` to the one I started with; for all the modified options, I set up the calls to `scripts/config` below.

CUATION: The `STACK_VALIDATION` option is worth special note. If it's set, the kernel build process tries to compile a program called `objtool` that is used to analyze generated object files during the kernel build itself. However, it tries to use the *native* compiler with *target* system include files. That can't possibly work. The `RETPOLINE` option automatically reselects `STACK_VALIDATION`, so we have to turn that off as well. In the final target kernel, both of those will be enabled because they are potentially important for the security of the system, but at this stage it really does not matter.

Configuration commands:

```
./scripts/config --enable HIGHMEM
./scripts/config --set-str LOCALVERSION cbl1
./scripts/config --enable DEVTMPFS
./scripts/config --disable DEVTMPFS_MOUNT
./scripts/config --enable EXT4_FS
./scripts/config --enable IKCONFIG
./scripts/config --enable IKCONFIG_PROC
./scripts/config --disable STACK_VALIDATION
./scripts/config --disable RETPOLINE
./scripts/config --disable UNWINDER_ORC
./scripts/config --enable UNWINDER_FRAME_POINTER
./scripts/config --disable STACKPROTECTOR
```

For AMD64 (aka X86_64) builds, we want to support the x32 ABI. This is an architecture-specific option; it will be ignored for non-AMD64 builds.

Configuration commands:

```
./scripts/config --enable X86_X32
```

The options we're setting might cause other options to become available. For example, when `HIGHMEM` is enabled, `DEBUG_HIGHMEM` becomes a valid option. Since litbuild scripts are supposed to be entirely automated and non-interactive, we need to do something that prevents the kernel configuration machinery from asking questions about any such options. Luckily, there's a configuration target that starts with the existing configuration and then uses the default settings for any newly-available symbols. That's exactly what we need.

Configuration commands:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-cbl-linux-gnu- olddefconfig
```

(Incidentally, there's also a Makefile target `listnewconfig` that will display the configuration settings that will be prompted about if they're not specified. That might be useful in preventing any attempts at interactive configuration questions, but so far I haven't figured out a way to make use of it.)

GCC 8 adds a bunch of compiler warnings about aliasing between functions with possibly-incompatible types. In some architectures, such as mipsel, this triggers warnings in a bunch of `SYSCALL_DEFINE` macros. By default, the kernel build aborts when it sees these warnings, but we can avoid this by tweaking the kernel build process to disable that specific compiler warning.

Configuration commands:

```
echo "KBUILD_CFLAGS += -Wno-error=attribute-alias" >> Makefile
```

For some reason, the `STACKPROTECTOR` setting from earlier can get lost at some point during the

configuration process. We don't want to be asked about that again when we run `make all`, so we disable it (again) here.

Configuration commands:

```
./scripts/config --disable STACKPROTECTOR
```

Similarly, a couple of additional options can magically appear at this point in the build. I haven't spent a lot of time trying to figure out why that happens; it's easy enough simply to disable them here.

Configuration commands:

```
./scripts/config --disable EFI_STUB
./scripts/config --disable KCSAN
./scripts/config --disable SHADOW_CALL_STACK
```

For ARM-architecture builds, there are a few additional settings that we'll be prompted about during the build unless we specify them here. I really don't understand why, but again, manually configuring these is not hard. (As with other architecture-specific options, these settings will be ignored entirely on non-ARM builds.)

Configuration commands:

```
./scripts/config --enable ARM64_PTR_AUTH
./scripts/config --enable ARM64_PTR_AUTH_KERNEL
./scripts/config --enable ARM64_BTI
./scripts/config --enable ARM64_BTI_KERNEL
./scripts/config --enable ARM64_E0PD
./scripts/config --enable ARM64_TLB_RANGE
./scripts/config --enable ARCH_RANDOM
./scripts/config --enable ARM64_MTE
```

Even after all the hoops we've jumped through to avoid any interactive prompts, I find that the `all` target sometimes still asks for more configuration decisions. Another `olddefconfig` can help with that.

Configuration commands:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-cbl-linux-gnu- olddefconfig
```

Now we can build the kernel and modules. Until Linux 4.18, it was possible to set `CROSS_COMPILE` as a configuration setting in `.config`, but that doesn't work any more so we have to put it on the `make` command line or in the environment. (Grumble, grumble.)

If the build fails because it is unable to find the header file `gelf.h`, that means you don't have the elfutils package installed. This is a build-time requirement for some architectures; if you need it but don't have it, you can fix that by setting up the Trustworthy Host-System Programs.

Compilation commands:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-cbl-linux-gnu- \
  HOSTLDFLAGS=-L/home/random/cbltools/lib all
make ARCH=arm64 CROSS_COMPILE=aarch64-cbl-linux-gnu- \
  HOSTLDFLAGS=-L/home/random/cbltools/lib Image.gz
```

Test commands:

```
(none)
```

The `install` Makefile target for Linux is a little bit weird. For most (but not all!) target architectures, it winds up running `boot/install.sh` from the relevant architecture directory. That `install.sh` script looks to see if there is anything executable called `installkernel` in the current user's `$HOME/bin` directory or in the system `/sbin` directory. If there is, it just exec's into that `installkernel` script or program. Otherwise, it installs the kernel image and `System.map` file itself. (The kernel is really the only thing you need. `System.map` is a symbol table that specifies the address in memory for every variable and function name contained within the kernel; it's useful when debugging kernel panics and "oopses.")

That `installkernel` scheme doesn't work very well for the scaffolding kernel, because if the host system provides an `/sbin/installkernel` script, it's very likely to do something distribution-specific that won't work for the cross-compiled scaffolding kernel.

In CBL, we avoid all of this confusion and complexity by bypassing the normal installation target entirely; we just copy things where we want them.

Installation commands:

```
mkdir -p /tmp/cblwork/sysroot/scaffolding/boot
make ARCH=arm64 \
  INSTALL_PATH=/tmp/cblwork/sysroot/scaffolding/boot \
  INSTALL_MOD_PATH=/tmp/cblwork/sysroot/scaffolding modules_install
export KERNELPATH=$(find . -name Image.gz -a -type f)
cp -v $KERNELPATH /tmp/cblwork/sysroot/scaffolding/boot/kernel
cp -v .config /tmp/cblwork/sysroot/scaffolding/boot/config
cp -v System.map /tmp/cblwork/sysroot/scaffolding/boot
unset KERNELPATH
```

8.32. make

Name	GNU make
Version	4.4.1
Project URL	http://www.gnu.org/software/make/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/make/

8.32.1. Overview

GNU make is a build automation program. To use make, you set up one or more configuration files — the primary one being called, by convention, `Makefile` — that declare recipes for producing intermediate and final program artifacts, and dependencies between artifacts, and the various targets that can be produced. At runtime, `make` looks to see what artifacts exist already and which source files have a timestamp indicating that they've changed after dependent artifacts were produced, figures out based on that analysis exactly which artifacts need to be rebuilt, and then executes the recipes that will produce those artifacts.

That's all pretty awesome!

The downside is that Makefiles are not particularly clear or readable, and for large projects they get pretty big and complicated. That's why there is a thing called "litbuild"!

But the vast majority of system components use `make` to automate their build process, so you really have to have it available regardless.

8.32.2. make (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.33. ncurses

Name	GNU new curses library
Version	6.4
Project URL	https://www.gnu.org/software/ncurses/ncurses.html
SCM URL	*(unknown)*
Download URL	ftp://ftp.invisible-island.net/ncurses/

Patches	• ncurses-6.4-branch-updates-20231028.patch

8.33.1. Overview

Ncurses ("new curses") is a library that provides terminal control features so programs can provide advanced text-based user interfaces. It's a free-software version of a similar library (which was simply called "curses") that was developed at Berkeley.

There are lots of programs, even just in the scaffolding we're setting up, that use ncurses. `bash` and `vim` are among them.

Patches are available in a version subdirectory of the overall package download location — for example, patches for ncurses 6.2 are in the "6.2" subdirectory of the ncurses package directory. Patches need to be applied in order; there are instructions in the README file there, but the basic idea is that you find the latest patch bundle, called something like `ncurses-6.1-20181020-patch.sh`, and run that as a shell script; then you apply all the patch files later than that in sequence. When I update ncurses for CBL, I fetch all the patches available at that point, verify their GPG signatures, and compile them into a single branch-update patch, available in the file repository at files.freesa.org.

Patch:

- ncurses-6.4-branch-updates-20231028.patch

8.33.2. ncurses (host-scaffolding-components phase)

When configuring the scaffolding ncurses, we specify `--without-ada`, since Ada isn't built in the CBL process. We also specify `--enable-overwrite` so that the header files are installed into `/tmp/cblwork/sysroot/scaffolding/include` rather than an ncurses subdirectory of it, because that makes it easier for other programs to find the headers, and `--with-build-cc` so that the native tools built as part of the ncurses build process use the native compiler instead of the cross-compiler. We also specify `--disable-stripping` because otherwise the `install` program might try to strip binaries using the host system `strip` rather than the cross-toolchain `strip`.

Configuration commands:

```
./configure --prefix=/scaffolding --with-shared \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu --enable-overwrite \
  --without-debug --without-ada --with-build-cc=gcc --disable-stripping \
  --with-versioned-syms
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.34. patch

Name	GNU patch
Version	2.7.6
Project URL	http://www.gnu.org/software/patch/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/patch/

8.34.1. Overview

Remember diffutils? The most important program included in diffutils is `diff`, which finds differences between two files or directory trees. Patch is the conceptual inverse of `diff` — it takes a file with all the differences found by `diff` as input (which is by convention called a "patch file"), and applies all of those differences as changes to files in a directory tree.

This is really handy. For example, the CBL repository contains a bunch of source tarfiles that were obtained directly from the project web sites; and it also contains patch files that can be applied to those source tarfiles to apply changes that we have found to be necessary or important when building a CBL system.

8.34.2. patch (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.35. pth

Name	GNU portable threads library
Version	2.0.7
Project URL	https://www.gnu.org/software/pth/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/pth/
Patches	• `pth-2.0.7-update-config-guess-1.patch` • `pth-2.0.7-linux-kernel-fix-1.patch`

8.35.1. Overview

Pth is a portable threads library; it provides cooperative priority-based scheduling for multiple threads within a program.

The build system for pth doesn't work well when parallelized, so we explicitly disable parallel `make` jobs by specifying `-j1`.

On some servers, the version of config.guess distributed with pth is too old to recognize the target triplet. It's easy enough to update it to the version distributed with GCC.

Patch:

- pth-2.0.7-update-config-guess-1.patch

In some cross-compilation scenarios, pth fails to compile because it considers versions of the Linux kernel later than 2.9 to be "braindead." I got a patch from https://bugzilla.redhat.com/attachment.cgi?id=591825 that corrects the problem.

CAUTION	This patch works by tweaking the configuration process so that specific kernel versions are treated as *not* braindead. This is done using a regular expression to see if the Linux kernel version string matches what it expects. This regular expression currently matches major kernel versions 3 through 6. When the Linux kernel version is updated to 7.x, this patch will need to be adjusted!

Patch:

- pth-2.0.7-linux-kernel-fix-1.patch

8.35.2. pth (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make -j1
```

Test commands:

```
(none)
```

Installation commands:

```
make -j1 DESTDIR=/tmp/cblwork/sysroot install
```

8.36. ruby

Name	Ruby
Version	3.2.2
Project URL	http://www.ruby-lang.org/en
SCM URL	*(unknown)*
Download URL	https://www.ruby-lang.org/en/downloads/
Dependencies	zlib, libffi, libressl, libyaml

8.36.1. Overview

Ruby is a programming language. There are several implementations of that language — rubinius and jruby are others — but the canonical one is the C implementation that was created by Yukihiro "Matz" Matsumoto, and that's also the best one for our purposes in CBL because it has the fewest upstream dependencies.

CBL includes Ruby as a core component because CBL itself is designed to be used with the litbuild program — CBL consists entirely of litbuild blueprints — and litbuild is written in Ruby.

On 64-bit ARM systems, the Ruby build process exposes an issue with GCC (documented at `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84521`) when compiled with `-fomit-frame-pointer` and an optimization setting of `-O1` or `-O2`. Since the default behavior for modern versions of GCC is to omit frame pointers, it's important on such systems to specify `-fno-omit-frame-pointer` when building Ruby. It's possible that other architectures have a simliar issue; if your ruby build crashes with a "stack smashing detected" error message, try adding that option to your builds as well. (The default configuration for CBL specifies `-fno-omit-frame-pointer` in the target-system `CFLAGS`.)

8.36.2. ruby (host-scaffolding-components phase)

Once we boot into the target system, we'll use litbuild to generate all the scripts that will build the final CBL components. That means we'll need ruby available!

As with the file package, ruby can have issues in cross-compilation scenarios unless there is a

native ruby installation of the same version. This doesn't *always* happen, but the installation of a scaffolding ruby 2.7.0 failed when the system ruby was version 2.6.5; if something similar goes wrong for you, a version mismatch is a good thing to check.

Dependencies

libressl.

The version of rubygems (about which see more below) distributed with ruby 2.7 and later really wants to load the ruby `openssl` library, which requires OpenSSL (or a fork of it, like LibreSSL).

Dependencies

libyaml.

Ruby 3.2 requires the native libyaml library to build psych, which is needed for the litbuild gem and possibly others.

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.37. sed

Name	GNU Stream Editor
Version	4.9
Project URL	http://www.gnu.org/software/sed/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/sed/

8.37.1. Overview

Sed is kind of like a text editor, but instead of allowing you to modify text files interactively (like ed and vim do), sed acts as a filter: it takes a text file as an input stream, makes changes to that input

stream, and produces the resulting modified text as an output stream. (The name, `sed`, means "Stream EDitor").

This is a fairly powerful and flexible thing to be able to do, but it's not something people often need to do — like `gawk`, `sed` is most often handy in the context of a bash script or something like that, where a more powerful general-purpose language like ruby or python or perl isn't convenient for whatever reason.

On the other hand, the automated build processes for some of the CBL components use sed, so you have to have it around anyway.

8.37.2. sed (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.38. tar

Name	GNU tar
Version	1.35
Project URL	http://www.gnu.org/software/tar/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/tar/

8.38.1. Overview

Tar is a program for manipulating archives of files. The name derives from its original purpose, back in the dawn of time, which was to manipulate archives on magnetic tape — ergo, "tar," for Tape ARchive. Actually storing archives on magnetic tape is pretty rare these days, but people still use tar-format files for all kinds of things.

These files are known as "tarfiles," for obvious reasons, or as "tarballs" for less obvious reasons. A speculation I found on Wikipedia is that the term references the tendency of actual tarballs (blobs of petroleum floating in the ocean) to get all sorts of things stuck to them; apparently, someone thought that was reminiscent of how the tar program collects a bunch of files together.

All of the source code distribution packages for CBL packages are compressed tarfiles, so the tar program is really important for CBL!.

The tar package also provides a program called "rmt" that lets you manipulate magnetic tape drives attached to other computers. This is an even more fringe thing to do in this day and age, but It only weighs in at about 50kb, so it’s not hurting anything.

8.38.2. tar (host-scaffolding-components phase)

Several of the tests run by the configure script don’t work right when tar is being cross-compiled. As with some other programs that use the GNU build system, we can short-circuit those tests by pretending that the configure script was run previously and knows the results from those tests.

Configuration commands:

```
echo "gl_cv_func_wcwidth_works=yes" > config.cache
echo "gl_cv_func_btowc_eof=yes" >> config.cache
echo "ac_cv_func_malloc_0_nonnull=yes" >> config.cache
echo "gl_cv_func_mbrtowc_incomplete_state=yes" >> config.cache
echo "gl_cv_func_mbrtowc_nul_retval=yes" >> config.cache
echo "gl_cv_func_mbrtowc_null_arg1=yes" >> config.cache
echo "gl_cv_func_mbrtowc_null_arg2=yes" >> config.cache
echo "gl_cv_func_mbrtowc_retval=yes" >> config.cache
echo "gl_cv_func_wcrtomb_retval=yes" >> config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --cache-file=config.cache
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

8.39. texinfo

Name	GNU texinfo
Version	7.1
Project URL	http://www.gnu.org/software/texinfo/
SCM URL	https://git.savannah.gnu.org/git/texinfo.git
Download URL	https://ftp.gnu.org/gnu/texinfo/

8.39.1. Overview

Texinfo is the official documentation system for the GNU project. It uses input files in a standard format to produce a variety of outputs, both printed and online.

Texinfo is used in the automated build process of many of the CBL component packages.

In release 7.1 of texinfo, the `install-info` command fails in some cases. I think this is because `install-info` now has code to ensure that the `/usr/share/info/dir` file is updated atomically: it writes the new version to a temporary location, then moves it over the original file, overwriting it. That doesn't work on LB Linux, because it uses the package user approach to package management, and `/usr/share/info`, like other install directories, has the sticky bit set. (If you don't know what any of that means, you can read the package-user manual — on Little Blue Linux systems, you can find it in `/usr/share/doc/package-users`.) For some reason I do not understand, many packages are not affected by this, but a few are. As I write this, so far, I've come across this with nettle and glibc, but perhaps there are others. Whenever this happens, I use a simple workaround: simply `chown` the `dir` file before installing the package, and `chown` it back afterward.

8.39.2. texinfo (host-scaffolding-components phase)

When cross-compiling, the configure script makes some incorrect guesses about the availability of functions provided by glibc. We can override those guesses in a `config.cache` file.

Even after doing that, the presence of the `gnulib` version of `getopt.h` causes the build to break. That's easy enough to work around simply by removing it.

Configuration commands:

```
echo "gl_cv_func_getopt_gnu=yes" > config.cache
echo "texinfo_cv_sys_iconv_converts_euc_cn=no" >> config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --cache-file=config.cache
rm -f gnulib/lib/getopt.h
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

The build process creates two scripts that expect perl to be in the location where it's found on the host system, rather than the place it will be available in the initial target system, so we fix them up after installation.

Installation commands:

```
sed -i -e 's@/usr/bin/perl@/scaffolding/bin/perl@' \
  /tmp/cblwork/sysroot/scaffolding/bin/makeinfo
sed -i -e 's@/usr/bin/perl@/scaffolding/bin/perl@' \
  /tmp/cblwork/sysroot/scaffolding/bin/texi2any
```

8.40. vim

Name	Vim
Version	9.0.2081
Project URL	http://www.vim.org/
SCM URL	https://github.com/vim/vim.git
Download URL	https://github.com/vim/vim/releases
Dependencies	ncurses

8.40.1. Overview

Back in the early days of Unix, someone wrote a text editor called "ed" (which stood for, get this, "EDitor"). Ed is a line editor, which means that generally it lets you modify a line at a time. Ed is still available, and in fact still actively developed — GNU ed 1.14.1 was released in January of 2017 — but most people want a more interactive editor, and so one of the old proprietary Unixes eventually included an editor called "vi" ("VIsual editor") to fill this need.

Since vi is proprietary, various people implemented similar programs, which they released under open source licenses of various sorts. One of the people who decided to do this is Bram Moolenaar, who created vim; its name stands for "Vi IMproved," because it's massively superior to vi. Bram Moolenaar died in August of 2023, but there were other contributors to the vim project who continue to develop the program.

Vim is distributed under a different software license than most of the packages that make up CBL: the Vim License. This license is GPL-compatible, but is described as "charityware" — users of vim are urged to make a donation for needy children in Uganda.

As far as that goes, you probably don't *need* to install vim at all. There are plenty of other text editors that some people prefer. The default editor in some distributions is nano, and some people like their text editor to be their primary tool for interacting with the computer and use emacs. Whatever your preferences, though, you absolutely do need a text editor of some sort, and vim is my favorite, so that's the one that winds up in the basic CBL blueprints.

8.40.2. vim (host-scaffolding-components phase)

As with many scaffolding pieces, the configure script for vim doesn't play nicely with cross-compilation, so we pre-fill a config.cache with a bunch of values that it won't be able to discover on its own.

An interesting difference between vim and most other packages we're building is that it automatically picks up entries from a config.cache file located in the `src/auto` directory, and apparently doesn't support providing a reference to config.cache in the top-level configure run.

Configuration commands:

```
mkdir -p src/auto
echo "vim_cv_getcwd_broken=no" > src/auto/config.cache
echo "vim_cv_memmove_handles_overlap=yes" >> src/auto/config.cache
echo "vim_cv_stat_ignores_slash=no" >> src/auto/config.cache
echo "vim_cv_terminfo=yes" >> src/auto/config.cache
echo "vim_cv_toupper_broken=yes" >> src/auto/config.cache
echo "vim_cv_tty_group=world" >> src/auto/config.cache
echo "vim_cv_tgetent=zero" >> src/auto/config.cache
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu \
  --enable-gui=no --disable-gtktest --disable-xim --disable-gpm \
  --without-x --disable-netbeans --with-tlib=ncursesw
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

There are a lot of programs that expect a text editor called "vi" to exist, so we'll create a symlink after installing vim.

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
ln -sv vim /tmp/cblwork/sysroot/scaffolding/bin/vi
```

8.41. xz

Name	XZ Utils LZMA compression program
Version	5.4.5
Project URL	http://tukaani.org/xz/
SCM URL	*(unknown)*
Download URL	*(unknown)*

8.41.1. Overview

The XZ Utils package provides a compression/decompression program with an interface similar to gzip, but using the LZMA algorithm. This is very slow when compressing, very fast when decompressing, and compresses more effectively than most of the other common compression programs, so it's used fairly often when providing archive files for download.

The compression algorithm used by the XZ Utils is the same as that used by `lzip`, but the file format is more complex and the compressed output it produces is usually slightly larger than that produced by `lzip`.

8.41.2. xz (host-scaffolding-components phase)

Configuration commands:

```
./configure --prefix=/scaffolding \
  --build=x86_64-unknown-linux-gnu --host=aarch64-cbl-linux-gnu
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make DESTDIR=/tmp/cblwork/sysroot install
```

Having constructed all the scaffolding we can build on the host system, it's time to set up for the second (target) stage of the build. The target-side build will be run automatically when the target system is booted.

[1] There's a `FIXME` comment in `magic/Makefile.am` that indicates that the `file` project developers consider this a bug that they may eventually fix, by building a native `file` as well as the cross-compiled `file` and using the native `file` to compile the magic database.

Chapter 9. Finish preparing the scaffolding for launch of the target system

Now that everything's built, we need to adjust some files that were incorrectly written.

9.1. Fix Scaffolding Libtool Wrapper Files

Libtool is a part of the GNU Build System, and is therefore used in the build machinery of many CBL components. One thing libtool does is create "wrapper libraries" with the `.la` extension; later invocations of libtool use those files to find the *real* location of library files.

The full host sysroot path winds up in many of the `.la` files in the scaffolding lib directory. This can cause problems when building the final CBL system (after booting into the target system), since those paths don't exist any more at that point. So it's a good idea to fix those paths after the scaffolding is otherwise complete.

Commands:

```
cd /tmp/cblwork/sysroot/scaffolding/lib
grep -l '/[a-zA-Z/]*sysroot/scaffolding/' *.la | while read FILE; \
  do \
  sed -i -e 's@/[a-zA-Z/]*sysroot/scaffolding/@/scaffolding/@g' $FILE; \
  done
```

Other libtool files wind up with cross-toolchain paths embedded in them. That won't work any better than the sysroot path does.

Commands:

```
grep -l '/tmp/cblwork/cross-tools/aarch64-cbl-linux-gnu' *.la | while read FILE; \
  do \
  sed -i -e 's@/tmp/cblwork/cross-tools/aarch64-cbl-linux-gnu@/scaffolding@g' $FILE; \
  done
```

And still other libtool files wind up with some path under the build directory. That's even worse, since in some cases the libraries won't even exist in the target system! But if they do exist, they'll be in `/scaffolding/lib`.

Commands:

```
grep -l -- '-L/tmp/cblwork/build' *.la | while read FILE; \
  do \
  sed -i -e 's@-L/tmp/cblwork/build/[^ ]* @-L/scaffolding/lib @g' $FILE; \
  done
```

We also need to copy all the source code and patches into the scaffolding so that we'll have it in the target system, and make the CBL blueprints available there.

9.2. Final Preparation of the Scaffolding

All of the programs and libraries needed to boot into the minimal target system are built and installed! But there are a few more things we'll need in the `/scaffolding` directory so we can finish the build.

For one thing, we're going to need the CBL blueprints! We could pre-create all the litbuild scripts that will be used in the target side of the CBL process, but it seems tidier and more elegant to use litbuild from within the target system. Also, while developing and debugging the target-side CBL process, it's handy to be able to modify the blueprints and re-generate scripts easily within the target environment.

Commands:

```
cp -a $LB_BLUEPRINT_DIR /tmp/cblwork/sysroot/scaffolding/cbl
```

We also need the source code and patches that will be used during the rest of the build process. We need to have that stuff available locally, rather than pulling it in from a network location, because the minimal scaffolding userspace has no way to access a network.

I'm making the assumption, here, that all of the source code in the `TARFILE_DIR` and `PATCH_DIR` locations will be needed in the target system. It's possible that there's extra stuff there that is *not* necessary, but the only consequence of that is that some extra, unnecesasry space is used in the scaffolding directory structure — not exactly a catastrophe.

Commands:

```
mkdir -p /tmp/cblwork/sysroot/scaffolding/materials
find /home/random/materials/ -type f -exec cp -n {} \
  /tmp/cblwork/sysroot/scaffolding/materials \;
if [ /home/random/materials != /home/random/materials ]; \
  then \
  find /home/random/materials/ -type f -exec cp -n {} \
  /tmp/cblwork/sysroot/scaffolding/materials \;; \
  fi
```

Since litbuild will be installed before anything else, we won't have lzip available at the time. The simplest thing to do is simply uncompress it now.

Commands:

```
lzip -d /tmp/cblwork/sysroot/scaffolding/materials/litbuild*tar.lz
```

And we need to provide an init for the target system to perform the automated build once that system is running.

9.3. Build and Install Entropy Adder

9.3.1. Overview

As mentioned elsewhere, you can add data to the entropy pool without any difficulty from any userspace process: all you have to do is write data to `/dev/urandom`. However, that doesn't help to initialize the entropy pool, because it doesn't increase the kernel's estimate of how much entropy is actually available. To do *that,* we have to execute a system call that is only available to processes running as the superuser.

The system call in question is `ioctl`, which is kind of a catch-all that exposes arbitrary functionality in device drivers. `ioctl` itself is short for "Input-Output Control." The idea is: since there is a huge variety of input/output devices that might be available to your system, and there's no way for Linux to provide specific system calls to exercise every distinct function of every one of those devices, any device driver can define a set of `ioctl` operations. Userspace applications can then use the `ioctl` system call to invoke any of those operations, using a file descriptor that references one of the device's special files under `/dev`.

The device driver for the `/dev/random` and `/dev/urandom` special files happens to provide an `ioctl` that lets you add data to the entropy pool. We're going to write a tiny C program that feeds some data to the kernel using that `ioctl`, so that the entropy pool will be fully initialized immediately.

Note that it is an *extremely* bad idea to use this program unless you are confident that the input data really is unpredictable, or unless (as in the case of the target-side CBL build) you are equally confident that there is no reason to be concerned about the quality of random data available to userspace processes.

In many cases, this program isn't needed in a full working system — the `rngd` daemon from the rng-tools package feeds the kernel entropy from hardware random number generators, like the one built into modern x86-architecture CPUs. But CBL supports multiple computer architectures, and some of those don't have anything suitable for rngd, so `addentropy` can be helpful in those cases.

The `addentropy` program is derived from `ec2seed`, at https://github.com/akkornel/ec2seed, which is available under the terms of the GNU GPL Version 3. Accordingly, there's no issue with distributing the addentropy source as part of CBL; all the code within CBL is similarly licensed under the GNU GPL Version 3.

As with almost all C programs, we start by including header files.

File /tmp/addentropy.c:

```
#include <errno.h>
#include <fcntl.h>
#include <linux/random.h>
#include <linux/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
```

```
#include <sys/types.h>
#include <unistd.h>
```

We define a constant for the number of bytes we're going to feed to the entropy pool. That's really not all that important, but it makes some later parts of the program slightly clearer.

File /tmp/addentropy.c (continued):

```
const int RANDOM_BYTE_COUNT = 1024;
```

Before and after adding data to the entropy pool, we can obtain the kernel's approximation of how much entropy it already contains, using another `ioctl` provided by the `/dev/random` device driver. Let's define a function for that.

File /tmp/addentropy.c (continued):

```
int entropy_available (const int fd)
{
    int num_bytes;
    if (ioctl(fd, RNDGETENTCNT, &num_bytes) == -1) {
        fprintf(stderr,
            "Unable to determine amount of entropy in the pool\n");
    }
    return num_bytes;
}
```

Now we can write the main routine for addentropy. As usual in C programs, we start by defining all of the variables we'll use.

File /tmp/addentropy.c (continued):

```
int main (int argc, char *argv[])
{
    __u32 *ebuf;
    int remaining, before, after, bytes_added, urandom_fd;
    ssize_t bytes_read;
    struct rand_pool_info *entropy;
```

We need to open a file descriptor on the `/dev/urandom` special file so that we can invoke `ioctl` operations on it. And we need to allocate a buffer that we can read the data in to.

File /tmp/addentropy.c (continued):

```
urandom_fd = open("/dev/urandom", O_WRONLY);
if (urandom_fd == -1) {
    fprintf(stderr, "Error opening random file\n");
    return 1;
}
entropy = malloc(sizeof(struct rand_pool_info) + RANDOM_BYTE_COUNT);
```

```
if (entropy == NULL) {
    fprintf(stderr,
        "Unable to allocate memory for the entropy buffer\n");
    return 1;
}
```

According to the kernel source file `random.c`, which is the source code for the random-number device driver, the `entropy_count` field of `rand_pool_info` structures is denominated in units that are an eighth of a bit each. I have no idea why. So maybe we should multiply `RANDOM_BYTE_COUNT` by 64 rather than by 8 here? But this appears to work perfectly well, so I'm not messing with it.

File /tmp/addentropy.c (continued):

```
entropy->entropy_count = RANDOM_BYTE_COUNT * 8;
entropy->buf_size = RANDOM_BYTE_COUNT;
ebuf = entropy->buf;
```

Now we need to obtain some data (which we will presume to be random). The way we're going to do that is simply read it from the standard input stream. Since we're using some low-level file manipulation functions, we have to deal with the possibility that any given call to `read` actually produces less data than we request; therefore, we set up a loop and `read` repeatedly until we have all the data we want.

File /tmp/addentropy.c (continued):

```
remaining = RANDOM_BYTE_COUNT;
do {
    bytes_read = read(STDIN_FILENO, ebuf, remaining);
    if (bytes_read < 0) {
        fprintf(stderr,
            "Error occurred while reading stdin: %s\n",
            strerror(errno));
        return 1;
    } else if (bytes_read > 0) {
        ebuf += bytes_read;
        remaining -= bytes_read;
    }
} while (bytes_read > 0 && remaining > 0);
```

Now we can add the data to the entropy pool, using the `RNDADDENTROPY` ioctl. We'll also print out a description of what happened.

File /tmp/addentropy.c (continued):

```
before = entropy_available(urandom_fd);
if (ioctl(urandom_fd, RNDADDENTROPY, entropy) == -1) {
    fprintf(stderr, "Error adding entropy to kernel\n");
    return 1;
}
```

```
after = entropy_available(urandom_fd);
bytes_added = RANDOM_BYTE_COUNT - remaining;
printf("Added %i bytes to the entropy pool.\n", bytes_added);
printf("Entropy available: %i -> %i\n", before, after);
```

There's nothing left to do other than clean up the resources we've allocated and terminate the program. Technically we don't have to clean up the resources — Linux will do that automatically as the process is reaped — but it's good practice to do it.

File /tmp/addentropy.c (continued):

```
    free(entropy);
    close(urandom_fd);
    return 0;
}
```

That's it!

The command we use to compile the program, and the location where it will wind up, depends on whether we're building this in the scaffolding or for the final target system.

9.3.2. Build and Install Entropy Adder (scaffolding phase)

For the scaffolding, we need to use the cross-compiler to compile `addentropy`, and it needs to wind up in the scaffolding bin directory.

Commands:

```
aarch64-cbl-linux-gnu-gcc -std=c89 -Wall -Wextra -Wpedantic -Werror \
  -Wno-unused-parameter -g -O2 \
  -o /tmp/cblwork/sysroot/scaffolding/bin/addentropy /tmp/addentropy.c
```

9.3.3. Complete text of files

9.3.3.1. /tmp/addentropy.c

```
#include <errno.h>
#include <fcntl.h>
#include <linux/random.h>
#include <linux/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
const int RANDOM_BYTE_COUNT = 1024;
```

```
int entropy_available (const int fd)
{
    int num_bytes;
    if (ioctl(fd, RNDGETENTCNT, &num_bytes) == -1) {
        fprintf(stderr,
            "Unable to determine amount of entropy in the pool\n");
    }
    return num_bytes;
}
int main (int argc, char *argv[])
{
    __u32 *ebuf;
    int remaining, before, after, bytes_added, urandom_fd;
    ssize_t bytes_read;
    struct rand_pool_info *entropy;
    urandom_fd = open("/dev/urandom", O_WRONLY);
    if (urandom_fd == -1) {
        fprintf(stderr, "Error opening random file\n");
        return 1;
    }
    entropy = malloc(sizeof(struct rand_pool_info) + RANDOM_BYTE_COUNT);
    if (entropy == NULL) {
        fprintf(stderr,
            "Unable to allocate memory for the entropy buffer\n");
        return 1;
    }
    entropy->entropy_count = RANDOM_BYTE_COUNT * 8;
    entropy->buf_size = RANDOM_BYTE_COUNT;
    ebuf = entropy->buf;
    remaining = RANDOM_BYTE_COUNT;
    do {
        bytes_read = read(STDIN_FILENO, ebuf, remaining);
        if (bytes_read < 0) {
            fprintf(stderr,
                "Error occurred while reading stdin: %s\n",
                strerror(errno));
            return 1;
        } else if (bytes_read > 0) {
            ebuf += bytes_read;
            remaining -= bytes_read;
        }
    } while (bytes_read > 0 && remaining > 0);
    before = entropy_available(urandom_fd);
    if (ioctl(urandom_fd, RNDADDENTROPY, entropy) == -1) {
        fprintf(stderr, "Error adding entropy to kernel\n");
        return 1;
    }
    after = entropy_available(urandom_fd);
    bytes_added = RANDOM_BYTE_COUNT - remaining;
    printf("Added %i bytes to the entropy pool.\n", bytes_added);
    printf("Entropy available: %i -> %i\n", before, after);
```

```
    free(entropy);
    close(urandom_fd);
    return 0;
}
```

9.4. Write the Scaffolding Init Script

As mentioned elsewhere (and often), the job of the Linux kernel is to initialize hardware, mount the root filesystem, and then run an `init` program (with PID 1) to start userspace. The `init` program does all the rest of the system startup process — mounting filesystems, spawning long-running daemon processes, perhaps starting a GUI environment.

There are a couple of other things to keep in mind about the process running with PID 1:

- First, it's *not allowed to terminate*. If it does, the Linux kernel will immediately panic and crash. So it's important for the init program to be extremely resilient.
- Second… this requires a bit of additional explanation. Whenever a process terminates, it becomes what is called a "zombie" process; the process that launched it, its "parent" process, is supposed to "reap"[1] it by collecting its exit code, whereupon it is removed from the process table. If that doesn't happen, the zombie process stays around indefinitely, cluttering up the process table and wandering around looking for brains to eat.[2] If the parent process terminates before a child process, the child becomes an "orphaned" process — it has no parent — and when *it* terminates, there's nothing to collect its exit status. That's where `init` comes in: any time a process is orphaned, the kernel sets its parent to PID 1. In addition to launching userspace processes, it's the responsibility of PID 1 to reap all these adopted processes when they terminate.

In CBL, we want the `init` program in the minimal target userspace to execute the target-side build process, which will result in a complete and functional (albeit bare-bones) GNU/Linux system. That means that it doesn't *have* to launch any interactive programs or processes. In fact, it doesn't even have to be a traditional `init` program at all! We take advantage of that flexibility by using a simple bash script as the `init` "program" — a plan which I think is sheer elegance in its simplicity! We already have `bash` available in the scaffolding, and we don't *want* anything in the target-side build to require any manual interaction or unnecessary complexity.

The downside to this approach is that we won't have a full GNU/Linux userspace available, which can make diagnosing problems more difficult. So we set up the init script so that if anything goes wrong, it drops to an interactive shell prompt — at that point, you can work at the console to figure out what's going on.

To make it clear that the script we're writing here is intended to run as `init`, we'll use the conventional `/sbin/init` path for it. (As with everything else on the initial target system, this path is within the `/scaffolding`, though.) The first line will tell the kernel how to execute it.

File /tmp/cblwork/sysroot/scaffolding/sbin/init:

```
#!/scaffolding/bin/bash
```

Commands:

```
chmod a+x /tmp/cblwork/sysroot/scaffolding/sbin/init
```

9.4.1. The Target-Side Build Environment

When the kernel runs `init`, it does so with a minimal (maybe empty?) environment. Bash does not work well unless some basic environment variables are set — like `PATH` and `LD_LIBRARY_PATH` for program and library lookups, for example — so we'll start by getting those set.

When problems occur during the target-side build, the `init` script bails out to an interactive shell that *also* has an empty environment. So it's convenient to have all the commands that set up the environment in a separate script that can be easily sourced in from that rescue shell. The same script can be sourced from `init` to avoid any duplication between the two.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
source /scaffolding/setenv.sh
```

File /tmp/cblwork/sysroot/scaffolding/setenv.sh:

```
export PATH=/scaffolding/bin:/scaffolding/sbin
export LD_LIBRARY_PATH=/scaffolding/lib
```

GNU/Linux systems have extensive support for internationalization and localization. Before we have the full system set up, it's a good idea to specify a simple default setting for those features.

File /tmp/cblwork/sysroot/scaffolding/setenv.sh (continued):

```
export LC_ALL=POSIX
```

As the final system programs and libraries are built, they should be used in preference to the ones in `/scaffolding`, so we'll put the directories where they'll live in front of the `/scaffolding` directories.

File /tmp/cblwork/sysroot/scaffolding/setenv.sh (continued):

```
export PATH=/usr/bin:/usr/sbin:$PATH
export LD_LIBRARY_PATH=/usr/lib:$LD_LIBRARY_PATH
```

The `TARGET_SYSTEM_MAKEFLAGS` parameter specifies the `MAKEFLAGS` that should be set for all target-side processes. (If a specific package has issues with parallel builds, it can be overridden to `-j1` or unset entirely for those packages.)

File /tmp/cblwork/sysroot/scaffolding/setenv.sh (continued):

```
export MAKEFLAGS="-j8"
```

There are some programs that expect `HOME` to be set to a reasonable value. We can go ahead and set it here in case any of the programs we're using is among those.

File /tmp/cblwork/sysroot/scaffolding/setenv.sh (continued):

```
export HOME=/root
```

We also need to set environment variables for all the litbuild configuration parameters that might not be using default parameter values. It's not hard to define these properly, because from this point onward we don't need to worry about `HOST` and `TARGET` and so on: everything is just going to be a native build. We do need to pass a number of parameters from the host-side build along through to the target-side build, though!

File /tmp/cblwork/sysroot/scaffolding/setenv.sh (continued):

```
export BOOT_DEVICE=''
export BOOTLOADER='manual'
export HOST_NAME='cblviton'
export DOMAIN_NAME='freesa.org'
export KERNEL_TARGET='Image.gz'
export LOGIN_FULL_NAME='A Little Blue User'
export LOGIN='lbl'
export TARGET_GCC_CONFIG='--enable-fix-cortex-a53-835769 --enable-fix-cortex-a53
-843419 --with-cpu=cortex-a72.cortex-a53 --enable-standard-branch-protection'
export TARGET_SYSTEM_CFLAGS='-O2 -fno-omit-frame-pointer -mcpu=native'
export TARGET_SYSTEM_MAKEFLAGS='-j8'
export TARGET_SERIAL_DEV='ttyAMA0'
export TARGET_MACHINE='normal'
```

Many of the parameters used in the litbuild blueprints will have static values defined by convention within CBL. For those, we don't care about the host-side parameters that were used, we can just set paths as we wish.

File /tmp/cblwork/sysroot/scaffolding/setenv.sh (continued):

```
export DOCUMENT_DIR=/tmp/cblwork/doc
export LOGFILE_DIR=/tmp/cblwork/logs
export PATCH_DIR=/scaffolding/materials
export SCRIPT_DIR=/tmp/cblwork/scripts
export TARFILE_DIR=/scaffolding/materials
export WORK_SITE=/scaffolding/build
```

9.4.2. Starting Up The Target System

Before we can execute the build process, we need to get the target system to a baseline functional state — kernel filesystems are mounted, the root filesystem is writable, and litbuild is available to generate scripts from blueprints. This could be done inline as part of the `init` script, but I'm writing it to a separate script, `startup-system.sh` instead. When the CBL process is modified, problems

frequently arise, leading to build failures in one place or another, and a technique I often use to resolve those issues is to reboot the target system with the `init` command-line parameter changed from `/scaffolding/sbin/init` to `/scaffolding/bin/bash` — so I wind up at an interactive bash prompt. At that point it's very convenient to be able to source `setenv.sh` and then run `startup-system.sh` to get the target system to a baseline usable state *without* kicking off the full target build again.

The actual `init` script will also start by running `startup-system.sh`, of course.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
/scaffolding/startup-system.sh
```

The commands in `startup-system.sh` are designed to be idempotent: that is to say, you can run them any number of times and they'll ensure that the system is in the expected state when they have completed, but won't make unnecessary changes. This is helpful in cases where the build process does not complete successfully and the target system needs to be restarted.

The first thing to do is remount the root filesystem so we can write to it.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh:

```
#!/scaffolding/bin/bash
mount -o remount,rw /
```

Commands:

```
chmod a+x /tmp/cblwork/sysroot/scaffolding/startup-system.sh
```

Linux systems have several magical filesystems. Two of these, `/sys` and `/proc`, don't actually contain real files and directories, they just contain things that *look* like files and directories, but actually provide insight into the state of the kernel and system — and in many cases also provide the ability to manipulate that state — using a file interface. For example, `/proc/cmdline` shows the command line that was passed to the kernel by the boot loader, and `/proc/filesystems` shows the set of filesystems that are supported by the kernel.

`/dev` is different — it's the canonical location on the system for nodes, aka "special" files. These represent I/O devices and allow those devices to be accessed as though they were files. Linux has a feature called "devtmpfs" that automatically populates a RAM-based filesystem with nodes for all of the I/O devices it knows about.

Additional directories that should have RAM-based filesystems mounted on them are `/run` and `/tmp`. `/run` is a canonical location for files containing volatile system state information — things like the runtime s6 scan directory — and `/tmp` is the conventional location for temporary files that should not persist across reboots. It's convenient to use a `tmpfs` for that purpose. This does limit the amount of data that can be written to `/tmp`: by default, a `tmpfs` filesystem is limited to half the physical RAM on the system. For small target systems, this can be a problematic constraint; if you wind up having any issues as a result, you can replace the `mount` command here with something like `rm -rf /tmp; mkdir -m 1777 /tmp` to ensure that everything from the previous boot (if any) has been cleared up

but without using a (size-limited) `tmpfs` for it.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -d /dev ]
then
  mkdir -m 0755 -p /dev /proc /run /sys /tmp
  mkdir -m 0755 -p /dev/shm
fi
if [ ! -f /proc/mounts ]
then
  mount -n -o nosuid,noexec,nodev -t proc proc /proc
fi
grep -q ' /sys ' /proc/mounts ||
  mount -n -o nosuid,noexec,nodev -t sysfs sys /sys
grep -q ' /dev ' /proc/mounts ||
  mount -n -o mode=0755,nosuid -t devtmpfs dev /dev
grep -q ' /dev/shm ' /proc/mounts ||
  mount -n -o mode=1777,nosuid -t tmpfs none /dev/shm
grep -q ' /run ' /proc/mounts ||
  mount -n -o mode=0755,nosuid -t tmpfs none /run
grep -q ' /tmp ' /proc/mounts ||
  mount -n -o mode=1777,nosuid -t tmpfs none /tmp
```

9.4.3. Setting the System Clock

Some types of computers — basically all common x86 computers, for example — have batteries that maintain the system clock when the power is turned off. Others, like the Raspberry Pi, do not! Build tools rely on file timestamps to determine what build steps need to be performed and which steps can be skipped, so we need to make sure that the system clock is set to something reasonable before we start building things.

We'll make sure the `init` script has a current timestamp now:

Commands:

```
touch /tmp/cblwork/sysroot/scaffolding/sbin/init
```

And when we run the `startup-system.sh` script, we'll set the clock to the time of the init script. This may not be good enough if the build is restarted, and some files have a timestamp later than the `init` script, but it's easy. (A more rigorous solution would be to inspect all the files on the root filesystem, and set the clock to the same timestamp as the latest timestamp found there. But that would take substantially longer, so I'm starting with the easy approach and will modify it further as necessary.)

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
echo "Setting the system clock"
date -s "$(date -r /scaffolding/sbin/init)"
```

9.4.4. Trifling with the entropy pool

There's a little kludge we're going to use to avoid long delays before the target-side build starts, but to explain what we're doing I'm going to have to digress a few steps — you can skip this whole section if you're not interested in some abstruse technical details.

Still with me? Brave soul. Okay, the Linux kernel has a built-in feature that lets you obtain cryptographically-strong random numbers. This is not something that computers are naturally good at, because computers are highly deterministic — there are plenty of pseudo-random number generation (PRNG) algorithms, but they all have to be seeded with some initial value and if you don't have a source for that initial seed value that's *really* random, you wind up with very predictable pseudo-random numbers. If you're doing something involving cryptography, that's no good! When a cryptographic algorithm calls for random bytes — in this context, sometimes this is called "entropy," although I remember hearing at some point that the terms aren't strictly synonymous — the values of those bytes really need to be unpredictable, or else the strength of the algorithm collapses entirely.

Linux makes up for this by collecting some bits of entropy from unpredictable events. The most easily-observed of these events are human inputs, like keystrokes and mouse movements; by measuring the time between keystrokes and taking the least significant few bits from each interval, or measuring the interval between receiving network packets or some other kinds of hardware interrupts or other things like that, Linux can gather a fair amount of really random data. It mixes these random bits into an "entropy pool" it maintains. (You can read all about this in `drivers/char/random.c` in the Linux source tree.)

There are two easy ways that userspace processes can request random data from the entropy pool: they can read it out of the character device `/dev/random`, which only provides as much random data as Linux estimates to be available and blocks after that's exhausted, and `/dev/urandom`, which is basically a PRNG that is periodically re-initialized with a really-random seed value — good enough for most purposes, and it never blocks. That latter source, `/dev/urandom`, is what the `SecureRandom` class in the Ruby standard library uses to initialize *its* PRNG; it's also used by lots of other programs that need random data.

Before Linux 4.16, that was the end of the story: there was `/dev/random`, which you would use if you wanted completely random bytes for things like cryptographic key generation and one-time pads; and `/dev/urandom`, which was perfect for getting more-or-less random data without any delays. In the 4.16 release series, though, this behavior changed slightly in order to address a security concern: now, if any process tries to read from `/dev/urandom` before the kernel's PRNG has been initialized with a really-random seed value, the read blocks until that initialization is complete.

This is still fine in most circumstances! But the CBL target-side build is unusual: it's intended to be completely automated, and it's not on a network because that would make things more complex than they need to be. So there aren't any keystrokes or network packets to measure timings from! That means that initializing the kernel's PRNG can take fifteen minutes or longer after booting the target system. It also turns out that some of the programs used in the target build — for example, the litbuild program used to generate the scripts for that build — wind up reading from one of the random devices, which causes the target-side build to hang — sometimes for quite a while — shortly after it starts.

You can feed entropy to the pool just by writing data to `/dev/urandom` (or, I think, `/dev/random`), but the kernel doesn't trust that any data you write that way is really random, so this doesn't help unblock the build. There's also a system call available to privileged processes (that is, processes that are running as root) that adds random data to the pool and assures the kernel that it really *is* random, so we can avoid unnecessary delays in the build by using a tiny C program to invoke that system call.

Dependencies

Build and Install Entropy Adder (scaffolding phase).

Of course, at this point we don't have any reliable source of random data to use with that program — that's the whole problem! But, luckily, we don't actually need one. We're not doing anything in the CBL build where the lack of cryptographically-strong random data is a problem. So we're just going to lie to the kernel: we'll write some *non*-random data to the entropy pool and assert that it is random.

Specifically, we're going to pretend that the program code for the `addentropy` program is random, even though it's really not at all!

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
echo "Adding fake randomness to the entropy pool"
addentropy < /scaffolding/bin/addentropy
echo "Fake randomness added"
```

9.4.5. Basic system directories

We need to create the rest of the basic sytem directory structure. This doesn't have to happen right now, but this is as good a time as any.

If the build process was interrupted and restarted, the directory structure and symbolic links will already exist, so we can skip this stuff. The same kind of logic applies in many later parts of this script, so there will be additional guard `if` statements around those blocks.

The directory structure for GNU/Linux systems is pretty well standardized — the Linux Foundation maintains the "Filesystem Hierarchy Standard" document,[3] and I generally like the conventions laid out by that document, so the directory structure created here for Little Blue Linux is mostly consistent with those guidelines.

There's one significant exception, though: the FHS mandates having all of the programs and libraries needed to boot the system in top-level directories `/bin`, `/lib`, and `/sbin`. Other programs and libraries that make up the operating system are in similar directories, but under `/usr` — that is, `/usr/bin`, `/usr/lib`, and `/usr/sbin`. A lot of GNU/Linux distributions have stopped doing this, because there are benefits to having all of the read-only files that make up the operating system in one place, and the `/usr` directory structure is a good place for that. That's how we set up Little Blue Linux, as well: the top-level directories *exist*, but they are symbolic links to the actual directories under `/usr`.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -d /bin ]
then
  echo "Creating standard system directories"
  pushd /
  chmod 0775 .
  mkdir -m 0755 -p boot etc home libexec media mnt opt usr var
  mkdir -m 0750 -p root
  pushd usr
  mkdir -m 0755 -p bin include lib libexec local sbin share src
  pushd share
  mkdir -m 0755 -p doc info locale man misc terminfo zoneinfo
  pushd man
  mkdir -m 0755 -p man1 man2 man3 man4 man5 man6 man7 man8
  popd # /usr/share/man
  popd # /usr/share
  pushd local
  mkdir -m 0755 -p bin include lib libexec sbin share src
  pushd share
  mkdir -m 0755 -p doc info locale man misc terminfo zoneinfo
  pushd man
  mkdir -m 0755 -p man1 man2 man3 man4 man5 man6 man7 man8
  popd # /usr/local/share/man
  popd # /usr/local/share
  popd # /usr/local
  popd # /usr
  pushd var
  mkdir -m 0755 -p cache lib local lock log mail opt spool
  mkdir -m 1777 -p tmp
  ln -s /run /var/run
  popd # /var
  popd # /
```

As promised earlier, we need to create symbolic links so that packages that install files in `/bin`, `/lib`, and `/sbin` are able to do that — the files will wind up in the corresponding locations in the `/usr` hierarchy.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
  ln -s usr/bin bin
  ln -s usr/lib lib
  ln -s usr/sbin sbin
else
  echo "Standard system directories are already present"
fi
```

Also, all the multilib directories that the GNU toolchain components sometimes insist on using should be symbolic links to `/usr/lib`. This is still a kludge, but it's the only way to avoid needing to specify some arbitrary set of lib directories in `ld.so.conf`.

If you're building for an architecture that uses *different* multilib directory names, you might need to create additional symbolic links here. If you do that, you'll probably also need to make a corresponding tweak in Create Symbolic Links For Scaffolding Lib Directories; also, look at the specs file when it's being adjusted to see whether the multilib paths are present in the linking specs. If they are, you may need to make changes there as well.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -L /usr/libx32 ]
then
  echo "Creating multilib symbolic links"
  for DIR in / /usr
  do
    pushd $DIR
    for MULTILIBDIR in lib32 lib64 libx32
    do
      ln -sv lib ${MULTILIBDIR}
    done
    popd
  done
else
  echo "Multilib symbolic links are already present"
fi
```

Conventionally, information about filesystems that are currently mounted is available in the file `/etc/mtab`, and some packages therefore expect `/etc/mtab` to contain this information. The historical convention is that the `mount` and `umount` programs, which attach and detach filesystems from the filesystem hierarchy, would also add and remove corresponding lines from the mtab file.

The mtab file isn't actually needed any more, though, because the `/proc` filesystem contains a file that always contains the kernel's view of what filesystems are mounted. We can create a symbolic link to the conventional location, for the use of any packages that look for it there.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -L /etc/mtab ]
then
  echo "Creating mtab symbolic link"
  ln -sv /proc/self/mounts /etc/mtab
else
  echo "mtab symbolic link is already present"
fi
```

The `mount` and `umount` programs do still rely on a configuration file at a standard location for filesystem information: `/etc/fstab`. This file contains information on what filesystems are part of the system, and where they should be mounted. As is generally the case, if you want to know all about `fstab`, you can read the manual page with `man 5 fstab`. On standard Little Blue Linux systems, we the root filesystem is an ext4 filesystem with the label `lblroot`, so we should write an entry for it. Also, the `/tmp` directory should be cleaned out automatically at boot time, and an easy way to do

that is simply to mount a `tmpfs` file system there.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -f /etc/fstab ]
then
  echo "Creating initial fstab"
  echo -n "# <file system> <mount point> <type> " > /etc/fstab
  echo "<options> <dump> <pass>" >> /etc/fstab
  echo "LABEL=lblroot / ext4 errors=remount-ro 0 1" >> /etc/fstab
  echo "tmpfs /tmp tmpfs mode=1777,nosuid,relatime 0 0" >> /etc/fstab
else
  echo "fstab is already present"
fi
```

9.4.6. User and Group database files

Users are defined in the file `/etc/passwd`. This file *can* also contain hashed versions of users' passwords, as well as other user account metadata — hence its name — but usually it does not! This is because `passwd` has to be readable by everyone (it's not really important why this is the case, so I won't get into that here) and shortly after UNIX systems became popular it became clear that it was a bad idea to allow all users to see even the hashed version of passwords.[4]

So in most UNIX operating systems, the `/etc/passwd` file (confusingly) does not contain even the hashed version of passwords; those are found in a file called `/etc/shadow`, which can only be accessed by root. The extraction of passwords into `/etc/shadow`, and maintenance of them there, is done by the shadow package, which we'll set up early in the target-side build.

These files have colon-delimited fields and newline-delimited records, so they're pretty easy to read; `man 5 passwd` describes the file format. The second field is for the hashed password itself. If the field is blank, that account requires no password to log in; if it contains a single "x," that indicates that the password is stored in `/etc/shadow` as described a moment ago.

Users can belong to *groups*. Group definitions are similar to user definitions, but are in the file `/etc/group` — and, similarly to the `passwd` and `shadow` files, the password for groups that require passwords are usually found in `/etc/gshadow`.[5]

Initially we're just going to create the `root` user and group, and a few other system users and groups that are expected to exist by various programs.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -f /etc/passwd ]
then
echo "Creating passwd file"
cat > /etc/passwd <<-EOF
root::0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:6::daemon:/sbin:/bin/false
syslog:x:18:18:syslogd user:/var/log/syslogd:/bin/false
```

```
klog:x:19:19:klogd user:/var/log/klogd:/bin/false
nobody:x:65534:65533:Unprivileged User:/dev/null:/bin/false
EOF
else
echo "passwd file is already present"
fi
if [ ! -f /etc/group ]
then
echo "Creating group file"
cat > /etc/group <<EOF
root:x:0:
bin:x:1:
sys:x:2:
kmem:x:3:
tape:x:4:
tty:x:5:
daemon:x:6:
floppy:x:7:
disk:x:8:
lp:x:9:
dialout:x:10:
audio:x:11:
video:x:12:
utmp:x:13:
usb:x:14:
cdrom:x:15:
adm:x:16:
input:x:17:
syslog:x:18:
klog:x:19:
mail:x:30:
wheel:x:39:
nogroup:x:65533:
EOF
else
echo "group file is already present"
fi
```

9.4.7. Process accounting files

There are some programs that write log data — like process resource usage data — to specific files, but only if those files already exist. Let's create them.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -f /var/log/btmp ]
then
  echo "Creating process accounting files"
  touch /var/log/{btmp,{last,fail}log,wtmp}
  chgrp 13 /var/log/{last,fail}log
```

```
  chmod 0664 /var/log/{last,fail}log
  chmod 0600 /var/log/btmp
else
  echo "Process accounting files are already present"
fi
```

9.4.8. Shell startup files

The root user should have shell startup files just as other users do, and there's no more-convenient place to set those up. Generally for the root user I like avoiding any significant shell startup activities, I only set environment variables, and I like having the same set of environment variables regardless of whether I'm using a login shell (for which the `.bash_profile` script is sourced) or a non-login shell (for which `.bashrc` is sourced), so I link those together.

More typically, `.bash_profile` can be used for any commands that should only be run at login time — like starting an ssh-agent process, for example — and `.bashrc` can be used for commands that should be run any time a new subshell is executed. (It's common for `.bash_profile` to source in `.bashrc` as well, but this is by no means necessary.).

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -f /root/.bashrc ]
then
  echo "Creating root bash startup scripts"
  echo 'export LITBUILDDBDIR=/var/litbuilddb' > /root/.bash_profile
  echo "export PS1='# '" >> /root/.bash_profile
  echo 'export PATH=/usr/bin:/usr/sbin' >> /root/.bash_profile
```

In addition to typical environment variables like `PATH`, we again want root's environment to include all of the configuration parameters that will be used when running litbuild on the final system. Some of these, like PATCH_DIR and TARFILE_DIR, will be useful after the build is complete, so those will be written to `.bash_profile`. Others won't, so we'll write those to a `.cblrc` file that gets sourced in from `.bashrc` — once the build is complete, we can remove the command that sources that file from `.bash_profile`.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
echo 'source /root/.cblrc' >> /root/.bash_profile
```

The code that sets these environment variables digs a little bit deeper into the `bash` bag of tricks than usual; the idea is, for each of the environment variables we want to set, we'll echo both the name of the variable and the result of expanding the name of the variable *as an environment variable* into the `.bash_profile` script — an indirect reference. To do this we need to escape the first `$`, because otherwise `bash` expands `$$` into its own process ID, which is not helpful.

I didn't actually figure out how to do indirect references in bash; I did a something search and found everything I needed in chapter 28 of the "Advanced Bash-Scripting Guide," which is part of the Linux Documentation Project.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
for var in KERNEL_TARGET PATCH_DIR TARFILE_DIR
do
  echo "export $var='$(eval echo \$$var)'" >> /root/.bash_profile
done
for var in BOOT_DEVICE BOOTLOADER DOMAIN_NAME HOST_NAME \
    LOGIN_FULL_NAME LOGIN TARGET_GCC_CONFIG TARGET_SYSTEM_CFLAGS \
    TARGET_SYSTEM_MAKEFLAGS DOCUMENT_DIR LOGFILE_DIR SCRIPT_DIR \
    WORK_SITE
do
  echo "export $var='$(eval echo \$$var)'" >> /root/.cblrc
done
```

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
  chmod 700 /root/.bash_profile
  ln -s .bash_profile /root/.bashrc
else
  echo "Root bash startup scripts are already present"
fi
```

9.4.9. Virtual memory for the initial target system

The Linux kernel can write pages of RAM to storage devices like disk partitions or files in the filesystem, which frees up those pages to be allocated to other programs; this is called "swapping", and the blocks of storage allocated to this purpose are referred to as "virtual memory" because they work like (very slow) RAM, or "swap space" because it's space allocated to swapping. The terms are pretty much interchangeable.

It's always a good idea to have some swap space available. That way, allocated memory that hasn't been used in a long time can be written out to the swap area. If any process needs to access those pages, the kernel can re-read them from swap, so the only drawback is that access to those pages of memory takes longer than if they were already in RAM.

A configuration parameter, `TARGET_SWAP_DEVICE`, can be used to specify a block device to be used as virtual memory. If it exists and isn't already in use, the following code will set it up. If the device exists but is something other than swap space, it will be ignored — just in case the parameter has an incorrect value and refers to an important filesystem or something.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ -b /dev/vdb ]
then
  echo "Checking for swap device /dev/vdb..."
  blkid /dev/vdb
  if [ $? -ne 0 ]
  then
    echo "Initializing swap device /dev/vdb"
```

```
    mkswap /dev/vdb
  fi
  if blkid /dev/vdb | grep -q swap
  then
    echo "Activating swap device /dev/vdb"
    swapon /dev/vdb
  fi
fi
```

CBL systems use s6 and s6-rc to manage the system state. We can set up the very first parts of that here — that is, the source directory that will contain service definitions, and the run-level bundle directories inside it. As other parts of the system are built, directories for other services will be created and, in many cases, added to the `rl-default` bundle.

We also set up an `network-services` bundle that will be part of the `rl-default` bundle if networking and network services should be part of the standard system run state.

This is all explained in much more detail in later sections: s6-rc, Configure the system initialization framework, and Construct the s6-rc service database.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
if [ ! -d /etc/s6-rc/source ]
then
  echo "Setting up initial s6-rc structures"
  mkdir -m 0755 -p /etc/s6-rc
  mkdir -m 0755 -p /etc/s6-rc/source
  mkdir -m 0755 -p /etc/s6-rc/source/rl-default
  echo "bundle" > /etc/s6-rc/source/rl-default/type
  mkdir -p /etc/s6-rc/source/rl-default/contents.d
  mkdir -m 0755 -p /etc/s6-rc/source/network-services
  echo "bundle" > /etc/s6-rc/source/network-services/type
  mkdir -p /etc/s6-rc/source/network-services/contents.d
fi
```

9.4.10. Installing litbuild

The litbuild program needs to be installed so that we can use it to create scripts to build the rest of the system. We could simply install the rubygem-packaged version of litbuild, but it's more consistent with the rest of CBL to install it from a source tarfile.

Of course, if the litbuild gem is already installed, we don't need to build and install it.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
echo "Looking for an installed litbuild..."
type lb
if [ $? -eq 1 ]
then
  echo "Building and installing the litbuild gem"
```

```
  pushd /tmp
  tar -x -f /scaffolding/materials/litbuild*tar
  cd litbuild*
  gem build litbuild.gemspec
  gem install -l litbuild*gem
  popd
  rm -rf /tmp/litbuild*
else
  echo "Litbuild gem is already installed"
fi
```

To use the skip-upon-restart feature of litbuild throughout the target-side build, we can set up a litbuild database directory.

File /tmp/cblwork/sysroot/scaffolding/startup-system.sh (continued):

```
export LITBUILDDBDIR=/scaffolding/litbuilddb
if [ ! -d $LITBUILDDBDIR ]
then
  echo "Creating litbuild database directory"
  mkdir $LITBUILDDBDIR
fi
echo "Target system is now started and functional"
```

9.4.11. The Target-Side Build

Now let's proceed with the target-side build *per se*.

By default, `bash` keeps track of the location for programs it runs, which lets it skip looking through the `PATH` when it needs to find those programs again. As we construct the final system programs, we want those to be used in preference to the scaffolding versions, so we disable this caching behavior.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
set +h
```

If anything goes wrong during the remainder of the build, we can bail out to an interactive shell. To do this, we'll set the `-e` bash option (which causes any failing command to terminate the script) but set traps so that script termination — normal or unexpected — will cause the process to execute an interactive bash shell.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
trap 'exec /scaffolding/bin/bash' EXIT
trap 'exec /scaffolding/bin/bash' ERR
set -e
```

Now we can start the target-side build itself! This consists of running litbuild to produce scripts,

then executing those scripts.

The first litbuild target builds the remainder of the scaffolding components — all the stuff we need for the real target-side build but which is difficult or impossible to cross-compile on the host system — and sets up the package-users framework.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
cd /scaffolding/cbl
echo "Beginning target-side-initial build"
LOGFILE_DIR=/root/cbl-logs/0-target-side-initial lb target-side-initial
/tmp/cblwork/scripts/00-target-side-initial.sh
rm -rf /tmp/cblwork
```

Second, we build the final system components — these will automatically be generated as package-user-style build scripts, because the package-users framework is installed now. This is a also a good time to switch to a final system litbuild database directory, since this is the point where we'll start to build out the final system!

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
export LITBUILDDBDIR=/var/litbuilddb
if [ ! -d $LITBUILDDBDIR ]
then
  echo "Creating litbuild database directory"
  mkdir $LITBUILDDBDIR
fi
echo "Beginning target-side-final build"
LOGFILE_DIR=/root/cbl-logs/1-target-side-final lb target-side-final
/tmp/cblwork/scripts/00-target-side-final.sh
rm -rf /tmp/cblwork
```

At this point, we have the `bash` program at the canonical system location, `/usr/bin/bash` (which is the same as `/bin/bash`, since `/bin` is a symbolic link to `/usr/bin`). That means we can change the traps we set earlier to exec into *that* shell if the init script terminates.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
trap 'exec /usr/bin/bash' EXIT
trap 'exec /usr/bin/bash' ERR
```

And, finally, after building all of the programs and libraries needed on the system, we can finish up the system: tidy things up, remove any remaining references to the `/scaffolding` directory, configure the init system, and perhaps install the boot loader.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
echo "Beginning complete-the-system build"
LOGFILE_DIR=/root/cbl-logs/2-complete-the-system lb complete-the-system
```

```
/tmp/cblwork/scripts/00-complete-the-system.sh
rm -rf /tmp/cblwork
```

There are a couple more things we can do to tidy up before we declare the build complete.

Before shutting down the system, we should remount the root filesystem read-only. This can be tricky because there may be some processes still lingering from the build process — I've seen processes still running as `glibc`, left over from the final system `glibc` build, for example. There might be others as well.

Unfortunately, these processes are holding on to open file descriptors on the root filesystem, so we have to terminate them in order to remount that filesystem read-only. And unfortunately, that's *also* tricky: killing those processes can, for some reason I find completely baffling, cause the init script to hang or crash.

If this happens, you won't get a `COMPLETE SUCCESS` but it's still safe to power-cycle the target system: we're using a journaling filesystem, so we won't need to do an extensive filesystem check on reboot or anything. It's always a good idea to force all pending I/O operations to complete before proceeding, though.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
echo "Synchronizing disks"
sync
echo "Finding lingering build processes..."
cd /proc
while ls -l | grep -v total | grep -q -v root
do
    pid=$(ls -l | grep -v total | grep -v root | \
        head -n 1 | awk '{print $9}')
    owner=$(ls -l | grep "$pid\$" | awk '{print $3}')
    echo -n "Found PID $pid owned by $owner, terminating..."
    kill -9 $pid && echo "killed" || echo "already gone"
    sleep 1
done
```

If that worked, we should be able to make the root filesystem read-only and declare victory. But sometimes — honestly I have no idea what the deal is — I *still* find that Linux thinks the root filesystem has files open for writing. So we'll put some explanations in the console messaging, just in case.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
echo "Synchronizing disks again"
sync
echo "Trying to remount the root filesystem read-only."
echo "(If this fails, it does not indicate a dire problem. The"
echo "filesystem has a journal, and we have just sync'ed it.)"
echo ""
set +e
```

```
mount -o remount,ro /
sync
```

The target system build is complete. Nothing remains but to turn off the computer!

Under normal circumstances, the s6-based init system handles shutdown and poweroff. The way this works is, the init framework starts a long-running process running `s6-linux-init-shutdownd`, which listens on a fifo for instructions from other programs like `s6-linux-init-shutdown` and performs clean shutdown operations when told to do so. Of course, we're just running a bash script as `init`, so we don't have any of this stuff running.

Looking at what `s6-linux-init-shutdown` does, though, it turns out that the very last thing it does is run `s6-linux-init-hpr` with an `-f` (for "force") option, which does a final filesystem sync operation and then uses the `reboot` system call to perform the actual shutdown (or halt, or reboot) operation. We can just run that program directly to power off the system.

File /tmp/cblwork/sysroot/scaffolding/sbin/init (continued):

```
echo "COMPLETE SUCCESS"
sync
echo ""
echo "Shutting down the system now."
sleep 5
s6-linux-init-hpr -f -p -W
```

If anything goes wrong with the `s6-linux-init-hpr` command, the trap we set earlier will hopefully cause the script to drop into an interactive shell when it terminates.

9.4.12. Complete text of files

9.4.12.1. /tmp/cblwork/sysroot/scaffolding/sbin/init

```
#!/scaffolding/bin/bash
source /scaffolding/setenv.sh
/scaffolding/startup-system.sh
set +h
trap 'exec /scaffolding/bin/bash' EXIT
trap 'exec /scaffolding/bin/bash' ERR
set -e
cd /scaffolding/cbl
echo "Beginning target-side-initial build"
LOGFILE_DIR=/root/cbl-logs/0-target-side-initial lb target-side-initial
/tmp/cblwork/scripts/00-target-side-initial.sh
rm -rf /tmp/cblwork
export LITBUILDDBDIR=/var/litbuilddb
if [ ! -d $LITBUILDDBDIR ]
then
  echo "Creating litbuild database directory"
  mkdir $LITBUILDDBDIR
```

```
fi
echo "Beginning target-side-final build"
LOGFILE_DIR=/root/cbl-logs/1-target-side-final lb target-side-final
/tmp/cblwork/scripts/00-target-side-final.sh
rm -rf /tmp/cblwork
trap 'exec /usr/bin/bash' EXIT
trap 'exec /usr/bin/bash' ERR
echo "Beginning complete-the-system build"
LOGFILE_DIR=/root/cbl-logs/2-complete-the-system lb complete-the-system
/tmp/cblwork/scripts/00-complete-the-system.sh
rm -rf /tmp/cblwork
echo "Synchronizing disks"
sync
echo "Finding lingering build processes..."
cd /proc
while ls -l | grep -v total | grep -q -v root
do
    pid=$(ls -l | grep -v total | grep -v root | \
        head -n 1 | awk '{print $9}')
    owner=$(ls -l | grep "$pid\$" | awk '{print $3}')
    echo -n "Found PID $pid owned by $owner, terminating..."
    kill -9 $pid && echo "killed" || echo "already gone"
    sleep 1
done
echo "Synchronizing disks again"
sync
echo "Trying to remount the root filesystem read-only."
echo "(If this fails, it does not indicate a dire problem. The"
echo "filesystem has a journal, and we have just sync'ed it.)"
echo ""
set +e
mount -o remount,ro /
sync
echo "COMPLETE SUCCESS"
sync
echo ""
echo "Shutting down the system now."
sleep 5
s6-linux-init-hpr -f -p -W
```

9.4.12.2. /tmp/cblwork/sysroot/scaffolding/setenv.sh

```
export PATH=/scaffolding/bin:/scaffolding/sbin
export LD_LIBRARY_PATH=/scaffolding/lib
export LC_ALL=POSIX
export PATH=/usr/bin:/usr/sbin:$PATH
export LD_LIBRARY_PATH=/usr/lib:$LD_LIBRARY_PATH
export MAKEFLAGS="-j8"
export HOME=/root
export BOOT_DEVICE=''
```

```
export BOOTLOADER='manual'
export HOST_NAME='cblviton'
export DOMAIN_NAME='freesa.org'
export KERNEL_TARGET='Image.gz'
export LOGIN_FULL_NAME='A Little Blue User'
export LOGIN='lbl'
export TARGET_GCC_CONFIG='--enable-fix-cortex-a53-835769 --enable-fix-cortex-a53
-843419 --with-cpu=cortex-a72.cortex-a53 --enable-standard-branch-protection'
export TARGET_SYSTEM_CFLAGS='-O2 -fno-omit-frame-pointer -mcpu=native'
export TARGET_SYSTEM_MAKEFLAGS='-j8'
export TARGET_SERIAL_DEV='ttyAMA0'
export TARGET_MACHINE='normal'
export DOCUMENT_DIR=/tmp/cblwork/doc
export LOGFILE_DIR=/tmp/cblwork/logs
export PATCH_DIR=/scaffolding/materials
export SCRIPT_DIR=/tmp/cblwork/scripts
export TARFILE_DIR=/scaffolding/materials
export WORK_SITE=/scaffolding/build
```

9.4.12.3. /tmp/cblwork/sysroot/scaffolding/startup-system.sh

```
#!/scaffolding/bin/bash
mount -o remount,rw /
if [ ! -d /dev ]
then
  mkdir -m 0755 -p /dev /proc /run /sys /tmp
  mkdir -m 0755 -p /dev/shm
fi
if [ ! -f /proc/mounts ]
then
  mount -n -o nosuid,noexec,nodev -t proc proc /proc
fi
grep -q ' /sys ' /proc/mounts ||
  mount -n -o nosuid,noexec,nodev -t sysfs sys /sys
grep -q ' /dev ' /proc/mounts ||
  mount -n -o mode=0755,nosuid -t devtmpfs dev /dev
grep -q ' /dev/shm ' /proc/mounts ||
  mount -n -o mode=1777,nosuid -t tmpfs none /dev/shm
grep -q ' /run ' /proc/mounts ||
  mount -n -o mode=0755,nosuid -t tmpfs none /run
grep -q ' /tmp ' /proc/mounts ||
  mount -n -o mode=1777,nosuid -t tmpfs none /tmp
echo "Setting the system clock"
date -s "$(date -r /scaffolding/sbin/init)"
echo "Adding fake randomness to the entropy pool"
addentropy < /scaffolding/bin/addentropy
echo "Fake randomness added"
if [ ! -d /bin ]
then
  echo "Creating standard system directories"
```

```
  pushd /
  chmod 0775 .
  mkdir -m 0755 -p boot etc home libexec media mnt opt usr var
  mkdir -m 0750 -p root
  pushd usr
  mkdir -m 0755 -p bin include lib libexec local sbin share src
  pushd share
  mkdir -m 0755 -p doc info locale man misc terminfo zoneinfo
  pushd man
  mkdir -m 0755 -p man1 man2 man3 man4 man5 man6 man7 man8
  popd # /usr/share/man
  popd # /usr/share
  pushd local
  mkdir -m 0755 -p bin include lib libexec sbin share src
  pushd share
  mkdir -m 0755 -p doc info locale man misc terminfo zoneinfo
  pushd man
  mkdir -m 0755 -p man1 man2 man3 man4 man5 man6 man7 man8
  popd # /usr/local/share/man
  popd # /usr/local/share
  popd # /usr/local
  popd # /usr
  pushd var
  mkdir -m 0755 -p cache lib local lock log mail opt spool
  mkdir -m 1777 -p tmp
  ln -s /run /var/run
  popd # /var
  popd # /
  ln -s usr/bin bin
  ln -s usr/lib lib
  ln -s usr/sbin sbin
else
  echo "Standard system directories are already present"
fi
if [ ! -L /usr/libx32 ]
then
  echo "Creating multilib symbolic links"
  for DIR in / /usr
  do
    pushd $DIR
    for MULTILIBDIR in lib32 lib64 libx32
    do
      ln -sv lib ${MULTILIBDIR}
    done
    popd
  done
else
  echo "Multilib symbolic links are already present"
fi
if [ ! -L /etc/mtab ]
then
```

```
  echo "Creating mtab symbolic link"
  ln -sv /proc/self/mounts /etc/mtab
else
  echo "mtab symbolic link is already present"
fi
if [ ! -f /etc/fstab ]
then
  echo "Creating initial fstab"
  echo -n "# <file system> <mount point> <type> " > /etc/fstab
  echo "<options> <dump> <pass>" >> /etc/fstab
  echo "LABEL=lblroot / ext4 errors=remount-ro 0 1" >> /etc/fstab
  echo "tmpfs /tmp tmpfs mode=1777,nosuid,relatime 0 0" >> /etc/fstab
else
  echo "fstab is already present"
fi
if [ ! -f /etc/passwd ]
then
echo "Creating passwd file"
cat > /etc/passwd <<-EOF
root::0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:6::daemon:/sbin:/bin/false
syslog:x:18:18:syslogd user:/var/log/syslogd:/bin/false
klog:x:19:19:klogd user:/var/log/klogd:/bin/false
nobody:x:65534:65533:Unprivileged User:/dev/null:/bin/false
EOF
else
echo "passwd file is already present"
fi
if [ ! -f /etc/group ]
then
echo "Creating group file"
cat > /etc/group <<EOF
root:x:0:
bin:x:1:
sys:x:2:
kmem:x:3:
tape:x:4:
tty:x:5:
daemon:x:6:
floppy:x:7:
disk:x:8:
lp:x:9:
dialout:x:10:
audio:x:11:
video:x:12:
utmp:x:13:
usb:x:14:
cdrom:x:15:
adm:x:16:
input:x:17:
```

```
syslog:x:18:
klog:x:19:
mail:x:30:
wheel:x:39:
nogroup:x:65533:
EOF
else
echo "group file is already present"
fi
if [ ! -f /var/log/btmp ]
then
  echo "Creating process accounting files"
  touch /var/log/{btmp,{last,fail}log,wtmp}
  chgrp 13 /var/log/{last,fail}log
  chmod 0664 /var/log/{last,fail}log
  chmod 0600 /var/log/btmp
else
  echo "Process accounting files are already present"
fi
if [ ! -f /root/.bashrc ]
then
  echo "Creating root bash startup scripts"
  echo 'export LITBUILDDBDIR=/var/litbuilddb' > /root/.bash_profile
  echo "export PS1='# '" >> /root/.bash_profile
  echo 'export PATH=/usr/bin:/usr/sbin' >> /root/.bash_profile
  echo 'source /root/.cblrc' >> /root/.bash_profile
  for var in KERNEL_TARGET PATCH_DIR TARFILE_DIR
  do
    echo "export $var='$(eval echo \$$var)'" >> /root/.bash_profile
  done
  for var in BOOT_DEVICE BOOTLOADER DOMAIN_NAME HOST_NAME \
      LOGIN_FULL_NAME LOGIN TARGET_GCC_CONFIG TARGET_SYSTEM_CFLAGS \
      TARGET_SYSTEM_MAKEFLAGS DOCUMENT_DIR LOGFILE_DIR SCRIPT_DIR \
      WORK_SITE
  do
    echo "export $var='$(eval echo \$$var)'" >> /root/.cblrc
  done
  chmod 700 /root/.bash_profile
  ln -s .bash_profile /root/.bashrc
else
  echo "Root bash startup scripts are already present"
fi
if [ -b /dev/vdb ]
then
  echo "Checking for swap device /dev/vdb..."
  blkid /dev/vdb
  if [ $? -ne 0 ]
  then
    echo "Initializing swap device /dev/vdb"
    mkswap /dev/vdb
  fi
```

```
  if blkid /dev/vdb | grep -q swap
  then
    echo "Activating swap device /dev/vdb"
    swapon /dev/vdb
  fi
fi
if [ ! -d /etc/s6-rc/source ]
then
  echo "Setting up initial s6-rc structures"
  mkdir -m 0755 -p /etc/s6-rc
  mkdir -m 0755 -p /etc/s6-rc/source
  mkdir -m 0755 -p /etc/s6-rc/source/rl-default
  echo "bundle" > /etc/s6-rc/source/rl-default/type
  mkdir -p /etc/s6-rc/source/rl-default/contents.d
  mkdir -m 0755 -p /etc/s6-rc/source/network-services
  echo "bundle" > /etc/s6-rc/source/network-services/type
  mkdir -p /etc/s6-rc/source/network-services/contents.d
fi
echo "Looking for an installed litbuild..."
type lb
if [ $? -eq 1 ]
then
  echo "Building and installing the litbuild gem"
  pushd /tmp
  tar -x -f /scaffolding/materials/litbuild*tar
  cd litbuild*
  gem build litbuild.gemspec
  gem install -l litbuild*gem
  popd
  rm -rf /tmp/litbuild*
else
  echo "Litbuild gem is already installed"
fi
export LITBUILDDBDIR=/scaffolding/litbuilddb
if [ ! -d $LITBUILDDBDIR ]
then
  echo "Creating litbuild database directory"
  mkdir $LITBUILDDBDIR
fi
echo "Target system is now started and functional"
```

Before we can start the target-system part of the process, we need to convey the scaffolding to the target system and launch the target-side build. I call that process the "bridge," because it takes us from the host-system side of CBL to the target-system side.

The exact steps you follow to bridge from the host to the target depend on whether the host, or target, or both, are physical or emulated. The different possibilities are described in different blueprints, which can be selected using the `TARGET_BRIDGE` configuration parameter. The default is the QEMU bridge, but the CBL repository has blueprints for Raspberry Pi target computers, a manual option that you have to figure out by yourself, and possibly other alternatives.

[1] Isn't all this jargon fun?

[2] Zombie processes don't consume any CPU or memory resources, just brains, so it's not a catastrophe to have zombie processes around, but the cluttered process table can eventually cause problems.

[3] at https://refspecs.linuxfoundation.org/fhs.shtml

[4] This is because people often used normal English words as passwords, and it's really easy to run every word in the dictionary through the password hashing algorithm; then if you find one of those hashes in the `passwd` file, you know what the corresponding password is. This is much less of a concern than it used to be, since passwords are now salted as well as being hashed, but that innovation was introduced after the hashed passwords had already been moved into `/etc/shadow`, and there was no reason to move them back to `/etc/passwd` at that point.

[5] Usually, groups don't have passwords, so this is seldom an important detail.

Chapter 10. QEMU Host-To-Target Bridge

When the target system is a virtual machine emulated by QEMU, the `sysroot` directory needs to be copied to a disk image file.

10.1. Create target disk images

QEMU usually operates on disk image files — within the emulated environment, these appear to be block special devices just like real storage hardware like hard disks and SSDs.

The native QEMU image file format is QCOW2. The "QCOW" part is "QEMU Copy-On-Write", which just means that no matter how large you define the image file as being, it starts out small and only grows as data is allocated within it. (The "2" just indicates that it's the second copy-on-write image format that the QEMU developers developed.)

The only way to write to a QEMU image file is to use some program that understands the image file format. Luckily, there is such a program in the QEMU package distribution: `qemu-nbd`, which serves disk image files as block devices using the Network Block Device (NBD) protocol. Then they can be accessed from the host system as though they were normal disk partitions!

CAUTION

Setting up the target images using `qemu-nbd` requires several commands to be run as root, typically using the `sudo` program. This has a couple of consequences. First, all programs run via `sudo` without absolute paths being specified must exist in a standard set of system paths (`bin` and `sbin` directories under `/`, `/usr`, or `/usr/local`). Second, `sudo` will sometimes ask you to enter your password before it will run the requested program, which is problematic when running the litbuild-generated scripts without any interactive user context. You can avoid that by taking the `lb-sudoers` file produced in the script directory and adding its contents to the `/etc/sudoers` file — or, typically, just move it to the `/etc/sudoers.d` directory (and then make sure it's owned by `root` and has a restrictive mode like `0400`).

Obviously, you need to have a kernel that includes the NBD driver to use `qemu-nbd` at all. If yours does not, you'll need to either rebuild it with NBD support (built-in or as a module), or find some other approach for setting up the target root filesystem.

If the NBD driver is built as a module, it will have to be loaded before the `/dev/nbd*` devices will be available.

Commands:

```
if [ ! -b /dev/nbd0 ]; then sudo modprobe nbd; fi
```

You might want to do multiple CBL builds simultaneously for multiple target machine types on a single host system. If more than one build process gets to this part of the build at the same time, it's important that each one use a different NBD device. You can see whether anything is attached to an NBD device file by looking in the `/sys` filesystem: `cat /sys/block/nbdN/size` will show a size of 0 if

no NBD device is attached to the NBD device file `/dev/nbdN`, or report the acutal size (in 512-byte blocks) of the attached NBD device if there is one. Finding the first unused NBD device is complicated enough that it seems worthwhile to use a script to do it.

File /tmp/cblwork/build/find_available_nbd:

```
export devnum=0
while [ "0" != $(cat /sys/block/nbd${devnum}/size) ]
do
  devnum=$((devnum + 1))
done
echo "Found unused NBD device: /dev/nbd${devnum}"
ln -s /dev/nbd${devnum} /tmp/cblwork/build/nbddev
```

(There's still a race condition if more than one concurrent CBL build runs that script simultaneously, but it ought to work well enough to avoid problems most of the time. This seems really unlikely to come up often enough to put in some kind of mutex.)

Now that we have that script available, we can run it to find a free NBD device and create a symbolic link to it.

Commands:

```
bash /tmp/cblwork/build/find_available_nbd
```

For CBL, we set up two separate target images: the first will be the root filesystem, and initially just has the contents of the sysroot directory (which, in the case of CBL, consists of the single directory `/scaffolding`).

Commands:

```
mkdir -p /tmp/cblwork
/home/random/cbltools/bin/qemu-img create -f qcow2 \
  /tmp/cblwork/cbl.qcow2 50G
```

Running `qemu-nbd` requires sudo, and can be fragile — some of the shared libraries needed by `qemu-nbd` may be in the `/home/random/cbltools/lib` directory. This is specified in the `LD_LIBRARY_PATH` for the host-side CBL build, but `LD_LIBRARY_PATH` generally does not work with sudo because it presents a security issue.

To work around this conundrum, we write the `qemu-nbd` commands we need to run into shell scripts, and then run those shell scripts under sudo. This is a distasteful workaround, since running shell scripts under sudo *also* presents a security issue — anyone who can write to those shell scripts can effectively run any command with superuser privilege — but it's the most straightforward approach I can think of here.

File /tmp/cblwork/build/qemu_nbd_connect:

```
#!/bin/bash
```

```
export PATH="/home/random/cbltools/bin:$PATH"
export LD_LIBRARY_PATH="/home/random/cbltools/lib:$LD_LIBRARY_PATH"
qemu-nbd -c /tmp/cblwork/build/nbddev /tmp/cblwork/cbl.qcow2
```

Commands:

```
chmod 775 /tmp/cblwork/build/qemu_nbd_connect
sudo /tmp/cblwork/build/qemu_nbd_connect
```

QEMU sometimes crashes during the CBL system build (this was first observed when building the final system glibc on a mipsel virtual machine; QEMU died with a segmentation fault). It's a good idea to use a journaling filesystem so that the filesystem might still be intact after restarting.

Whatever filesystem you decide to use for your target disk image, the scaffolding kernel needs to include support for it built-in, not as a module! Otherwise, the kernel won't be able to mount the root filesystem. Similarly, you need to compile in support for whatever type of partition table you create on the disk image — here, that's a standard PC BIOS partition table. You can use `fdisk` to create the partition table, but it's designed to be interactive; a more convenient program is `sfdisk`, which is explicitly designed to be used from scripts. All we do here is create a single partition that covers the entire volume.

You might wonder why we're bothering to create a partition table on the disk, when we're always using the entire volume in a single partition. That's just in case the final system uses a boot loader like GRUB, which expects to be able to find a chunk of otherwise- unused space on the volume to store one of the boot loader stages. You can read more about this all in [setup-bootloader-grub].

Commands:

```
echo 'start=2048, type=83' | sudo sfdisk /tmp/cblwork/build/nbddev
```

The `nbddev` symbolic link is a reference to the full volume device; further manipulations need to be done using the device node for the partition we just created. That's not hard to do: just add `p1` to the full-volume device node, which we can find using `readlink`. We can create another symbolic link pointing to that device file, for convenience.

Commands:

```
partition="$(readlink -f /tmp/cblwork/build/nbddev)p1"
ln -s $partition /tmp/cblwork/build/nbdpart
sudo mke2fs -L lblroot -t ext4 -O ext_attr /tmp/cblwork/build/nbdpart
mkdir -p /tmp/cblwork/build/mnt
sudo mount /tmp/cblwork/build/nbdpart /tmp/cblwork/build/mnt
sudo chmod 777 /tmp/cblwork/build/mnt
```

Now that the root filesystem image is ready, copy the sysroot files to it!

Commands:

```
cp -a /tmp/cblwork/sysroot/* /tmp/cblwork/build/mnt
```

Since the root filesystem is now all prepared, unmount it and remove the NBD association. I've noticed that sometimes Linux doesn't write all the buffered changes to disk as quickly as it can, so the umount operation takes a long time; you can ask the kernel to synchronize disks as quickly as possible by using the `sync` command first.

As with connecting the image to an NBD device, we need a script to run under sudo to *disconnect* the image.

File /tmp/cblwork/build/qemu_nbd_disconnect:

```
#!/bin/bash
export PATH="/home/random/cbltools/bin:$PATH"
export LD_LIBRARY_PATH="/home/random/cbltools/lib:$LD_LIBRARY_PATH"
qemu-nbd -d /tmp/cblwork/build/nbddev
```

Commands:

```
sync
sudo umount /tmp/cblwork/build/mnt
chmod 775 /tmp/cblwork/build/qemu_nbd_disconnect
sudo /tmp/cblwork/build/qemu_nbd_disconnect
```

The second disk image used by CBL is for swap space. We could just set up a swap file on the root disk image, but it's tidier to use a separate disk for swap — that way, once the CBL build process is complete, we can simply discard the swap disk if we don't need it any more!

Commands:

```
/home/random/cbltools/bin/qemu-img create -f qcow2 \
  /tmp/cblwork/swap.qcow2 50G
```

10.1.1. Complete text of files

10.1.1.1. /tmp/cblwork/build/find_available_nbd

```
export devnum=0
while [ "0" != $(cat /sys/block/nbd${devnum}/size) ]
do
  devnum=$((devnum + 1))
done
echo "Found unused NBD device: /dev/nbd${devnum}"
ln -s /dev/nbd${devnum} /tmp/cblwork/build/nbddev
```

10.1.1.2. /tmp/cblwork/build/qemu_nbd_connect

```
#!/bin/bash
export PATH="/home/random/cbltools/bin:$PATH"
export LD_LIBRARY_PATH="/home/random/cbltools/lib:$LD_LIBRARY_PATH"
qemu-nbd -c /tmp/cblwork/build/nbddev /tmp/cblwork/cbl.qcow2
```

10.1.1.3. /tmp/cblwork/build/qemu_nbd_disconnect

```
#!/bin/bash
export PATH="/home/random/cbltools/bin:$PATH"
export LD_LIBRARY_PATH="/home/random/cbltools/lib:$LD_LIBRARY_PATH"
qemu-nbd -d /tmp/cblwork/build/nbddev
```

Once we have the disk image files created, we can launch the target system. This will automatically execute the target side of the CBL process; after it is complete, we copy some files to and from the final system image.

10.2. Boot the target device as a virtual machine

The QEMU distribution doesn't include BIOS files for all machine architectures — in particular, it does not include any MIPS BIOS files, which means that when running MIPS virtual machines (which is one of the architectures commonly used in CBL) we can't make the boot process look like it would on an actual computer with a boot loader that loads and executes the kernel.

Luckily, we don't have to do that: QEMU is able to act as a boot loader itself. You just have to give it a `-kernel` command line argument. (You can also give it an initrd image, if you want, but we don't need to do that.)

Normally, QEMU opens a window that can be used by the guest to display graphics using an emulated VGA video controller. If you provide the QEMU option `-nographic`, it will skip that and map stdin and stdout from the QEMU process to a serial device within the guest virtual machine. Since the Linux kernel can be told to use a serial interface as its console with a command-line argument, that's a great way to run QEMU for our purposes.

When running QEMU in this mode, `ctrl-a` can be used to send instructions to the emulator; use `ctrl-a h` to print a brief help message showing what options are available. That's not useful when running QEMU non-interactively, as the CBL blueprint-generated scripts do, but you can also simply run the `qemu` command from this script interactively to play around with the emulated device however you like. When you're doing that, it's helpful to be able to (for example) terminate the emulator without sending it a kill signal from some other window, and that's what the `ctrl-a` commands are for.

A handy trick when running QEMU interactively, before the target system is built, is to change the `init` command line argument to `/scaffolding/bin/bash` so you get a shell prompt rather than starting the CBL target-side build process. If you do this, you'll want to start by sourcing the `/scaffolding/setenv.sh` script — otherwise there won't be any `PATH` or anything to make it

convenient to run commands.

Native QEMU Builds

Suppose you have more than one computer and they have different CPU architectures — say, an ARM Chromebook and an AMD64 (aka `x86_64`) laptop cmoputer. The fastest way to perform the Cross-Built Linux process is to run everything natively, since emulated CPUs are much, much slower than real CPUs. So you might want to run the host-side build on the Chromebook, and the target-side build on the AMD64 laptop.

This is not particularly difficult — you need to use the `manual` version of the `TARGET_BRIDGE`, or write your own blueprint if you wish the whole build to be automated — but still means that the laptop computer won't be usable for anything else while the CBL build is running.

QEMU can help with that, because it can operate as a virtual machine hypervisor as well as a system emulator. If the target-side build is running in a virtual machine, you can still use the computer for whatever other work you wish to do.

If you want to do this, you'll probably want to enable KVM acceleration, if the system supports it; this uses the kernel-based virtual machine feature of Linux to run virtual machines at almost the same speed as the native hardware. It's not clear, from the QEMU `-help` output and documentation, how best to do that! There's an `-enable-kvm` option in the "debug/expert options" section, but you can also just add an `accel=kvm` parameter to the `-machine` option, and there's also an `-accel` option that has `kvm` as one of the alternatives you can select. Do these do different things? Are they all the same? I have no idea. The best way to discover the answer is probably to read the source code, and I have not had the energy to do that so far. What I do is set this in the `TARGET_QEMU_MACHINE` parameter, when the target is an AMD64 computer that supports KVM; I set this to `pc-i440fx-2.8,accel=kvm,usb=off`, and that seems to work well.

Regardless of how you enable KVM acceleration, you'll need read and write access to the `/dev/kvm` character special device, unless you're running QEMU as root. On most systems I use, the `udev` facility sets up `/dev/kvm` to be in a group called `kvm` and with mode `0660`, so you can set this up by adding the user running QEMU to the `kvm` group.

You may also want to specify `-cpu host`, which will tell QEMU not to do any CPU emulation at all, and simply virtualize CPU instuctions from the guest to the host system.

The QEMU process will emulate a full computer system for the entire target-side build — building the rest of the scaffolding, toolchain, kernel, and other software components that will comprise the final system. This is a lot of work, and QEMU sometimes has stability problems, especially when emulating a foreign machine architecture, so you may find that the target build fails in one way or another. The failure modes I've observed personally are:

- The QEMU process crashes with a segmentation fault.
- There is a kernel panic at some point during the build.
- No visible issues have occurred, but the build hangs forever (this has been observed while

running the glibc test suite, and when installing the final system QEMU package).

In most cases, simply restarting the qemu process allows the build to proceed fairly gracefully from where it left off, so it makes sense to detect those situations and try restarting the target-side build a few times if any of them happen.

That's a complicated-enough job that embedding all the necessary logic into this blueprint is ugly and confusing, so instead I've collaborated with Eric Herman to implement it in a small program, yoyo. It simply runs another program — in this case the QEMU process — restarts it if it crashes, and kills and restarts it if it appears to have hung.

The `yoyo` program is part of the Trustworthy Host-System Programs and is also part of the basic LB Linux system. If you haven't built the Trustworthy Host-System Programs, and the host system is not LB Linux, you'll need to build and install the `yoyo` program somewhere on your `PATH` for this to work properly.

Commands:

```
yoyo /home/random/cbltools/bin/qemu-system-aarch64 \
  -kernel /tmp/cblwork/sysroot/scaffolding/boot/kernel \
  -machine type=virt \
  -cpu cortex-a57 \
  -smp cpus=8 \
  -append "root=/dev/vda1 \
  console=ttyAMA0 \
  ro init=/scaffolding/sbin/init" \
  -drive file=/tmp/cblwork/cbl.qcow2,index=0,media=disk \
  -drive file=/tmp/cblwork/swap.qcow2,index=1,media=disk \
  -nographic -m size=8192
```

10.3. Copy Files After the Build is Complete

After the target build is complete, it's often helpful to copy some files to and from the image files.

Just as when we created the target images, we'll use `qemu-nbd` for this purpose. We may not be able to use the same NBD device as we did before, though, since other processes may be running and they may also be using NBD, so we'll start by removing the previous symbolic links and finding an unused device again — really, this is just like the setup we did before we ran the target-side build.

Commands:

```
rm /tmp/cblwork/build/nbd{dev,part}
bash /tmp/cblwork/build/find_available_nbd
sudo /tmp/cblwork/build/qemu_nbd_connect
partition="$(readlink -f /tmp/cblwork/build/nbddev)p1"
ln -s $partition /tmp/cblwork/build/nbdpart
sudo mount /tmp/cblwork/build/nbdpart /tmp/cblwork/build/mnt
```

The most important part of this exercise is to copy the final system kernel image file. We often use

QEMU as the boot loader for LB Linux systems, and you need to have the Linux kernel file available to QEMU to do that.

Commands:

```
cp /tmp/cblwork/build/mnt/boot/kernel* /tmp/cblwork
```

I often like to refer back to the log files produced during the CBL process. We have the log files from the host side of the build, and the console output from the target virtual machine; but there are also logs produced from the target build in the `/root/cbl-logs` directory on the final system, as well as in all of the package user `logs` directories.

Commands:

```
mkdir /tmp/cblwork/logs/target
sudo chown -R --reference=/tmp/cblwork/logs/target \
  /tmp/cblwork/build/mnt/root/cbl-logs
sudo cp -a /tmp/cblwork/build/mnt/root/cbl-logs /tmp/cblwork/logs/target/
mkdir /tmp/cblwork/logs/target/package-users
pushd /tmp/cblwork/build/mnt/usr/src
tar -c -f - */logs | \
  (cd /tmp/cblwork/logs/target/package-users; tar -x -f -)
popd
```

Now that we've done that, we can unmount the target filesystem and remove the NBD association, as before.

Commands:

```
sync
sudo umount /tmp/cblwork/build/mnt
sudo /tmp/cblwork/build/qemu_nbd_disconnect
```

The Target-Side Build

The second half of the CBL process, which is executed by the bash script created in Write the Scaffolding Init Script, runs on the target system. It uses the scaffolding prepared using the cross-toolchain to build a complete minimal GNU/Linux system.

As with the host-side build, there are a few distinct stages to the target-side build: first, we're going to get the target system to the point that the package users framework is installed; then we're going to install all the real target system components using that framework. Finally, once all the software that comprises the target system is built and installed, we'll install a boot loader so that the computer will actually load and run the Linux kernel when it's turned on, and set up an `init` framework that will get the system to a working and useful state.

Chapter 11. Target Side Of The CBL Process, Through Package-Users

As with other portions of the CBL process, we set up a database directory that will be used by litbuild-generated scripts to figure out if they need to be run and bail out if not.

Environment variable: LITBUILDDBDIR

`/scaffolding/litbuilddb`

The first thing to do is adjust the scaffolding GCC so it knows how to link programs, and set up some symbolic links that will be used for the first several packages.

11.1. Ensure That Files Are Owned By Root

Depending on how the target root filesystem — which at this point only includes the `/scaffolding` directory — was built, some or all of it may be owned by a user other than root. This leads to a couple of problems.

First, as soon as the `lbl` user is created, it's likely that it will suddenly own most of the scaffolding and the top-level directory in the filesystem, which is weird and ugly.

Second, and more important, the configuration file repository setup process won't work if anything being put into the repository is owned by a user that does not exist, and this will often be the case for the top-level directory in the filesystem.

So, before we do anything else, we're going to make sure everything is owned by root. We can ensure the top-level directory has the correct permissions, as well.

Commands:

```
chown 0:0 /
chmod 755 /
chown -R 0:0 /scaffolding
```

11.2. Adjusting the GCC specs (scaffolding-gcc phase)

For an overview of specs-adjustment, see Adjusting the GCC specs.

Just like the specs file had to be adjusted for the cross-gcc so that it would use the correct dynamic loader, the target-native gcc we built as part of the scaffolding has to be adjusted — otherwise, the programs it builds won't be able to find the dynamic loader. Also, we have to override GCC's notion of the header file location.

Commands:

```
gcc -dumpspecs | \
  sed -e 's@/lib/ld@/scaffolding/lib/ld@g' \
```

```
  -e 's@/lib32/ld@/scaffolding/lib/ld@g' \
  -e 's@/lib64/ld@/scaffolding/lib/ld@g' \
  -e 's@/libx32/ld@/scaffolding/lib/ld@g' \
  -e '/^\*cpp:$/ { n; s/$/ -isystem \/scaffolding\/include/ }' > \
  $(dirname $(gcc --print-libgcc-file-name))/specs
```

11.3. Create Symbolic Links For Bash

Some programs and libraries have build processes that assume that there is a shell program at `/bin/bash` or `/bin/sh`. That assumption is wrong during the first part of the target-side build, because we haven't built the final system `bash` yet. So for now, we can set up symbolic links that point to the scaffolding `bash`.

We'll remove these symbolic links right before we install the final system `bash`.

Commands:

```
ln -s /scaffolding/bin/bash /bin/bash
ln -s /scaffolding/bin/bash /bin/sh
```

We can't start building the final system programs and libraries yet, because we need some more components that are difficult or impossible to cross-compile. These are still part of the scaffolding, and are installed in `/scaffolding` just like the other components. Since they're built from within the target system as native programs, CBL refers to these as "target scaffolding" components.

11.4. lzip

Name	Lzip compression utility
Version	1.23
Project URL	http://www.nongnu.org/lzip/lzip.html
SCM URL	*(unknown)*
Download URL	http://download.savannah.gnu.org/releases/lzip/

11.4.1. Overview

Lzip is a data compression program that uses the LZMA algorithm, just like XZ Utils does. Because it doesn't have the same convoluted project history as XZ Utils, though, it seems to have a slightly tidier codebase and a simpler format for compressed files. (The lzip homepage observes, "The lzip manual provides the code of a simple decompressor along with a detailed explanation of how it works, so that with the only help of the lzip manual it would be possible for a digital archaeologist to extract the data from a lzip file long after quantum computers eventually render LZMA obsolete" — which seems like a good idea if you're going to use a program for long-term archival storage.)

Also, it tends to compress files just a litle bit better than XZ Utils does, and substantially better than

other common compression utilities.

All of the CBL source materials (except lzip) are compressed with lzip because it provides better space savings than any of the other compression programs.

11.4.2. lzip (pre-target-scaffolding phase)

Lzip doesn't make any allowances for being cross-compiled: if `host`, `target`, and so on are specified, they are simply ignored. That means it really can't be built until we are booted into the target system. We need to build it before we build anything else on the target system, though, because the other source code archives are all lzip-compressed!

Configuration commands:

```
bash ./configure --prefix=/scaffolding
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

11.5. Construction of scaffolding as native programs

We're finally in the target system and able to start building things there! But, unfortunately, we can't start right in on the final system components. There is one more set of programs we need to build into the `/scaffolding` area first: programs and libraries that are problematic to cross-compile.

Since these packages need to be compiled natively, we needed to wait until we got the target system working to address them. But these are not going to be a part of the final system any more than the other pieces of scaffolding are, so like the other components we've built up to this point we will install these packages into `/scaffolding` and avoid touching the rest of the filesystem.

(There's one exception: the `shadow` package is going to be set up to store its configuration files in the `/etc` directory, which will of course be part of the final system. But those are just configuration files — there are no binaries there.)

Since we're finally in the target system, we don't have to worry about `DESTDIR` any more — we can just set these to install to `/scaffolding`, since it's now in the correct location as a top-level directory.

Dependencies

Create Symbolic Links For Bash.

On the other hand, we *do* need to worry about `bash` being installed in a non-standard location: the `configure` scripts provided with these packages have an interpreter directive (a.k.a. "shebang" line) that tells them to run under `/bin/sh`, which doesn't exist yet; all we have is `/scaffolding/bin/bash`. We work around this for the native scaffolding builds as well as the rest of the initial target-system setup, up to the point where we build the final system's bash shell, by setting up symbolic links from `/bin/bash` and `/bin/sh` to the scaffolding bash. That's a little bit kludgy, but the other workarounds I've thought of have been worse.

We'll use the same symbolic-link trick for a few other programs that are expected to reside at standard system locations — for example, the `perl` build expects there to be a `/bin/pwd` program at that specific path — but since those are only needed for specific packages we will create those as needed and remove them once they have served their purpose.

Notice that, here, we're building and installing all this stuff as root! That's not the practice we use when setting up the final system components, but for this handful of programs there's really no point in worrying about any potential problems from building and installing as root. We'll be able to check, trivially, that they haven't done anything improper to the rest of the system — because we don't really *have* the rest of the system yet, anything outside of `/scaffolding` is a sign of an issue! And once the final system is installed we're going to delete `/scaffolding` entirely; so if anything installed here clobbers other scaffolding files or anything like that, it won't cause any long-term problems.

11.5.1. tcl

Name	Tcl (Tool Command Language)
Version	8.6.13
Project URL	http://www.tcl.tk/
SCM URL	*(unknown)*
Download URL	http://www.tcl.tk/software/tcltk/download.html

11.5.1.1. Overview

Tcl is an interpreted programming language. I don't use it for anything, myself, so I can't say whether or not it's a good language to learn. For CBL, the driving aspect is that we want to run the automated test suites for the toolchain components, and some of those test suites are implemented in the DejaGnu framework. DejaGnu depends on the Expect tool, which in turn is an extension to Tcl; ergo, to run the toolchain tests, we need Tcl.

The distributed tarfile for this package has a nonstandard name and unpacks to a nonstandard location, so you'll need to modify it appropriately or use the one from the files.freesa.org repository.

11.5.1.2. tcl (target-scaffolding phase)

Build Directory	`unix`

Tcl has support for cross-compilation built into its configure script, but it doesn't seem to work very well so it's built on the target side instead.

Build Directory

`unix`

This package is one of the irritating ones that necessitate a symbolic link at `/bin/sh` during the first part of the target-side build: there are a lot of subdirectories under `/pkg` that have configure scripts with a shebang path of `/bin/sh`. The build process for tcl runs these configure scripts without specifying an interpreter, even if `CONFIG_SHELL` is specified.

There are other alternatives we could use rather than creating the bash symlinks: we could, for example, modify `Makefile.in` so that instead of running `i/configure` it runs `bash i/configure`. But the symbolic links approach isn't really *bad* and is a lot less work.

Configuration commands:

```
bash ./configure --prefix=/scaffolding
```

The tcl package includes a copy of the sqlite database engine, to make it easy to use from tcl. That's probably a good idea in most circumstances, but the main sqlite source code file is over seven megabytes in size, and some of the target CBL systems are quite resource-constrained (for example, QEMU-emulated mipsel virtual machines have a maximum of two gigabytes of RAM, and the GnuBee personal cloud devices have a paltry half-gigabyte) and can have issues compiling it. And we don't need that package for this scaffolding version of tcl!

Configuration commands:

```
rm -rf ../pkgs/sqlite3*
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

Normally, Tcl doesn't install some of its header files because they define internal-only functions and data structures — since they are intended only to be used inside Tcl, they shouldn't be necessary when building other packages. However, the Expect package is an *extension* to Tcl and, as such, makes use of some Tcl internals.

Installation commands:

```
make install-private-headers
```

11.5.2. expect

Name	Expect
Version	5.45.4
Project URL	http://expect.sourceforge.net/
SCM URL	*(unknown)*
Download URL	https://sourceforge.net/projects/expect/files/Expect/
Patches	• `expect-5.45.4-update-config-guess-1.patch`
Dependencies	tcl

11.5.2.1. Overview

Expect is a tool for automating interactive applications, particularly command-line applications like `ftp` and `passwd`. It is written as an extension to Tcl.

On some servers, the version of config.guess distributed with expect is too old to recognize the target triplet. It's easy enough to update it to the version distributed with GCC.

Patch:

- expect-5.45.4-update-config-guess-1.patch

11.5.2.2. expect (target-scaffolding phase)

As with Tcl, Expect is built as part of the CBL scaffolding because it's needed to run the automated tests for some toolchain components.

Configuration commands:

```
bash ./configure --prefix=/scaffolding --with-tcl=/scaffolding/lib \
  --with-tclinclude=/scaffolding/include
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

Expect installs one of its libraries into a subdirectory of the normal lib directory, and sets an RPATH in the `expect` binary so it can find the library. Since we're going to reset the RPATH in *all* programs and libraries shortly, this is a bad idea; we can just move the library to the normal location to avoid problems.

Installation commands:

```
mv /scaffolding/lib/expect*/* /scaffolding/lib
```

11.5.3. dejagnu

Name	DejaGnu
Version	1.6.3
Project URL	https://www.gnu.org/software/dejagnu/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/dejagnu/
Dependencies	expect

11.5.3.1. Overview

DejaGnu is a framework for testing programs — essentially, a library of Tcl procedures that can be used to construct a test harness, which then provides a single front-end for a suite of automated tests. Some of the toolchain components have automated test suites that use the DejaGnu framework.

11.5.3.2. dejagnu (target-scaffolding phase)

As with Tcl and Expect, DejaGnu is installed as part of the CBL scaffolding so that the automated test suites can all be run.

Configuration commands:

```
bash ./configure --prefix=/scaffolding
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

11.5.4. perl

Name	Perl 5
Version	5.38.0
Project URL	https://www.perl.org/
SCM URL	git://perl5.git.perl.org/perl.git
Download URL	https://www.cpan.org/src/5.0/

11.5.4.1. Overview

Perl is a general-purpose programming language. It is the oldest of what have historically been thought of as "scripting languages": high-level programming languages, typically interpreted rather than compiled, that make it easy to implement a lot of functionality without a lot of code. It incorporates features from a bunch of other programs (like bash, AWK, and sed).

Other languages, like Python and Ruby, also fit in the "scripting language" niche, but are relative newcomers: Perl has existed since 1987 and has been under active development that whole stime.

11.5.4.2. perl (target-scaffolding phase)

Perl is needed by many of the automated build systems for CBL packages, including toolchain components. The Perl build system has some limited support for cross-compilation, but historically it has not been reliable and is not very complete. There are third-party projects, like perl-cross, to try to work around that lack, but using anything like that would be a pretty heavy-weight addition to the CBL build requirements. So, like the other programs in this section, the scaffolding Perl is built on the target system *per se* after booting the minimal scaffolding userspace.

One of the Perl source files has a hardcoded directory location that causes problems, so we need to adjust it prior to building.

Configuration commands:

```
sed -i 's@/usr/include@/scaffolding/include@g' ext/Errno/Errno_pm.PL
```

The Perl build-configuration scheme is a script called `Configure` that is generally run interactively and allows many options to be selected manually. Of course, in CBL we want to avoid any interactivity during the build process. Luckily, Perl provides a facility for doing that as well: a script called `configure.gnu`. It takes options like autoconf-produced configuration scripts and translates them into an invocation of the `Configure` script. The configure command used here is the one generated by running `configure.gnu --prefix=/scaffolding -Dcc="gcc"`.

The Perl configuration and build automation makes a number of assumptions, not just about what is available on the host system but also where it can be found. So this is a case where we need to

create another symbolic link outside of the `/scaffolding` area during the build. If the build crashed previously — or if the system crashed during the build, as qemu sometimes does — the link will already exist so we should not bail out if that command fails.

Configuration commands:

```
set +e
ln -s /scaffolding/bin/pwd /bin/pwd
set -e
./Configure -ds -e -Dprefix=/scaffolding -Dcc=gcc
```

It appears that when perl 5.30.1 is built with GCC 10.1 using the default settings, the initial "miniperl" program does not work right; I've gotten "Attempt to free unreferenced scalar" messages and segmentation faults. Reducing the optimization level and removing a stack-protector setting appears to work around the issue.

Configuration commands:

```
sed -i -e 's@-O2@-O0@' Makefile
sed -i -e 's@-fstack-protector-strong@@' Makefile
```

The build process for perl sometimes crashes on resource-constrained systems. It's a good idea to retry a couple of times, if this happens.

Compilation commands:

```
make || make || make || make
```

There's no point in running the automated tests for this perl, since it's only going to exist for a little while.

Test commands:

```
(none)
```

Installation commands:

```
make install
rm -f /bin/pwd
```

11.5.5. git

Name	git version control system
Version	2.42.1
Project URL	https://git-scm.com/
SCM URL	*(unknown)*

Download URL	https://mirrors.edge.kernel.org/pub/software/scm/git/
Dependencies	perl

11.5.5.1. Overview

Git is the best version control system around. It was originally written by Linus Torvalds, who had previously been using a program called Bitkeeper to manage the Linux kernel source code. Bitkeeper was a proprietary program, but the kernel developers had permission to use it for free; when that permission was revoked, Linus decided to write an entirely new program to replace it.

In CBL, we use git to manage configuration files, as described in the package-users documentation. (I use git for all kinds of other things as well.)

11.5.5.2. git (target-scaffolding phase)

This is a very basic version of git, with many options disabled; it is only used to set up the repository that is used in CBL to manage configuration files.

Configuration commands:

```
./configure --prefix=/scaffolding --without-openssl
```

Git expects perl to live at `/usr/bin/perl`, so we'll just make that true for a minute.

Compilation commands:

```
set +e
ln -s /scaffolding/bin/perl /usr/bin
set -e
make
```

We skip the automated tests at this point, as is typical for the scaffolding components.

Test commands:

```
(none)
```

Installation commands:

```
make install
rm -f /usr/bin/perl
```

11.5.6. patchelf

Name	PatchELF
Version	0.18.0

Project URL	http://nixos.org/patchelf.html
SCM URL	git://github.com/nixos/patchelf
Download URL	https://github.com/NixOS/patchelf/releases

11.5.6.1. Overview

PatchELF is a little utility for modifying ELF executables and libraries. ELF — which stands for "Executable and Linkable Format" — is a standard format for programs, libraries, object code, and things like that. It's the standard binary format used on GNU/Linux systems.

Among other things, ELF files can have segments that specify runtime behavior. The `interpreter`, if there is one, specifies the program that will be used as a dynamic linker when running the program, to load in all the shared libraries that it needs; the `RPATH` specifies a set of directories that the dynamic linker should look at to find those libraries, in addition to the standard lib directories; `DT_NEEDED` entries can be used to specify what shared libraries the program or library depends on… there are a bunch of other segment types as well. PatchELF lets you modify those segments to change the runtime behavior of a program or library without rebuilding it from scratch.

CBL includes several components that use `libtool`, a part of the GNU build system, to produce programs and libraries. Because of the way that the cross-toolchain and scaffolding is set up, a lot of the scaffolding libraries have an `RPATH` that include host system directories. It might be possible to adjust the build process so that doesn't happen, but it's much easier just to remove the `RPATH` entirely. That's something that PatchELF can do for us.

11.5.6.2. patchelf (target-scaffolding phase)

PatchELF is also useful to fix the dynamic loader path in programs that insist on looking in multilib directory locations for it (like glibc and gcc). Since one of the programs we'll eventually want to adjust this way is the dynamic loader itself, we need the scaffolding `patchelf` to be statically linked.

Configuration commands:

```
CXXFLAGS="-static" ./configure --prefix=/scaffolding
```

Compilation commands:

```
make
```

The tests for PatchELF don't all pass on all systems: at least on some systems, for example, the `no-rpath-kfreebsd-i386` test fails. Since this native scaffolding component is only going to be used a couple of times, to reset the `RPATH` and dynamic loader path during the first stages of the target-side CBL build, we don't need to worry about the automated tests. We can just verify that it did the right thing after we use it.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

11.5.7. popt

Name	popt
Version	1.16
Project URL	http://rpm5.org/files/popt/
SCM URL	*(none)*
Download URL	http://rpm5.org/files/popt/
Patches	• `popt-1.16-update-config-guess-1.patch`

11.5.7.1. Overview

Popt is a library that facilitates command line option parsing. It's similar to the `getopt` functions provided in the C standard library, but has some differences that some open source project teams find worthwhile.

11.5.7.2. popt (target-scaffolding phase)

Configuration commands:

```
./configure --prefix=/scaffolding
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

11.5.8. python

Name	Python
Version	3.12.0
Project URL	https://www.python.org/

SCM URL	https://github.com/python/cpython
Download URL	https://www.python.org/downloads/

11.5.8.1. Overview

Python is an interpreted (or "scripting") language, like perl and ruby.

There are two incompatible versions of python currently available: python 3, which is the recommended and modern version, and python 2, which has been deprecated for some time and was officially retired on 1 January, 2020. Some packages that were written for python 2 have not yet been modified to work with the python 3 interpreter, so it *may* be desirable to have both installed on your system.

In CBL, we always prefer the latest stable version of everything, so this section — for python 3 — installs to the `/usr` directory where most system packages are installed. There's a separate blueprint for python 2, but that version really only ought to be used if you need a package that hasn't yet been ported to python 3; it installs to the `/opt/python2` directory instead. For any package that requires python 2, you can put the `/opt/python2/bin` directory in your `PATH` while building and installing it.

(Python has some configuration logic that's supposed to make it easy to install multiple major versions of the language in the same directory prefix, like `/usr`, without having them conflict with each other, but it doesn't actually work right; that's why the python 2 blueprint installs to a location under `/opt`.)

Note that the python source archive is distributed as `Python-x.y.z.tar` (with a capital P), and unpacks to a directory also with a capital P. The version available from the files.freesa.org repository has been converted to use the standard naming convention, but if you obtain the source archive from the upstream site you'll have to do that yourself.

11.5.8.2. python (target-scaffolding phase)

Python is a build-time dependency of glibc (as of glibc 2.29), so we need a scaffolding version of python. Like perl, it is problematic to cross-compile python, so this happens in the target scaffolding build.

Python's dependencies are built earlier, in the host-scaffolding section, so they don't have to be addressed here.

Configuration commands:

```
./configure --prefix=/scaffolding --disable-ipv6
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

This installation should be available using the program name `python` as well as `python3`. The same applies to some other programs.

Installation commands:

```
ln -sf python3 /scaffolding/bin/python
ln -sf pip3 /scaffolding/bin/pip
ln -sf idle3 /scaffolding/bin/idle
```

11.5.9. rsync

Name	rsync
Version	3.2.7
Project URL	https://rsync.samba.org/
SCM URL	https://git.samba.org/rsync.git
Download URL	https://download.samba.org/pub/rsync/src/
Dependencies	popt

11.5.9.1. Overview

`rsync` is a program that lets you synchronize files or directory structures incrementally and efficiently from place to place — from one location on a computer to a different location, or among multiple computers. It's really handy.

There are two source distribution files for rsync: the first contains the rsync package sources *per se*, and the second is a collection of patches that can optionally be applied to the main source directory to provide additional functionality. We don't apply any of the patches here, so only the basic source distribution is needed.

`rsync` needs the zlib compression library and a library called "popt" that provides functions for parsing command line options. It includes copies of both of those libraries, but in CBL we always prefer to use the latest stable version of all libraries directly from their own distributions. (zlib does not need to be declared as a dependency at the top level because when the target-scaffolding rsync is being built, it will find the host-scaffolding zlib.)

Dependencies

popt.

11.5.9.2. rsync (target-scaffolding phase)

There are optional features of rsync that depend on packages that we are not building as part of CBL, so we simply disable those features.

Configuration commands:

```
./configure --prefix=/scaffolding --with-included-popt=no \
  --with-included-zlib=no --disable-xxhash --disable-zstd \
  --disable-lz4
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

11.5.10. shadow

Name	Shadow utilities
Version	4.14.2
Project URL	https://github.com/shadow-maint/shadow
SCM URL	git://github.com/shadow-maint/shadow
Download URL	https://github.com/shadow-maint/shadow/releases

11.5.10.1. Overview

"Shadow" is a suite of programs that relate to passwords and logins and things. The `passwd` command that lets you change your password, for example, is part of the shadow suite.

Long, long ago, in the mysterious days of the 1970s, the hashed version of user and group passwords appeared directly in `/etc/passwd` and `/etc/group`. It turns out that's a bad idea, because those files have to be readable by everyone and that makes it possible to run dictionary-style attacks against all the passwords used on a server — that is, running every word in the dictionary through the same hashing algorithm, and checking to see whether any of the hashed password entries in `/etc/passwd` matches. One of the ways that issue was addressed was by moving the hashed passwords into `/etc/shadow` and `/etc/gshadow`. That meant that changes had to be made to all the programs that use or manipulate passwords. All of the programs that were affected by this change got bundled together into a single package, and (probably because there's no compelling reason to do things any differently) have stayed bundled together ever since. That's the shadow

package.

The hashing algorithm used by default on most systems — including Little Blue Linux — is SHA-512. This is a good hashing algorithm that is considered cryptographically strong (as of August 2021, anyway), but can be executed fast enough that it's feasible to mount a brute-force attack on a hashed password value — that is, the specified salt value can be added to each possible password, and the resulting candidate passwords can be run through SHA-512 looking for a match to the stored value in `/etc/shadow`, in a matter of minutes on dedicated hardware.

The computational difficulty of cracking passwords like this can be addressed in a variety of ways, but one of the simplest is to run the hashing algorithm more than one time, taking the output of the algorithm in each round and using it as the input of the hashing algorithm for another round. By default, the SHA-512 algorithm is run 5000 times, which is fast enough that passwords can be validated quickly but increases the time taken to brute-force attack a password by a factor of 5000.

In these days of dedicated ASICs used to mine bitcoin, this is not nearly enough to protect passwords from brute-force attacks, so you may want to increase this by setting `SHA_CRYPT_MIN_ROUNDS` and `SHA_CRYPT_MAX_ROUNDS` in `/etc/login.defs`.

11.5.10.2. shadow (target-scaffolding phase)

CBL uses the package users scheme (about which you can read more elsewhere) to track which packages own files and directories and to prevent packages from stepping on files owned by other packages. That means we need to have programs like `adduser`, `addgroup`, and `su` available throughout the final system build. Those programs are part of shadow.

This is the only piece of scaffolding that will be touching the filesystem outside the `/scaffolding` directory: since we're going to be using these programs to create users and groups and things that will be part of the final CBL system, we're setting the system configuration file directory to `/etc` rather than the default `/scaffolding/etc`. If you look at the files that are created there by the installation process, you'll see that none of them are programs or libraries; in fact, none of them are binary files. So it's not really a problem to install them there.

The shadow package allows user names to be up to 32 characters, but limits group names by default to 16 characters. Since we generally create a group for every user, with the same name as the user, it doesn't make sense to restrict group names to be shorter than user names can be.

In version 4.14, a new optional dependency on the `libbsd` library was introduced. Since `libbsd` is not otherwise required, we simply configure this package without it.

Configuration commands:

```
./configure --prefix=/scaffolding --sysconfdir=/etc \
  --with-group-name-max-length=32 --without-libbsd
```

Compilation commands:

```
make
```

A couple of the default values used by `useradd` — which you can see by running `useradd -D` — are wrong for CBL. THe `useradd` program's defaults can be overridden by specifying different values in `/etc/default/useradd`, so we write the new settings there.

Test commands:

```
make check
```

Installation commands:

```
make install
mkdir /etc/default
echo "CREATE_MAIL_SPOOL=no" >> /etc/default/useradd
```

11.6. Remove RPATH from Scaffolding Programs

For some reason — possibly because of the way that libtool is used? — some of the scaffolding binaries have `RPATH` segments that specify locations that were present on the host system but are not present on the target system — a directory relative to the location of the cross-toolchain, or the full host-system sysroot path, for example. (If you don't know what an `RPATH` is, you can review the A Word About The Dynamic Linker section.)

All we need to do here is to find the binaries that have an `RPATH` segment that includes a directory from the original host system, and remove that `RPATH` segment entirely. That way, `LD_LIBRARY_PATH` will be used to find the dynamic object files the binary wants, which is fine. The PatchELF program can be used to do this.

Sometimes, on some machine architectures, we've seen bugs in PatchELF that cause it to corrupt binaries rather than simply modifying their runtime paths. That's less likely to happen when removing the `RPATH` entirely, but it's still a good idea to make a backup of programs and libraries before running PatchELF on them; that's what we do here.

To save time, we only look for program and library binaries in the few directories that should contain them. It's possible that we're missing some, but this appears to be good enough to prevent any problems from happening later in the build.

Commands:

```
cd /scaffolding
find bin lib libexec sbin usr/bin -type f | while read FILE; \
  do \
  if file $FILE | grep -q ' ELF '; \
  then \
  if readelf -a $FILE | grep -q 'Library r.*path:.*sysroot'; \
  then \
  echo "Removing RPATH in $FILE"; \
  cp -a ${FILE} ${FILE}-orig; \
  patchelf --remove-rpath $FILE; \
  fi; \
```

```
  fi; \
  done
```

The first package we install on the final system is the "Package Users" package. This sets up a framework that we can use so that *every file in the final system* clearly shows what package it belongs to.

11.7. package-users

Name	Package-Users Support Files and Scripts
Version	0.8.2
Project URL	http://git.freesa.org/freesa/package-users
SCM URL	http://git.freesa.org/freesa/package-users
Download URL	https://files.freesa.org/
Dependencies	Construction of scaffolding as native programs

One of the main things that distinguishes different families of GNU/Linux distributions is the approach they take toward managing software packages. Redhat systems typically use the `yum` and `rpm` tools to download and install, upgrade, or remove `rpm` files, which are pre-compiled binary packages structured in a particular way. Debian systems, and derivatives like Ubuntu, similarly use the `apt` and `dpkg` programs to manipulate `deb` files, which are pre-compiled binary packages structured in a different way.

Gentoo systems, and derivatives like Funtoo, don't use binary packages at all; they use a program called `emerge` that uses instruction files in a specific format to download, compile, and install packages from source code. `emerge` maintains a database of all the packages that have been installed this way, along with lists of all the files that were installed as part of those packages. When a package is removed, `emerge` consults that database and systematically removes the files it originally installed as part of that package.

CBL takes a lighter-weight approach to package management (although it is a lot more similar to Gentoo than to the systems that use pre-built binary package files): a separate operating-system user is created for each package that's installed, and the package is configured, compiled, and installed *as that user*. As a result, it's obvious which files and directories were installed as part of a particular package: if you run `ls -l`, the owner and/or group for each file will indicate the package that owns it.

Matthias S. Benkmann, who thought up this approach and published it as a "hint" for Linux From Scratch, named it "package users" and developed a number of utility scripts and configuration files to streamline the use of package users in a GNU/Linux system. CBL includes those utilities and related files, substantially customized, extended, and adapted so they fit in well as a part of the Little Blue Linux system, as the `package-users` package. That's what we're going to set up now, and our goal is for the final result of this installation to look as though package-users was installed using the package-users scheme itself. If you want to know more about the package-users approach to system configuration, you can read the document `package-users-manual.txt` found in the package-users package; it's installed in `/usr/share/doc/package-users` on CBL systems. You can also, if you're

interested, find the original LFS "hint" version of the document at: http://www.linuxfromscratch.org/hints/downloads/files/more_control_and_pkg_man.txt

Nothing in the package-users package *per se* needs to be built; all we're doing here is installing it.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

The package users scheme relies on an `install` group, which all package users will belong to (including the `package-users` user that will own this stuff). The convention used in CBL is that the `install` group has GID 9999, and package users and groups have UIDs and GIDs starting with 10000. That leaves user and group IDs from 1000 to 9998 for normal system users.

Installation commands:

```
echo "install:x:9999:package-users" >> /etc/group
echo "package-users:x:10000:" >> /etc/group
echo "package-users:x:10000:10000:Package users \
  framework:/usr/src/package-users:/bin/bash" >> /etc/passwd
```

Now we can install all the helper scripts and the template for package user home directories and stuff like that. Most of that can just be copied in from the tarfiles contents.

Installation commands:

```
chown -R 10000:10000 etc usr
cp -a etc usr /
install -o 10000 -g 10000 -m 755 -d /etc/pkgusr/skel-package/src
install -o 10000 -g 10000 -m 755 -d /etc/pkgusr/skel-package/patches
install -o 10000 -g 10000 -m 755 -d /etc/pkgusr/skel-package/logs
ln -s /etc/pkgusr/bash_profile /etc/pkgusr/skel-package/.bash_profile
ln -s /etc/pkgusr/bashrc /etc/pkgusr/skel-package/.bashrc
chown -R 10000:10000 /etc/pkgusr/skel-package
mkdir -p /usr/share/doc/package-users
cp -a doc/* /usr/share/doc/package-users
chown -R 10000:10000 /usr/share/doc/package-users
```

The package-users startup scripts provide a convenient location to add compilation flags that should be used for all packages — this could be used, for example, to enable some of GCC's

optimizations (like `-O2` or `-Os`) globally for all packages that don't have a specific `CFLAGS` or `CXXFLAGS` environment variable specified in their blueprint.

CBL assumes that you want to use the same set of optimization flags for C and C++ programs. If that's not the case, this is where you should make an adjustment!

Installation commands:

```
echo "export CFLAGS='-O2 -fno-omit-frame-pointer -mcpu=native'" >> \
  /etc/pkgusr/bash_profile
echo "export CXXFLAGS='-O2 -fno-omit-frame-pointer -mcpu=native'" >> \
  /etc/pkgusr/bash_profile
```

Similarly, `MAKEFLAGS` should be set based on the corresponding parameter.

Installation commands:

```
echo "export MAKEFLAGS='-j8'" >> \
  /etc/pkgusr/bash_profile
```

That's it, the package users files are all installed! Pretty simple, right? But there are a few more things left to do to make it look as though the package users files *themselves* were installed as a package user.

Installation commands:

```
cp -a /etc/pkgusr/skel-package /usr/src/package-users
chown -R 10000:10000 /usr/src/package-users
echo $(basename $(pwd) | sed 's@users-@users @') > \
  /usr/src/package-users/.project
chown -R 10000:10000 /usr/src/package-users/.project
cp -a /home/random/materials/package-users*tar* /usr/src/package-users/src
```

When we copied everything to `/`, it changed the ownership of some standard system directories, so let's change them back. This is also a good time to set all the install directories to have the correct group and mode, and put a list of local filesystems (used by some of the package-user scripts to scan for files owned by a particular package) into the `/etc/pkgusr` directory.

Installation commands:

```
chown 0:0 /etc /usr{/bin,/lib,/sbin,}
set_install_dirs
setup_scan_filesystems
```

Typically, package users are manipulated from the `root` account. It's convenient for root to have bash completion set up to use user accounts for some commands. (By doing this, you can do things like type `pinky g` and then hit tab a couple of times and the shell will show you all the users that start with a `g`.)

Installation commands:

```
echo 'complete -o default -o nospace -A user su pinky sudo' \
  >> /root/.bash_profile
```

In systems that follow the CBL process, configuration files are managed in a version-control system. This allows system configuration changes to be tracked and audited, and makes it very easy to revert problematic changes. To distinguish configuration repository changes that are made automatically by litbuild-generated scripts from manual changes, these are always committed as `Little Blue Linux <default@localhost>`.

Installation commands:

```
cfgrepo-init 'Little Blue Linux' default@localhost
```

The way the configuration file repository is set up by default, both `git commit` and `git as-default` will use the same authorship information (name and email address) for commits. All of the configuration repository commits from litbuild-generated blueprints will be added using `as-default`; to make it easy to distinguish those from manual modifications, we can modify the author information used by `git commit`, and add another `as-lbl` alias for convenience.

Installation commands:

```
cfggit config --global user.name 'A Little Blue User'
cfggit config --global user.email 'lbl@freesa.org'
cfggit config --global alias.as-lbl \
  "commit --author='A Little Blue User \
  <lbl@freesa.org>'"
```

The configuration repository can be populated either while individual packages are set up and installed, or after everything is built — really, it doesn't matter, as long as files are added to the repository *before they are manually modified.*

This is done using the `cfggit add` and `cfggit commit` commands (optionally using one of the `as-$user` aliases rather than `commit`). Typically, blueprints should include `configuration-files` directives so this will be done automatically.

We don't want the file listing for package users to include anything under the build directory, but since we `chown`ed it earlier, it will. We can `chown` it back so that doesn't happen.

Installation commands:

```
chown -R 0:0 .
list_package package-users >> /usr/src/package-users/.project
```

And as a finishing touch, we can create the file that litbuild will use in the future to determine whether the package is already installed.

Installation commands:

```
echo $(basename $(pwd) | sed 's@package-users-@@') > \
  /usr/src/package-users/.default
```

The rest of the build will be done in a separate section, using the package users framework we just installed.

Chapter 12. Target Side Of The CBL Process, With Package-Users

Once the package-users framework is installed, we can start building the components that make up the final system.

12.1. Executing the CBL Process Manually, Part Two

If you're executing the build process manually, this is the point where the process you should use to set up packages will change!

TODO WRITE ME TODO WRITE ME TODO WRITE ME TODO WRITE ME TODO WRITE ME

12.2. A Word About Tests

Since the programs and libraries we are building here will comprise the final CBL system, it is *highly* desirable to run all of the automated tests that are provided as a part of the package distributions. Unfortunately, the CBL system is not complete enough, at this stage, to run all the test suites reliably. Some test suites fail; for these, the standard practice in CBL is to log the fact that tests failed, but still continue the build process, by modifying the test command to:

```
make -k check || echo "Exit code $?: continuing anyway"
```

It's a good idea to inspect log files for these packages after the build is complete — when the CBL system is booted into the full userspace — and see whether anything looks problematic. If any errors look worrisome, you can re-do the build for those packages and see whether the issue is resolved..

In a few cases, the test suite is even more problematic than simply failing with an error — for example, it might cause the build to hang for hours. In those cases, my standard practice is to skip the test suite entirely during the CBL build process. For these packages, CBL provides a blueprint called `rebuild-untested-packages` that can be run to rebuild them — and run their tests — once the complete userspace is available.

12.3. A Word About Package Names

In CBL, our standard practice is to use the latest stable released version of everything, and the convention we use for blueprint names is simply to omit any indication of the version number of packages — the kernel blueprint is simply `linux.txt`, rather than `linux-4.13.txt`.

Sometimes that doesn't work, though: for example, there are significant compatibility issues between Python 2 and 3, and some Python programs have not yet been updated to work with Python 3; that means it's desirable to have both Python 2 and 3 installed. CBL blueprints for these old-version packages have a version number suffix in their name: the blueprint for Python 2 is called `python2.txt`.

12.4. Building the System

With that out of the way, we can start building the target system *per se*.

Of course, we start with a toolchain: the CBL system toolchain.

12.5. Construction of the final system C library

Finally we are in a position to build the programs and libraries that will make up the actual CBL system!

We start with the toolchain, because it's the foundation of the system. In particular, we start with the kernel headers and C library: everything (except the kernel) gets linked against the C library, so we need it before anything else, and (as you may recall from when we built the cross-toolchain) the C library needs the kernel headers so that it knows how to invoke system calls.

From this point on, as we build out the final system components, less and less of the scaffolding will actually be needed or used!

12.5.1. linux (final-system-glibc phase)

For an overview of linux, see linux.

Just as in the initial cross-toolchain build, we need to install the kernel header files so that the C library will know how to make system calls properly.

This time, it's a little bit simpler because we don't need to specify a target architecture: the target architecture is the native architecture. But other than that, this is just the same as the cross-toolchain kernel headers installation.

Configuration commands:

```
make mrproper
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
make INSTALL_HDR_PATH=_dest headers_install
cp -rv _dest/include/* /usr/include
rm -rf _dest
```

12.5.2. glibc (final-system-glibc phase)

For an overview of glibc, see glibc.

Build Directory	`../build-glibc-2`

Dependencies

linux.

This will be the C library for the final system. It's kind of a big deal.

As with many toolchain components, glibc should always be built from outside the source tree.

Build Directory

`../build-glibc-2`

The rest of the configuration is pretty typical for a system glibc. As usual, for the default CBL process we use options as close to the defaults as possible. An adjustment you might want to make is to specify `--enable-obsolete-rpc` in the configuration — this will install obsolete header files related to remote procedure calls, which might still be used by very old packages. If you don't know that you will ever use a program that might need those headers, it's safe to skip it.

Since we are going to install the timezone tools from the `tzdb` package, we don't need the ones that are provided with glibc; these can be skipped with `--disable-timezone-tools`.

Configuration commands:

```
CFLAGS="-g -O2" ${LB_SOURCE_DIR}/configure --prefix=/usr \
  --disable-timezone-tools
```

Compilation commands:

```
make
```

It is really a good idea to run the test suite for glibc — as previously described, it is the most foundational package in the system. Unfortunately, this is sometimes very problematic in the limited scaffolding environment: with release 2.31, and with some other earlier versions, the test suite for glibc leaves some processes still running. The result is that the build hangs after running the glibc tests.

It's disappointing to skip the tests at this point, but seems like the most pragmatic approach.

Even if we ran the tests at this point, we'd expect to have a couple of hundred test failures — because the CBL system isn't on a network while we're doing this build; the final system GCC isn't installed yet and the `libgcc_s.so` library isn't always found in the `/scaffolding` directory structure; and, generally, the glibc tests are just fragile.

It's a good idea to rebuild glibc and run its full test suite after the CBL system is complete. If any test failures that occur then seem problematic, it's a very good idea to follow up on those!

Test commands:

```
(none)
```

Glibc has a post-installation sanity check script, `test-installation.pl`, that compiles a simple program, runs `ldd` against it, and compares the output to what it expects. Sometimes this script works, but in at least some cases it decides that it's a problem for the dynamic linker to find `libgcc_s.so.1` under the `/scaffolding` directory. Since that library is a part of GCC, and the final system GCC isn't installed yet at this point, there's no other `libgcc_s` for it to find. There are other issues, as well — for example, it sometimes tries to link the test program against libraries that aren't even being built.

Since we're going to be doing our own sanity check for the final system toolchain, we can simply take the easy work-around of skipping the one glibc provides.

Installation commands:

```
pushd ${LB_SOURCE_DIR}
sed '/test-installation/s@$(PERL)@echo skipping@' -i Makefile
popd
```

The rest of the installation is pretty typical.

The dynamic linker is configured using a file `ld.so.conf`, which is not created by the glibc installation; we just create an empty one. Also, the "locales," which are used by glibc for internationalization and localization support, don't get installed by default. Strictly speaking, those aren't *required*, but it's a good idea to have at least your own locale configured, and having them all doesn't take up very much extra time or space.

Installation commands:

```
touch /etc/ld.so.conf
make install
make localedata/install-locales
```

The glibc package includes a daemon program called `nscd`, the "name service cache daemon," which — as the name implies — caches the results of certain types of database lookups. It is not necessary, and not particularly useful unless there are a lot of users or groups, or you're using a distributed authentication database like LDAP, or… maybe there are other circumstances where it is useful. For CBL, we simply skip it entirely.

12.5.3. Adjusting the GCC specs (final-system-glibc phase)

For an overview of specs-adjustment, see Adjusting the GCC specs.

Dependencies

glibc (final-system-glibc phase).

After the final system libc is installed, we can remove the adjusted specs file that causes gcc to use the program interpreter from the scaffolding C library. That will cause its behavior to revert to normal, which is what we want from this point forward.

Commands:

```
rm -f $(dirname $(gcc --print-libgcc-file-name))/specs
```

12.5.4. Set up configuration files for glibc

The GNU C library uses a "Name Service Switch" configuration file to control where to look for name-service information. This blueprint configures it in a way that works fine most of the time.

File /etc/nsswitch.conf:

```
passwd: files
group: files
shadow: files
hosts: files dns
networks: files
protocols: files
services: files
ethers: files
rpc: files
```

The dynamic linker from the GNU C library finds shared libraries in a set of pre-configured locations, plus whatever directories are named in the `ld.so.conf` configuration file. Conventionally, the `/usr/local` directory tree has a `lib` directory; in CBL, where all packages can be intentionally installed as a part of the system, the whole `/usr/local` structure is of questionable value, but there's no reason not to set it up.

File /etc/ld.so.conf:

```
/usr/local/lib
```

After modifying `ld.so.conf`, it's a good idea to run `ldconfig` so that the dynamic linker's cache contains information about all the libraries installed on the system.

Commands:

```
ldconfig
```

12.5.4.1. Complete text of files

/etc/ld.so.conf

```
/usr/local/lib
```

/etc/nsswitch.conf

```
passwd: files
group: files
shadow: files
hosts: files dns
networks: files
protocols: files
services: files
ethers: files
rpc: files
```

12.5.5. Verify that a toolchain works properly (final-system-glibc phase)

For an overview of verify-toolchain, see Verify that a toolchain works properly.

Dependencies

Adjusting the GCC specs (final-system-glibc phase), Set up configuration files for glibc.

This is very similar to the previous toolchain test. Since we're running on the target device at this point, we don't need to run it through the emulator, though.

File /tmp/cblwork/build/hello.c:

```
#include <stdio.h>
int main(void)
{
    printf("Hello, Real Live CBL System World!\n");
    return 0;
}
```

As before, compile it:

Commands:

```
gcc /tmp/cblwork/build/hello.c -o /tmp/cblwork/build/hello
```

Make sure it isn't trying to find a dynamic loader in the `/scaffolding` directory:

Commands:

```
readelf -a /tmp/cblwork/build/hello | tee /tmp/cblwork/build/program_info
grep 'interpreter: /lib' /tmp/cblwork/build/program_info
```

And run it!

Commands:

```
/tmp/cblwork/build/hello | grep 'Hello, Real Live CBL System World'
```

12.5.5.1. Complete text of files

/tmp/cblwork/build/hello.c

```
#include <stdio.h>
int main(void)
{
    printf("Hello, Real Live CBL System World!\n");
    return 0;
}
```

12.6. Construction of the final system toolchain

Once the final system glibc is installed and configured, we can build the rest of the toolchain for the final system. This consists of binutils and GCC — which, of course, have some dependencies of their own.

12.6.1. binutils (final-system-toolchain phase)

For an overview of binutils, see binutils.

Build Directory	`../build-binutils`

Build Directory

`../build-binutils`

On some target systems — I've experienced this with QEMU-emulated MIPS virtual machines — the binutils build can crash, or cause QEMU to abort with a segmentation fault, when building the `gold` linker. If this happens to you, you can change the configure command argument here to specify `--enable-gold=no`.

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/usr --sysconfdir=/etc \
  --enable-shared --enable-gold=yes --enable-64-bit-bfd \
  --enable-plugins --enable-threads --disable-multilib
```

As with some of the other binutils builds (but unlike the host-scaffolding build, oddly enough!), some of the warning messages present in GCC 8 and later can present problems when building the binutils. The same makefile tweak we used earlier can be used to ensure that those warnings are not converted to errors.

Configuration commands:

```
make configure-host
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=stringop-truncation@' bfd/Makefile
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=stringop-truncation@' gas/Makefile
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=format-overflow@' binutils/Makefile
```

By default, binutils installs programs into a multiarch location that includes a target-triplet directory. Since CBL doesn't use multiarch or multilib, this is not necessary and we override the normal behavior (here, and during installation).

Compilation commands:

```
make tooldir=/usr
```

It would be great if all the binutils tests passed, but I always get a few failures.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make tooldir=/usr install
```

12.6.2. gmp (final-system-toolchain phase)

For an overview of gmp, see gmp.

Build Directory	`../build-gmp`

This GMP will be installed in the location where it will live permanently on the CBL system, but because we are enabling C++ support (which we need to do to get all the various dependencies and GCC built) it will actually be set up to link against the C++ standard library in the `/scaffolding` area. We'll fix that after installing the final system GCC.

Build Directory

`../build-gmp`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/usr --enable-cxx
```

Compilation commands:

```
make
make html
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.6.3. mpfr (final-system-toolchain phase)

For an overview of mpfr, see mpfr.

Even though gmp is installed in the conventional location for system libraries (`/usr`), the scaffolding programs wind up finding the one in `/scaffolding/lib` during this build instead, unless told explicitly where to look.

Configuration commands:

```
./configure --prefix=/usr --with-gmp=/usr
```

Compilation commands:

```
make
```

Another issue is that the automated test programs are built with an `RPATH` that puts the `/scaffolding/lib` directory before the directory that contains the freshly-built `libmpfr.so`. There are several ways to fix that, but the easiest one is to use `LD_PRELOAD` to get the dynamic linker to do what we want. (If this paragraph confused you, you might want to review the section A Word About The Dynamic Linker.)

There's also a regression in MPFR 4.2.0, when building against glibc 2.37: the output of `mpfr_vsprintf` is formatted incorrectly, leading to a test failure. To get past that roadblock, we'll just ignore the error for now.

Test commands:

```
LD_PRELOAD=${LB_SOURCE_DIR}/src/.libs/libmpfr.so make check \
  || echo "exit code $?, proceeding anyway"
```

Installation commands:

```
make install
```

12.6.4. mpc (final-system-toolchain phase)

For an overview of mpc, see mpc.

Configuration commands:

```
./configure --prefix=/usr --with-gmp=/usr --with-mpfr=/usr
```

Compilation commands:

```
make
make html
```

Test commands:

```
make check
```

Installation commands:

```
make install
make install-html
```

12.6.5. isl (final-system-toolchain phase)

For an overview of isl, see isl.

Configuration commands:

```
./configure --prefix=/usr --with-gmp-prefix=/usr
```

Compilation commands:

```
make
```

One of the tests fails on some of my builds.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

GDB is the GNU debugger. ISL includes a Python script that provides pretty-printers for most of the structures it defines when using GDB; if you ever wind up debugging a program that uses ISL, it's handy to have those pretty-printers available.

For some reason, the build process for ISL doesn't install it in a location where GDB will find it, but it's easy to move it there ourselves.

Installation commands:

```
mkdir -pv /usr/share/gdb/auto-load/usr/lib
mv -v /usr/lib/libisl*gdb.py /usr/share/gdb/auto-load/usr/lib || \
  echo nevermind
```

12.6.6. zlib (final-system-toolchain phase)

For an overview of zlib, see zlib.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
mkdir -pv /usr/share/doc/zlib
cp -rv doc/* /usr/share/doc/zlib
```

12.6.7. gcc (final-system-toolchain phase)

For an overview of gcc, see gcc.

Build Directory	`../build-gcc`

Dependencies

binutils, zlib.

Build Directory

`../build-gcc`

It's hard to believe, but this is the final time you'll need to build GCC! (At least, for this CBL system — unless you want to upgrade it at some point, or add support for additional languages.)

When configuring GCC this time, you may want to enable additional languages; only C and C++ are *necessary*, but if you want to have compilers handy for Objective-C, Go, Fortran, or one of the other languages for which GCC has "front ends," you can add them to the `--enable-languages` list.

Also, if you want to save some time on this build, you can add `--disable-bootstrap`.

One of the libraries that is included in the GCC source distribution (`libiberty`) is also included in the binutils and gdb packages, and possibly others. It contains support functions used by programs in all of those packages. My general practice is to install libraries and header files to system locations under `/usr` so that any program that needs to use the functions in those libraries will be able to do so without needing to rebuild them, but this is problematic with libiberty: the GCC build process compiles a version of libiberty without the `-fPIC` ("position-independent code") option, and if that library is found when binutils is built, binutils will try to use it and fail because it needs a position-independent version of the library. Since it appears that the build processes for these packages expect that there is no system libiberty library, we explicitly tell GCC not to install it.

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/usr --libexecdir=/usr/lib \
  --enable-languages=c,c++ --disable-multilib --with-system-zlib \
  --disable-install-libiberty --enable-fix-cortex-a53-835769 --enable-fix-cortex-a53
-843419 --with-cpu=cortex-a72.cortex-a53 --enable-standard-branch-protection
```

Compilation commands:

```
make
```

The default stack size is insufficient for running the GCC tests, so we bump it up a bit before running them. As with binutils and glibc, some GCC tests will fail; at some point, you should review the test results and see if any of the failures seem problematic. The `test_summary` script can generate a summary of the test results.

At this point in the CBL process, the vast majority of GCC tests either cannot run or fail immediately — probably because some necessary piece of software is not yet installed or working as expected. If you want a better sense for how well the system GCC works against the automated tests, you should probably rebuild GCC after the system build is fully complete.

Test commands:

```
ulimit -s 32768
make -k check || echo "Exit code $?: continuing anyway"
${LB_SOURCE_DIR}/contrib/test_summary
```

Installation commands:

```
make install
```

Several packages expect the C compiler to be available as `cc`, so we create a symbolic link to help them find it.

Installation commands:

```
ln -sv gcc /usr/bin/cc || echo "Link already exists"
```

GCC provides a pretty-printer that can be used with GDB, so we should put that where GDB will be able to find it.

Installation commands:

```
mkdir -p /usr/share/gdb/auto-load/usr/lib
mv -v /usr/lib/libstdc++*gdb.py /usr/share/gdb/auto-load/usr/lib || \
  echo nevermind
```

12.6.8. libxcrypt (final-system-toolchain phase)

For an overview of libxcrypt, see libxcrypt.

It may not be strictly necessary to build `libxcrypt` as part of the final system toolchain. However, since the hashing functions it contains were formerly part of glibc, and glibc is a dependency of every C and C++ program, it seems prudent to make sure the replacement is available for all of the post-toolchain builds.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.6.9. Remove References to `/scaffolding`

12.6.9.1. Overview

You might recall that when the scaffolding was built, back on the host system, some references to host system paths were built into the scaffolding programs and libraries. In just the same way, when some of the final target-system components were built, references to libraries in the `/scaffolding` directory are inserted into those components in `RPATH` segments or in libtool `.la` files. We don't want to link anything against the scaffolding version of any libraries, so it's best to clean

this up.

12.6.9.2. Remove References to `/scaffolding` (final-system-toolchain phase)

Dependencies

binutils (final-system-toolchain phase), gmp (final-system-toolchain phase), gcc (final-system-toolchain phase).

Two libraries were installed in `/usr/lib` with `RPATH` segments specifying the `/scaffolding/lib` location — both of these because they contain references to functions defined in the C++ standard library, which is part of the GCC package.

Since the final system GCC has now been installed — along with the C++ standard library — we don't want to link anything against the scaffolding version of that library.

We're not keeping the original library file around this time — that would be unnecessary clutter in the final system, and if PatchELF writes broken library files it will cause obvious problems in later build stages.

Commands:

```
cd /usr/lib
find . -type f -a -name 'libgmpxx.so*' | while read FILE; \
  do \
  patchelf --remove-rpath $FILE; \
  done
find . -type f -a -name 'libgprofng.so*' | while read FILE; \
  do \
  patchelf --remove-rpath $FILE; \
  done
```

There are also libtool `.la` files related to those libraries, which also contain references to the scaffolding lib directory. I don't know enough about libtool to be sure of whether that could cause problems, but it is easy enough to correct those files.

Commands:

```
sed -i -e 's@/scaffolding/lib/@/usr/lib/@' libgmpxx.la
sed -i -e 's@/scaffolding/lib/@/usr/lib/@' libgprofng.la
```

12.6.10. tzdb

Name	IANA Time zone database files
Version	2023c
Project URL	http://www.iana.org/time-zones
SCM URL	*(unknown)*
Download URL	*(unknown)*

12.6.10.1. Overview

Time zones are really complicated, and the rules for how they behave change more often than you might expect — in 2016, for example, there were ten revisions to the time zone rules. The Internet Assigned Numbers Authority, IANA, releases new timezone database files whenever the rules change.

12.6.10.2. tzdb (final-system-toolchain phase)

It's not clear whether the timezone database is needed this early, but we're including it here because it was historically distributed as a part of glibc and there might be obscure dependencies on it somewhere.

Configuration commands:

```
(none)
```

GCC supports several variants of the C programming language. The default, starting with GCC 13, is the `gnu17` version, which means the ISO C language as revised in 2017 plus some extensions to the language defined by the GNU project. The `zic` program included with the timezone database has had problems when the GNU extensions are present, so from an abundance of caution we explicitly request one of the standard versions that does not include those extensions. The `-std` argument does this.

Compilation commands:

```
make cc='gcc -std=c17'
```

Test commands:

```
(none)
```

Installation commands:

```
make cc='gcc -std=c17' install
```

If you want the system's timezone to be your local time, you can use `tzselect` to identify the correct timezone file and then copy it to `/etc/localtime`. For CBL, the assumption is that the system will be set to GMT, and individual users can override that by setting the `TZ` environment variable in their `.bashrc` or `.bash_profile` scripts.

Once we have the final system toolchain in place, we can use it to build the rest of the programs and libraries that make up the CBL system.

12.7. Construction of the final system components

The foundation of the final CBL system has been laid. Now we can use that foundation, and a gradually-decreasing set of scaffolding programs and libraries, to build the rest of the target system.

Most of the things being built in this section are more-or-less neccessary parts of a GNU/Linux system. Others are really not vital — for example, lsof can be nice to have around, but many people never use it at all. If you'd like to wind up with a more minimal system than base Little Blue Linux is, you can remove blueprints from this section.

12.7.1. Construction of the skarnet.org suite of programs

This installs a suite of related programs and utilities written by Laurent Bercot and published on skarnet.org. The most important of these are s6, which provides the PID 1 `init` program used by Little Blue Linux, and s6-rc, which provides service management functionality using s6 as its basis.

s6 and the various other packages related to it provide all the functionality needed to initialize and manage the system state — but without making any policy decisions about how the system ought to be set up, and without usurping functionality that is provided by (and properly belongs in) other programs. In my opinion, it provides all the benefits of other modern init systems (like systemd or upstart) without the drawbacks they bring.

Although each package within the skarnet.org suite of software is independent from the others and provides distinct functionality, in CBL we often refer to the entire set of packages as "s6" just as a convenient shorthand, instead of using more precise terminology like "s6, s6-rc, and s6-linux-init," when talking about the mechanics of the init process. (The name "s6" is itself a kind of shorthand; it stands for "skarnet.org's small and secure supervision software suite".)

Not all of the skarnet.org packages are absolutely necessary for CBL, but none of them is particularly large, they are easy to build, and they can be very handy! So here we build all the "s6" packages, plus the execline and skalibs packages that they depend on, without worrying too much about whether some components are unnecessary.

The skarnet.org software is all extraordinarily simple and stable code; for CBL, our practice is to apply all the commits on the master branch as a branch-update to the most recent release.

12.7.1.1. skalibs

Name	Skarnet Libraries
Version	2.13.1.1
Project URL	http://skarnet.org/software/skalibs/
SCM URL	*(unknown)*
Download URL	*(unknown)*
Patches	• `skalibs-2.13.1.1-branch-updates-20231030.patch`

Overview

Skalibs is a collection of common routines used by all the skarnet.org software.

skalibs (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --datadir=/etc
```

Compilation commands:

```
make
```

The skarnet.org projects don't have automated test suites.

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/skalibs
cp -r doc/* /usr/share/doc/skalibs
```

12.7.1.2. execline

Name	Execline scripting language
Version	2.9.3.0
Project URL	http://skarnet.org/software/execline/
SCM URL	*(unknown)*
Download URL	*(unknown)*
Patches	• `execline-2.9.3.0-branch-updates-20231102.patch`
Dependencies	skalibs

Overview

Execline is a non-interactive scripting language designed to be used for all the service and process management scripts that will be run as part of the skaware-based (s6, s6-rc, s6-linux-init) system management programs. It serves essentially the same purpose that `bash` does in the sysvinit-style init scheme, but without providing all of bash's rich feature-set.

Providing strictly limited functionality is a huge benefit! Any complexity in the startup process provides room for things to go wrong or be misunderstood.

The easiest way to explain how execline works is to start by describing the way *other* shell programs work, and then talk about how execline is different. You already have a lot of experience

with bash, since that's the basic command-line shell that is invariably part of all GNU/Linux systems.

When running a shell program like bash, the shell displays a prompt to indicate that it's waiting for you to type a command. When you do, the shell parses the command line, then *forks* a subprocess — that is, it uses the `fork` system call to create a new child process that is a copy of itself. The child process then uses the `exec` system call, which replaces the program being run by the current process (the child process) with a different program specified by the command line.

After forking the child process, the parent process — the shell program process itself — sits around and waits for that child process to terminate. When it does, it displays another prompt.

All this happens every time you run a command! When you run the command `ls -l` to show the contents of a directory, bash:

- forks, to create a child process that is a copy of itself, and waits for that child process to end;
- meanwhile, the child process execs into the `/bin/ls` program with the command-line argument `-l`;
- the `/bin/ls` program uses functions defined in the `ls` source code, as well as functions provided by the C standard library, to discover the contents of the current directory, format them into printable strings, displays those strings, and then terminates with an exit code of 0; and
- the original bash process collects that error code, puts it into the `$?` environment variable, and displays a new command prompt.

Actually, There Is A Little More To It Than That

This is a little bit of an over-simplification, because bash has a lot of built-in commands — for example, when you run the command `cd /usr/local/share/doc`, bash doesn't fork or exec a subprocess at all; it just changes its own current working directory to `/usr/local/share/doc` and then displays a new prompt. And the bash parent process doesn't necessarily wait for the child process to terminate, either: when you background a process with `&`, bash interprets the ampersand as being an instruction to display a new command prompt immediately after forking the child process, before the chidl process terminates.

Scripts work the same way as interactive input. When you run a bash script, the program interpreter doesn't need to prompt for commands; it runs each command in the script as though you had entered it on the command line. But the way it runs those commands is just the same: for each command, it forks a copy of itself, and then the copy execs into the command line from the script. (You can type the full text of a shell script at a shell prompt, if you want, and bash will behave just as though you had run that script.)

That brings us to execline. Execline is similar to bash, but it *doesn't fork*. It just execs.

This is different enough from how other scripting languages work that it can be hard to wrap your mind around! (I had to bounce around on the documentation pages for execline and the various s6 sub-projects for a couple of days before I got it.)

The execline program *per se* reads the entire script, parses it into one long command line, and then execs into it. Hence the name "execline:" it *execs* a command *line*.

(For historical reasons, the execline program actually gets installed with the name `execlineb`; in CBL, this gets symlinked to `execline` so you can use either command name, and the execline scripts set up by CBL generally use the command name `execline`.)

The vast majority of the execline "language" is outside of the execline program: the language is made up of dozens of tiny programs that are intended to be used as components in a single long command line. In most cases, these programs do something with one or more of their arguments, and then exec into the *rest of the command line* — a technique sometimes called "chain loading." This is the single most important thing to understand about execline and the various `s6` packages! Almost everything in those packages is a tiny program intended to be used this way, as part of a single long command line. You can think of most of the `s6` packages as being extensions to the execline "language."

Unlike bash, execline doesn't have *any* built-in commands! Take, for example, this simple execline script:

```
#!/bin/execline
cd /usr/local/share/doc
ls -l
```

The first line indicates that the script should be run by the program `/bin/execline`. Execline ignores the comment line and parses the rest of the script into a command line:

```
cd /usr/local/share/doc ls -l
```

Then it execs into that command line:

- it runs the program `cd`, which is one of the programs that make up the execline language, giving it the arguments `/usr/local/share/doc ls -l`.
- The `cd` program changes the current working directory of the process to the directory named in its first argument, in this case `/usr/local/share/doc`, and then execs into the rest of its command line. In this case that means it runs the program `ls` with the argument `-l`.
- The `ls` program parses its own argument, `-l`, and prints out a directory listing in the "long" format.

It's common for an execline script to consist of a bunch of invocations of programs that are part of the execline language (or an extension to it), with one external program at the end. There will be more about this later on!

One of the programs in s6-portable-utils, `s6-echo`, works a lot like the `echo` shell command; it simply writes all its arguments to its standard output stream. If you have an execline script that is not behaving the way you'd like it to, a handy trick I've found is to just add `s6-echo` in front of the problem area; when `s6-echo` is encountered, it prints out the rest of the parsed script and then terminates, so you can see whether the rest of the script looks like you expect it to. If you want to see what environment variables are set at that point, you can additionally add an invocation of `s6-`

`env` before `s6-echo`.

execline (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --exec-prefix=/ --enable-shared \
  --disable-allstatic
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/execline
cp -r doc/* /usr/share/doc/execline
```

The main execline program is installed, as mentioned above, as `execlineb`. This is for historical reasons — the first released version of execline had no support for brace-delimited blocks; when the need for blocks became obvious, the author added that support in a new version of `execline` he called `execlineb` for "execline with blocks." Eventually, the original `execline` program was deprecated and removed, so `execlineb` is the only version still around in current versions of the package. I like just using `execline` in the shebang line of my scripts, so I create a symbolic link for that purpose.

Installation commands:

```
if [ ! -e /bin/execline ]; \
  then \
  ln -s execlineb /bin/execline; \
  fi
```

12.7.1.3. s6-dns

Name	s6 DNS client programs and libraries
Version	2.3.5.5
Project URL	http://skarnet.org/software/s6-dns/
SCM URL	*(unknown)*

Download URL	*(unknown)*
Patches	• s6-dns-2.3.5.5-branch-updates-20231012.patch
Dependencies	skalibs

Overview

This is a DNS client library and collection of related programs. These programs allow you to make DNS queries simply and efficiently.

s6-dns (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --exec-prefix=/ --enable-shared \
  --disable-allstatic
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/s6-dns
cp -r doc/* /usr/share/doc/s6-dns
```

12.7.1.4. s6

Name	skarnet.org small and secure supervision software suite
Version	2.11.3.2
Project URL	http://skarnet.org/software/s6/
SCM URL	*(unknown)*
Download URL	*(unknown)*
Patches	• s6-2.11.3.2-branch-updates-20231027.patch
Dependencies	skalibs, execline

Overview

s6 is the *process supervision* component of the skarnet.org suite of packages. The purpose of s6 is to make sure that processes that are supposed to be running are actually running; if a supervised process ends, an `s6-supervise` process will restart it. All of the `s6-supervise` processes are spawned and managed by a top-level `s6-svscan` process.

The way this works is: each `s6-svscan` process monitors a *scan directory*, which contains any number of *service directories*. The `s6-svscan` process will create an `s6-supervise` process for each of the service directories in the scan directory.

Service directories have various optional and mandatory files and subdirectories in them; the most significant is an executable named `run`, which is typically an execline script that runs the process that is supposed to be managed. The primary action of the `s6-supervise` process is to spawn a subprocess that executes the `run` script in its service directory. After that, it just hangs out and waits. If the process ends, `s6-supervise` will optionally take clean-up actions in a `finish` script if one exists, and then spawns another subprocess to re-execute the `run` script.

If you want to send a signal to the supervised process — for example, many daemon programs will reload their configuration files if they receive a `HUP` signal — you can use the `s6-svc` program, which can tell a `s6-supervise` process to send signals to the process it's supervising. Similarly, if you want to control the state of the `s6-svscan` process, you can use the `s6-svscanctl` program.

Aside from those few primary programs — `s6-svscan`, `s6-supervise`, `s6-svscanctl`, and `s6-svc` — s6 consists of a bunch of programs that extend the execline language and use the same "chain loading" paradigm. The execline extensions in the s6 package are useful for managing processes; for example, s6 contains a program `s6-setuidgid` that simply sets the effective user ID and group ID of the process (and then execs into the command line formed by the rest of its arguments).

s6 (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --exec-prefix=/ --enable-shared \
  --disable-allstatic
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/s6
```

```
cp -r doc/* /usr/share/doc/s6
```

12.7.1.5. s6-linux-init

Name	s6 linux init tools
Version	1.1.1.1
Project URL	http://skarnet.org/software/s6-linux-init/
SCM URL	*(unknown)*
Download URL	*(unknown)*
Patches	• `s6-linux-init-1.1.1.1-branch-updates-20230924.patch`
Dependencies	skalibs, execline, s6

Overview

s6-linux-init is the last component in the s6 `init` system: it provides a program (`s6-linux-init`) that is intended to be run as the initial PID 1. It does a few basic system initialization tasks and then sets up s6 as a process supervisor and s6-rc as a service manager.

This package also provides a handful of other programs that provide a command-line user interface similar to the legacy sysvinit package: for example, it includes a `s6-linux-init-telinit` program (for which a wrapper `telinit` is provided), which allows the selection of any of a number of system states called "runlevels," and other programs that facilitate shutting down or rebooting the system using the same commands that have worked for years with sysvinit.

Since the CBL process builds a system from the ground up using no legacy policy decisions related to the init system, the sysvinit compatibility layer is basically irrelevant here, but as with other skarnet components, having it around adds only a tiny bit of clutter to the system. If even that seems objectionable, it's not difficult to remove the unnecessary components! That's left as an exercise for the interested system administrator.

We're not actually going to set up the init system here; that will be done later, at Configure the system initialization framework.

s6-linux-init (skarnet-org phase)

Configuration commands:

```
./configure --enable-shared --disable-allstatic
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/s6-linux-init
cp -r doc/* /usr/share/doc/s6-linux-init
```

12.7.1.6. s6-linux-utils

Name	s6 linux utilities
Version	2.6.1.2
Project URL	http://skarnet.org/software/s6-linux-utils/
SCM URL	*(unknown)*
Download URL	*(unknown)*
Patches	• `s6-linux-utils-2.6.1.2-branch-updates-20231030.patch`
Dependencies	skalibs

Overview

This is a collection of small utilities that work on Linux systems. Like the s6-portable-utils programs, Many of them are smaller or simpler versions of utilites available in other packages, and you can certainly use those instead. Some of these programs can be very helpful in combination with other s6-related programs, though, like `s6-logwatch`, which is more effective than `tail -f` for log directories managed by `s6-log`.

The s6-linux-utils are required for CBL because s6-linux-init depends on them.

All of the programs in this package are prefixed with `s6-`, so they don't collide with programs installed by different packages.

s6-linux-utils (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --exec-prefix=/ --enable-shared \
  --disable-allstatic
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/s6-linux-utils
cp -r doc/* /usr/share/doc/s6-linux-utils
```

12.7.1.7. libressl (final-system-components phase)

For an overview of libressl, see libressl.

CBL includes multiple TLS libraries. LibreSSL, as one of the primary ones that is commonly used, can be installed directly under `/usr`, so its headers and shared libraries will be found easily by other packages. If a package really needs to be linked against OpenSSL instead, that can be done by specifying appropriate directives when configuring them.

Configuration commands:

```
./configure --prefix=/usr --enable-nc \
  --with-openssldir=/etc/ssl --enable-extratests
```

Compilation commands:

```
make
```

Some of the tests fail on the partial system.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.1.8. s6-networking

Name	s6 networking utilities
Version	2.5.1.3
Project URL	http://skarnet.org/software/s6-networking/
SCM URL	*(unknown)*

Download URL	*(unknown)*
Patches	• s6-networking-2.5.1.3-branch-updates-20231016.patch
Dependencies	skalibs, execline, s6, s6-dns, libressl

Overview

s6-networking is a collection of small networking utilities for Unix systems. Among other things, this package includes command-line client and server management, clock synchronization, and similar programs.

s6-networking (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --exec-prefix=/ --enable-shared \
  --disable-allstatic --enable-ssl=libressl
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/s6-networking
cp -r doc/* /usr/share/doc/s6-networking
```

12.7.1.9. s6-portable-utils

Name	s6 portable utilities
Version	2.3.0.2
Project URL	http://skarnet.org/software/s6-portable-utils/
SCM URL	*(unknown)*
Download URL	*(unknown)*
Patches	• s6-portable-utils-2.3.0.2-branch-updates-20231030.patch
Dependencies	skalibs

Overview

This is a collection of small utilities that work on most Unix systems; they are not GNU- or Linux-specific. Many of them are smaller or simpler versions of utilites available in other packages, like `cat` or `chmod` or `chown`; for these, you will probably wind up using the version from GNU coreutils or whatever other package provides them. But some of the portable utilities, like `s6-update-symlinks`, provide functionality that is not commonly available elsewhere.

The s6-portable-utils are required for CBL because s6-linux-init depends on them.

All of the programs in this package are prefixed with `s6-`, so they don't collide with programs installed by different packages.

s6-portable-utils (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --exec-prefix=/ --enable-shared \
  --disable-allstatic
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/s6-portable-utils
cp -r doc/* /usr/share/doc/s6-portable-utils
```

12.7.1.10. s6-rc

Name	s6-rc service manager
Version	0.5.4.1
Project URL	http://skarnet.org/software/s6-rc/
SCM URL	*(unknown)*
Download URL	*(unknown)*
Patches	• `s6-rc-0.5.4.1-branch-updates-20230922.patch`
Dependencies	skalibs, execline, s6

Overview

s6-rc is a *service manager*, intended to serve the same fundamental purpose as other init systems: it is a suite of programs that can start and stop long-running daemon processes and can run one-time initialization scripts, in the proper order according to a dependency hierarchy. s6-rc ensures that all long-running daemon processes are correctly supervised by s6, and it runs one-time initialization scripts in a controlled (and therefore predictable) environment.

You can read more about how services can be defined in the Construct the s6-rc service database section (or in the s6-rc documentation), and you can read about how it fits into the CBL process in the Configure the system initialization framework section.

s6-rc (skarnet-org phase)

Configuration commands:

```
./configure --prefix=/usr --exec-prefix=/ --enable-shared \
  --disable-allstatic
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
mkdir /usr/share/doc/s6-rc
cp -r doc/* /usr/share/doc/s6-rc
```

12.7.2. attr (final-system-components phase)

For an overview of attr, see attr.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "exit code $?, proceeding anyway"
```

Installation commands:

```
make install
```

12.7.3. acl

Name	POSIX Access Control Lists utilities
Version	2.3.1
Project URL	http://savannah.nongnu.org/projects/acl
SCM URL	*(unknown)*
Download URL	http://download.savannah.nongnu.org/releases/acl/
Dependencies	attr

Linux — and POSIX systems in general — have always supported a basic access-control mechanism that governs who may access which files and directories, and what type of access is permitted; that's the *mode*, which can be modified with `chmod`, and the owner and group for the file, which can be modified with `chown` and `chgrp`.

Sometimes you might want to have more fine-grained access controls, when the simple owner/group/world permission model isn't enough to do what you want. This is supported using "access control lists," which are implemented through extended attributes in the filesystem. The ACL package allows access control lists to be viewed and administered.

The acl test suite can only be run after the GNU coreutils package has been built and linked with the libraries provided by this package. If you want to run the tests, you can do that with `make tests`.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.4. libffi (final-system-components phase)

For an overview of libffi, see libffi.

In the minimal CBL system, the tests all fail.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Even though the configure script for libffi says that the default location for header files is `PREFIX/include`, the installation targets and pkg-config file always install header files to `${libdir}/libffi-${version}/include`. Some packages, like LLVM, don't use pkg-config to find header files, so we can create symbolic links in the canonical system location.

Installation commands:

```
make install
find /usr/lib/libffi*/include -type f | while read filename; \
  do \
  ln -f -s $filename /usr/include; \
  done
```

12.7.5. pkgconf (final-system-components phase)

For an overview of pkgconf, see pkgconf.

Configuration commands:

```
find . -exec touch -r README.md {} \;
./configure --prefix=/usr
```

Compilation commands:

```
make
```

pkgconf has automated tests implemented in the Kyua test framework. Unfortunately, the Kyua framework introduces a cycle: Kyua depends on pkg-config (as well as Lutok and SQLite). We could build and install pkgconf, the other dependencies, and then Kyua, and then re-build pkgconf so that we can run its tests. On the other hand: that's a lot of work just to run pkgconf's automated tests; pkgconf itself is a very simple program that is unlikely to fail dramatically; and if it *does* fail, the only effect of that failure is that some other program will be slightly harder to build.

Test commands:

```
(none)
```

Installation commands:

```
make install
ln -sf pkgconf /usr/bin/pkg-config
```

12.7.6. ncurses (final-system-components phase)

For an overview of ncurses, see ncurses.

Dependencies

pkgconf.

The configuration here is much the same as for the scaffolding version of the library. We don't need to specify `--with-build-cc`, since the system compiler is the one that the build process will find; we enable wide character support; and we tell the build process to generate and install `.pc` files that are used by pkg-config.

Configuration commands:

```
./configure --prefix=/usr --with-shared \
  --enable-overwrite --without-debug --without-ada --enable-widec \
  --with-pkg-config --enable-pc-files --enable-mixed-case \
  --with-cxx-shared --with-versioned-syms
```

Compilation commands:

```
make
```

There are automated tests for ncurses, but they only work against a fully installed ncurses package. If you wish to run the tests, look at the file `test/README` in the source distribution.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

When wide character (UTF-8) support is enabled in ncurses, its libraries are installed with filenames that end in "w" — like `libncursesw.so`, rather than `libncurses.so`. The expectation is that, if you have programs that don't work right with wide character support, you'll install a separate set of ncurses libraries with wide characters disabled.

In most cases, this isn't necessary, because programs that aren't written specifically to use wide-character support generally work fine when linked against the wide-character libraries instead. This is the case for the base CBL programs, and I assume it's the case with other programs that will be installed into the CBL system. (If you run across a problem, you can simply build and an additional version of ncurses with wide-character support disabled.)

Accordingly, we set up linker scripts and symbolic links so that the wide-character libraries and config program will be found by programs that are looking for the old-style ones.

Installation commands:

```
cd /usr/lib
for LIB in menu form ncurses ncurses++ panel; \
  do \
  echo "INPUT(-l${LIB}w)" > lib${LIB}.so; \
  ln -sfv lib${LIB}w.a lib${LIB}.a; \
  done
ln -svf ncursesw6-config /usr/bin/ncurses6-config
```

12.7.7. readline

Name	GNU Readline Library
Version	8.2.1
Project URL	https://tiswww.case.edu/php/chet/readline/rltop.html
SCM URL	http://git.savannah.gnu.org/cgit/readline.git
Download URL	https://ftp.gnu.org/gnu/readline
Dependencies	ncurses

GNU Readline is a library that applications can use if they wish to edit command lines using the same keystroke commands that work in the common text editors Emacs or vi. This is handy for command shells like `bash`, and for the read-evaluate-print-loop interpreters that are generally provided by scripting languages like ruby.

Like bash, patches for readline are found in a separate directory and need to be applied separately. Also like bash, the patches have already been applied to the source archive on files.freesa.org.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make SHLIB_LIBS=-lncurses
```

Test commands:

```
(none)
```

Installation commands:

```
make SHLIB_LIBS=-lncurses install
```

12.7.8. libyaml (final-system-components phase)

For an overview of libyaml, see libyaml.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.9. ruby (final-system-components phase)

For an overview of ruby, see ruby.

Dependencies

readline, libressl, libyaml.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc --enable-shared \
  --enable-debug-env
```

Compilation commands:

```
make
```

Ruby has a lot of tests that rely on the network being available. These all time out after a couple of minutes (each) but there are enough of them that it takes an absurd amount of time for the test target to run… and with enough failures that it would make sense to re-run the tests once the system is complete.

Test commands:

```
(none)
```

12.7.9.1. About Ruby Packages

Ruby programs are almost universally packaged in a format called "ruby gems," which can be created and managed using the `gem` program provided as a part of standard Ruby installations. Gem files are not especially complicated — they are tar files containing a compressed `data` tarfile with the gem contents, a compressed `metadata.yaml` file with a bunch of metadata about the gem, and a compressed `checksums` file with SHA256 and SHA512 checksums of the `data` and `metadata.yaml` files.

The `gem` program allows you to download and install a gem, along with any other gems it depends on, from an external repository — by default, it uses the repository `https://rubgems.org/`, but you can change that if you wish.

There are several different ways you can set up ruby gems on a LB Linux system.

The approach taken by the basic CBL process is to treat them just like other software packages: you start with a tar file containing the complete project source code, build it — which generally just means packaging it as a gem, with the `gem build` command, specifying a gemspec file as a command-line argument — and install it, again typically using the `gem install` command.

This is a fair amount of work, since you have to have a blueprint for every gem package, find and download the source tarfiles for them, and so on. An alternative approach that is substantially less work is to use the `gem` program to download and install gems and their dependencies. If you do this, you won't have each gem installed as a separate package user, of course. You could have a single `ruby-gems` package user to own all gem files; or you could create a package user for each gem that you actually *care* about, and have that package user also coincidentally own all the other gem dependencies that are needed by it. Or you could do something else entirely! It's your system.

Installation commands:

```
make install
```

12.7.10. asciidoctor

Name	Asciidoctor
Version	2.0.20
Project URL	https://asciidoctor.org/
SCM URL	https://github.com/asciidoctor/asciidoctor
Download URL	https://github.com/asciidoctor/asciidoctor/releases
Dependencies	ruby

AsciiDoc is a documentation format based on normal ASCII text files with a simple set of formatting conventions; files written in the AsciiDoc format can be transformed into a variety of document formats.

It is also the format used for the narrative sections of litbuild blueprints; when litbuild produces a human-readable document to tell the story of how to construct a package, it produces an AsciiDoc output file that can then be further transformed into whatever final output form is desired.

The documentation for some packages that are part of CBL — notably, the git version control system — is also written in AsciiDoc.

There are at least two programs that can be used to process AsciiDoc files: the original one is a eponymous python-language package, available at `https://asciidoc.org/`; there is also a more recent ruby-language implementation called Asciidoctor. For CBL we prefer Asciidoctor, primarily because it's still under active development: the most recent release of AsciiDoc, as of this writing, was made in November of 2013, while the current version of Asciidoctor was released in April 2019.

Since Asciidoctor is distributed as a ruby gem, it doesn't really have to be built from a source package — you can simply `gem install asciidoctor` and Bob's your uncle. Here, though, we're still going to start with a source tarfile, because that's how we roll. (Eventually, we'll also run the automated test suite as part of the build process, because that's *also* how we roll; but that requires a large number of additional gems to be installed to fulfill dependencies, and I don't want to write blueprints for them all right now.)

Configuration commands:

```
(none)
```

Rubygems — which is provided as part of the ruby package — provides a facility to construct a gem from a source package, using a `gemspec` file that tells it what should be packaged that way.

Compilation commands:

```
gem build asciidoctor.gemspec
```

As mentioned earlier, there are automated tests for Asciidoctor; unfortunately, they can only be run if a bunch of additional ruby packages are installed, and I don't have energy or enthusiasm enough

to write blueprints for them at the moment.

Test commands:

```
echo 'Skipping tests (would be rake test:all)'
```

Installation commands:

```
gem install -l asciidoctor*gem
```

12.7.11. m4 (final-system-components phase)

For an overview of m4, see m4.

This is a standard GNU-build-system package.

I have one test failure in some circumstances — `test-execute.sh` fails, with "test-execute-child subprocess failed: No such file or directory". That probably deserves some attention, but I don't want to abort the build because of it.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.12. autoconf

Name	GNU Autoconf
Version	2.71
Project URL	https://www.gnu.org/software/autoconf/autoconf.html
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/autoconf/
Dependencies	m4

Autoconf is part of the GNU build system, which strives to make it easy to write programs that will run on a wide variety of computer systems by auto-detecting the value of things that vary from system to system (like the number of bits in an `int` variable, or pointer, for example).

If you'd like to know more about the GNU build system, you can watch a video tutorial at https://www.dwheeler.com/autotools/ or read the free autotools ebook you can find at http://freesoftwaremagazine.com/books/.

Some of the automated tests for autoconf require automake to be installed; you might want to re-run the tests after installing automake.

Strictly speaking, autoconf builds and installs perfectly well against the scaffolding version of m4. But the full path to the `m4` program is baked into the script `autom4te`, so it's convenient to have the final system m4 in place already when this package is built.

Dependencies

m4.

At least one test — the test for autoscan — will fail at this point. As with other packages, review the test results after the build to see if anything seems worrisome.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.13. automake

Name	GNU Automake
Version	1.16.5
Project URL	https://www.gnu.org/software/automake/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/automake/

Automake is part of the GNU build system, like autoconf.

The Automake test suite is expansive and surprisingly unreliable — a lot of the `lex` tests, fail, for example. As with other packages, review the test results after the build to see if anything seems especially problematic.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.14. berkeley-db

Name	Oracle Berkeley DB
Version	18.1.40
Project URL	http://www.oracle.com/technology/products/berkeley-db/index.html
SCM URL	*(unknown)*
Download URL	*(unknown)*
Dependencies	libressl
Build Directory	`build_unix`

Berkeley DB is a library that provides lightweight database functionality. It is used by a wide variety of programs that need database functionality — reliable and fast transactions, for example — but don't want to use an external RDBMS like PostgreSQL or MySQL.

Downloading the Berkeley DB source release from Oracle currently requires you to sign up for an Oracle account. It's a little frustrating. The tarball also has the non-conventional name `db-` and must be repackaged for use in CBL. You can fetch the tarfile from files.freesa.org instead, if you wish.

This package supports TLS connections if OpenSSL or LibreSSL are avialable.

Dependencies

libressl.

Build Directory

`build_unix`

Berkeley DB has gone through several versions. Programs that are designed to use version 1.85 of Berkeley DB can't normally use modern versions; there's a configure flag to enable support for that database file version. You can enable it if you think you'll need any program that needs it.

Configuration commands:

```
../dist/configure --prefix=/usr --disable-compat185 --enable-cxx
```

Compilation commands:

```
make
```

To run the automated test suite for Berkeley DB, you need to have TCL installed, and you need to configure with `--enable-tcl` and `--enable-test`. After building the database, run `tclsh` to run the TCL shell; then run `source ../test/test.tcl`, `run_parallel 5 run_std`, and `exit` from the TCL shell. The tests run for several hours.

The documentation suggests that, after running the tests, it's a good idea to rebuild the package without the `--enable-test` switch.

Given the arduous nature of the test suite, CBL skips it.

Test commands:

```
(none)
```

Two of the documentation files that the Makefile tries to install don't seem to be built. If you care about Berkeley DB enough to look into what's going on here, and figure out the right thing to do about it, please let us know. All we're doing here is telling the Makefile not to do anything with the files that don't exist.

Installation commands:

```
sed -i -e 's@bdb-sql@@' Makefile
sed -i -e 's@gsg_db_server@@' Makefile
make docdir=/usr/share/doc/berkeley-db install
```

12.7.15. gperf

Name	GNU perfect hash function generator
Version	3.1
Project URL	https://www.gnu.org/software/gperf/
SCM URL	*(unknown)*
Download URL	https://mirrors.kernel.org/gnu/gperf/

GNU gperf is a hash function generator. For any set of keywords, it can produce C or C++ source

code for a hashing function that will recognize any of the input keywords with a single operation against a lookup table.

This isn't directly important for most people! As with many components of CBL, gperf is a part of the system only because other important components rely on it.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.16. bzip2 (final-system-components phase)

For an overview of bzip2, see bzip2.

Configuration commands:

```
(none)
```

bzip2 installs a library that can be used by other programs to do bzip2 compression and decompression.

As before, we need to force `libbz2` to be compiled with `-fPIC`.

Compilation commands:

```
sed -i 's@^CFLAGS=@CFLAGS=-fPIC @' Makefile
make
```

Test commands:

```
make check
```

The Makefile for bzip2 expects the `man` directory to be directly under `/usr` instead of under `/usr/share`.

Installation commands:

```
sed -i 's@$(PREFIX)/man@$(PREFIX)/share/man@' Makefile
make PREFIX=/usr install
```

12.7.17. libcap

Name	Linux Capabilities utilities
Version	2.69
Project URL	https://sites.google.com/site/fullycapable/
SCM URL	*(unknown)*
Download URL	https://git.kernel.org/pub/scm/libs/libcap/libcap.git/refs/

Historically, the privilege model for Unix systems was pretty simple: everyone had a separate account with only basic permissions, and if you needed to do something that was outside those basic permissions (like modify system programs or processes, or mount filesystems, or whatever), you would use `su` or `sudo` to elevate your privileges to the super-user account, conventionally called "root". The super-user account has no restrictions whatsoever.

This isn't always the best model to use! It leads to situations where sometimes processes are run as root, with full permissions to do anything at all to the system, when the only non-standard permission they need to do is bind to a TCP port lower than 1024, or something of that sort. (It's common for web servers to do whatever they need root privileges for *first*, and then drop those privileges before they do anything else, precisely because it's a bad idea for processes to run with superuser privilege.)

That's why the "capabilities" feature was added to Linux. If a program needs to be able to bind to TCP port 300, it can be given the `CAP_NET_BIND_SERVICE` capability; it will be able to bind to low ports, but it won't be able to destroy the root filesystem.

libcap is a package for setting and viewing these capabilities.

The libcap package doesn't use the GNU build system, so there isn't a configure step.

Configuration commands:

```
(none)
```

Compilation commands:

```
make
```

Test commands:

```
make test
```

The install target for libcap runs a command that requires root privilege, unless that behavior is overridden by setting the `RAISE_SETFCAP` variable. We'll try to run that command as root, post-installation.

Installation commands:

```
make RAISE_SETFCAP=no install
```

Post-installation (as `root`) commands:

```
/sbin/setcap cap_setfcap=i /sbin/setcap || echo nevermind
```

12.7.18. coreutils (final-system-components phase)

For an overview of coreutils, see coreutils.

Many of the programs in coreutils can support extended attributes and file capabilities, if the system has support for these. For example, `cp` will be able to copy extended attribute metadata as well as file data if the support library is available when the coreutils are built. This is worthwhile, so we specify dependencies that aren't strictly necessary.

Dependencies

attr, acl, libcap.

By default, the `hostname` program isn't installed, and `kill` and `uptime` are. We want to override this behavior; CBL uses the `kill` and `uptime` programs from the util-linux and procps packages, respectively, and we want `hostname`.

Configuration commands:

```
./configure --prefix=/usr \
  --enable-no-install-program=kill,uptime \
  --enable-install-program=hostname
```

Compilation commands:

```
make
```

The coreutils package has tests that can only be run as root, as well as tests that can be run by normal users. If you want to run the as-root tests, you can do that with `make NON_ROOT_USERNAME=coreutils check-root` — but you'll need to do that outside of the standard build process, since the package users system means that the tests are not normally run as root.

Some coreutils tests don't reliably pass, so we do the usual thing and throw in an `|| echo` to force the pipeline to succeed and allow the build process to continue; as always, it's a good idea to review the test results manually.

I've also occasionally seen the tests hang entirely, but this is rare and I can't reproduce it so I don't

have a solution for it, other than to skip the test suite entirely — and since the issue only comes up very occasionally, that seems unnecessarily extreme.

Test commands:

```
make RUN_EXPENSIVE_TESTS=yes -k check || \
  echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.19. bison (final-system-components phase)

For an overview of bison, see bison.

Environment	• MAKEFLAGS: -j1

Environment variable: MAKEFLAGS

```
-j1
```

Sometimes this build is fragile when used with a lot of parallel make processes. To avoid any issues, we can just disable `make` parallelism.

The flex package depends on bison to build, but bison depends on flex to run its tests. Circular dependencies are always frustrating! This one can be avoided just by skipping the bison tests.

If it's really important to you to run the tests, rebuild bison after installing flex.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.20. flex

Name	The Fast Lexical Analyser
Version	2.6.4
Project URL	https://github.com/westes/flex
SCM URL	*(unknown)*
Download URL	https://github.com/westes/flex/releases
Dependencies	bison

Flex is the "Fast Lexical Analyser." It is a tool for generating "scanners", which are programs that recognize lexical patterns in text. This is mostly useful when writing compilers: the part of a compiler that scans source code and turns it into tokens is usually generated using a lexical analyzer like flex.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

In at least some cases (observed with the x86 x32 ABI), the flex test suite fails with a segmentation fault in the cxx_restart test. That might be because of a flaw in the scaffolding C++ compiler, or something related to the ABI, but it's not important enough to crash the build.

Installation commands:

```
make install
```

The original lexical analyzer used on UNIX systems was called `lex`, and some programs still try to use it. Flex has a lex emulation mode; CBL therefore sets up a wrapper script that invokes flex in lex emulation mode when programs try to run lex.

Installation commands:

```
echo '#!/bin/bash' > /usr/bin/lex
echo 'exec /usr/bin/flex -l "$0"' > /usr/bin/lex
chmod -v 755 /usr/bin/lex
```

12.7.21. iproute2

Name	IP and network traffic control utilities
Version	5.19.0
Project URL	https://wiki.linuxfoundation.org/networking/iproute2
SCM URL	git://git.kernel.org/pub/scm/network/iproute2/iproute2.git
Download URL	https://mirrors.edge.kernel.org/pub/linux/utils/net/iproute2/
Dependencies	flex

iproute2 is a collection of utilities that allow TCP/IP networks to be configured and controlled. Its two main components are the programs `ip`, which controls IPv4 and IPv6 configuration, and the program `tc`, which lets you configure traffic control.

The `ip` program replaces a lot of other programs — when looking at documents and howtos that talk about how to set up networking on GNU/Linux systems, you may find references to programs like `ifconfig` and `route`. The functionality provided by those programs has been subsumed into iproute2.

Similarly:

1. the program `bridge` from this package provides a superset of the functionality implemented by the `brctl` program from the bridge-utils package; and
2. the program `ss` ("Socket Statistics") from this package can be used in place of the program `netstat` from the `net-tools` package; it provides more TCP and state information than `netstat` does.

iproute2 does not use the GNU build system, so there is no configure script.

Configuration commands:

```
(none)
```

Compilation commands:

```
make
```

This package has no automated tests.

Test commands:

```
(none)
```

The programs from this package are installed into `/sbin` by default, but some of them are useful for non-root users to inspect the state of the system. Accordingly, we will move those to `/bin`.

Installation commands:

```
make install
mv /sbin/ip /bin
mv /sbin/ss /bin
```

12.7.22. perl (final-system-components phase)

For an overview of perl, see perl.

Environment	• `BUILD_ZLIB`: `False` • `BUILD_BZIP2`: `0`

The CBL system has zlib and bzip2 already, so the copies bundled with perl are unnecessary. Of course, we have to make sure they are already present on the final system.

Dependencies

zlib, bzip2.

Environment variable: BUILD_ZLIB

`False`

Environment variable: BUILD_BZIP2

`0`

As with the target-scaffolding perl, this Perl build wants to find a `pwd` program in the `/bin` directory.[1] Rather than creating a symbolic link to the scaffolding `pwd` again, we can just force the final system `coreutils` to be built first.

Dependencies

coreutils.

For some reason, perl uses the loopback network device, so it needs to be enabled. It also expects there to be an `/etc/hosts` file, so we might as well provide one.

Dependencies

iproute2.

Pre-build (as `root`) commands:

```
ip link set lo up
echo "127.0.0.1 localhost" > /etc/hosts
```

If something has gone wrong previously and the CBL build has been restarted, it's possible that there's still a symbolic link to the scaffolding perl from the location where the perl binary will be installed. If so, we need to get rid of it before building and installing the final system perl.

Pre-build (as `root`) commands:

```
if test -L /usr/bin/perl; then rm -f /usr/bin/perl; fi
```

The `configure.gnu` script is used again here. The "man" directives are a way of telling Perl to build and install man pages even though `groff` is not yet available, and the "pager" directive tells Perl to use the `less` pager instead of `more` even though, at this point, it's also not available. "useshrplib" tells Perl to build a shared `libperl` library, rather than statically linking its functions into the `perl` binaries.

Configuration commands:

```
./configure.gnu --prefix=/usr -Dvendorprefix=/usr \
  -Dman1dir=/usr/share/man/man1 -Dman3dir=/usr/share/man/man3 \
  -Dpager="/bin/less -isR" -Dusethreads -Duseshrplib
```

Compilation commands:

```
make
```

Also, some of the tests fail, so we once again continue regardless.

Test commands:

```
make test || echo "Exit code $?: continuing anyway"
```

A bunch of scripts that were built and installed before the final system perl is set up have the scaffolding perl path baked into them. Now that we have the real perl available, we can modify those scripts so they know where to find it.

Post-installation (as `root`) commands:

```
pushd /usr/bin
grep -l /scaffolding/bin/perl * | while read FILE; \
  do \
  sed -i -e 's@/scaffolding/bin/perl@/usr/bin/perl@g' $FILE; \
  done
popd
```

Installation commands:

```
make install
```

12.7.23. expat (final-system-components phase)

For an overview of expat, see expat.

The tests don't pass reliably — for Intel-architecture builds, it seems okay, but ARM fails entirely.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

By default, the documentation doesn't get installed.

Installation commands:

```
make install
mkdir -p /usr/share/doc/expat
install -v -m644 doc/* /usr/share/doc/expat
rm -f /usr/share/doc/expat/Makefile*
```

12.7.24. python (final-system-components phase)

For an overview of python, see python.

Dependencies

expat, libressl, readline, libxcrypt.

12.7.24.1. About Python Packages

If you are familiar with other interpreted languages — also called "scripting" languages, like ruby, perl, or javascript — the way that python projects are packaged may seem somewhat confusing.

With perl, for example, each separate project is called a "module"; you download the source code for a module as a compressed tar file, generally from the Comprehensive Perl Archive Network (CPAN), unpack it, and then run a series of commands that are basically similar to the GNU build system `configure; make; make check; make install` sequence: `perl Makefile.PL; make; make test; make install`. (In most cases, perl users use the `cpan` program to do this, because `cpan` will resolve dependencies and install pre-requisite modules that the desired module needs, as well, but you *can* just run those commands yourself.) Other languages have their own common practices — ruby has "rubygem" files, the javascript node.js runtime has the "node package manager" or "npm" — so you have to learn the common practice for those languages when you start to use them, but there is a common practice you can easily learn about.

In the python realm, things are not so straightforward, because the original tools that were created to facilitate packaging and distribution were later deemed problematic or inadequate. As of python

3.12, the landscape is still in flux, so there are a few old common practices that you may encounter, and a few newer practices that you may encounter as well, and what seems like dozens of packages that you need to install in order to get things to work.

If you'd like to know more about the backstory that led to the current situation, you can find articles online; I found the page at https://bbc.github.io/cloudfit-public-docs/packaging/this_way_up.html very informative, although it was written in October of 2021 and is therefore somewhat out-of-date.

The best thing about the python packaging situation, from my perspective, is that one format that appears to be available for all python projects is the "sdist" or "source distribution" format, a compressed tar file just like the compressed tar files used for C-language packages from the GNU project and elsewhere. This, of course, aligns perfectly with my practice in CBL! Once you unpack a source distribution, though, there are multiple ways you might need to go about building and installing the project.

As of release 3.12, there are two common practices with python projects: some projects still use setuptools (which was part of the python distribution up through release 3.11, but is now deprecated and needs to be installed separately), and others comply with a standard interface called "PEP 517".

Installing packages that use setuptools

For packages that use setuptools, you will find a script `setup.py` in the top directory of the source distribution. For these packages, you can build the package by running `python setup.py build`, and install it with `python setup.py install`. The installation routine checks to make sure that the package's dependencies are also installed, and downloads them if they are not. For any python package installed through CBL blueprints, the blueprint notes what those dependencies are, so the package's dependencies will already be installed and do not need to be downloaded or installed as part of this process.

There's an issue with the package installation routine that makes it tricky to use with the package-users scheme. Suppose you're installing a package called `A`, which depends on a package `B`. Suppose also that `B` installs a script `/usr/bin/b`. When you install `B`, this works as expected: it installs `/usr/bin/b` and all is well. However, when you install `A`, the installation process checks to make sure that `B` is available, and then — for some unfathomable reason! — tries to install exactly the same script `/usr/bin/b`. Since the package user for `A` is not allowed to tamper with files owned by `B`, the installation process fails with an error at that point.

Until I think of a better solution, what I do in this case is simply move `/usr/bin/b` out of the way before installing `A`, and then move it back (overwriting the one installed by `A`) afterward. Not difficult, but definitely irritating.

Installing PEP 517 packages

THe PEP 517 standard defines two kinds of programs that are used to build and install python packages: "build frontends" and "build backends". There are a plethora of programs that have been implemented to serve these roles, and python projects that use PEP 517 can use whichever programs the project authors prefer. The source dsitribution for these projects contain a file `pyproject.toml` in their top directory; among other things, that file is expected to contain

information about the specific build programs that should be used to build the package. A package called build can be used as a generic build tool for these projects; to use it, you run `python -m build --wheel` to construct a binary package called a "wheel" (conventionally with the file extension `.whl`), and then install the wheel with the `pip` program that is provided as part of the standard python distribution. I often find that the build stage requires me to add the `--no-isolation` flag so the build is done in the context of the standard system python, so that the build system does not try to download and install the package's dependencies unnecessarily.

A Few More Words About `pip` And Virtual Environments

The `pip` program mentioned earlier is the most common way to install python packages; it is like the `gem` program packaged with ruby, or the `cpan` program that comes with perl. `pip` allows packages and their dependencies to be installed from some upstream source where they're available in a pre-built binary form — in the case of python, these are "wheel" files, which have replaced an older and obsolete "egg" format. That's not the practice I use in CBL, but if you want to manage python packages on your system using `pip` I won't call you a bad person. If you decide to use `pip` to install packages for use with the system python installation, you should probably create a single package user (perhaps `python-packages`) that will own them all, or just install them all as the `python` package user.

Another alternative is to set up a python *virtual environment* (aka "venv") for each specific purpose. A venv looks like a full installation of python, with its own set of packages distinct from the packages installed in the system python, but actually only has symbolic links to the real system python instead of having a full copy of everything. To set up a venv, you just have to identify a directory where it will live, `$HOME/my-venv` for example, and run `python -m venv $HOME/my-venv`. Then you can *activate* the venv by sourcing `$HOME/my-venv/bin/activate` — this changes the prompt to indicate what venv you're using, and manipulates a few environment variables so that when you subsequently run `pip` or `python` or whatever, the one that is found is the one in the venv. Any packages you install while a venv is active will be installed into that venv directory structure.

To stop using a venv, you just run `deactivate`; this is a shell function, defined by `activate`, that puts the environment back the way it was originally.

When installing packages, if you don't want to rely on the canonical package repository located at `pypi.org` (the "Python Package Index"), you can set up a local python package repository and use `pip` to install packages from it. A helpful tool for doing this is the `pip2pi` package. I have not done this, though, so I can't be more explicit!

12.7.24.2. The System Python Installation

Now that we've discussed how to use the python installation, let's get it built and installed!

For our system Python build, we specify a few additional configure options, including an "ensurepip" directive that causes the `pip` package-management program to be installed.

There is a configuration setting `--enable-optimizations` that runs a suite of tests, and then uses profiling data collected during the test run to improve the performance of the python installation itself. This is desirable, but the tests hang in the partially-built CBL environment. You may want to re-build python, with the optimization setting enabled, once the final system is complete; this is done in Rebuild the Packages Whose Tests Have Not Been Run, so if you decide to run that, you'll

wind up with an optimized Python as a consequence. If you don't want to rebuild all the packages whose test suites are skipped during the basic CBL process, you can just rebuild python with optimizations with `lb python::rebuild-untested-packages`.

Configuration commands:

```
./configure --prefix=/usr --with-system-expat --enable-shared \
  --with-lto --with-ensurepip=install --with-openssl=/usr
```

Compilation commands:

```
make
```

There is a `make test` Makefile target that runs an extensive suite of automated tests. Like the tests that run as a result of the optimization setting, these tests hang indefinitely in the initial partial CBL environment, so we skip them.

Test commands:

```
(none)
```

When installing multiple versions of python into the same prefix (in our case, `/usr`), it's recommended to designate one of them the *primary* version and install it using `make install`. Other versions can be installed with `make altinstall`, so that they're installed as e.g. `python2` and `python2.7` but not as `python`. Since CBL prefers for the latest stable version of a package to be primary, python 3 is installed that way.

Installation commands:

```
make install
```

Since this will be the primary system python, it should be available using the program name `python` as well as `python3`. The same applies to some other programs.

Installation commands:

```
ln -sf python3 /usr/bin/python
ln -sf pip3 /usr/bin/pip
ln -sf idle3 /usr/bin/idle
```

If you will be installing python packages — which is certainly the case in the CBL process — you'll want the site-packages directory to be set up as a package-users installation directory.

Post-installation (as `root`) commands:

```
tmpdir=$(mktemp -d)
echo /usr/lib/python3*/site-packages >> $tmpdir/sitedir
echo /usr/lib/python3*/site-packages/ '*' | tr -d ' ' >> $tmpdir/sitedir
```

```
cat /etc/pkgusr/install_dirs $tmpdir/sitedir > $tmpdir/instdirs
sort < $tmpdir/instdirs | uniq > /etc/pkgusr/install_dirs
rm -rf $tmpdir
```

Similarly, when python packages are installed, a reference to them is (at least sometimes) recorded in a file called `easy-install.pth`. To permit this, we can put it into the install group and make it group-writable.

Installation commands:

```
pushd /usr/lib/python3*/site-packages
touch easy-install.pth
chown python:install easy-install.pth
chmod g+w easy-install.pth
popd
```

12.7.25. eudev

Name	Userspace Device-File Daemon
Version	3.2.14
Project URL	https://github.com/eudev-project/eudev
SCM URL	https://github.com/eudev-project/eudev
Download URL	https://github.com/eudev-project/eudev/releases
Dependencies	gperf, perl, python

UNIX systems treat almost everything as a file, including I/O devices like hard disks, optical drives, mouses, scanners, and so on. The files that represent these devices are called "special" files (or sometimes "device" files or simply "nodes"); instead of containing streams of data (like normal files), or a structured set of names mapped to inodes (like directories), special files are associated with device drivers in the operating system kernel. Conventionally, these special files are found in the top-level `/dev` directory.

Historically, system administrators were expected to create special files corresponding to all the hardware devices available on a system. These days, that's a somewhat unwieldy expectation — now that computer systems have USB and firewire and other hot-pluggable devices, the kinds of hardware devices that might be available on a system can change from minute to minute. Also, individual devices are defined by a *major* number that indicates which device driver is used to access that device and a *minor* number that indicates specifically which device should be accessed; the minor number is not always predictable and might be different each time the system is booted. So there's no good way to pre-create special files for all the hardware devices that might ever be attached to a system.

Originally, the `udev` program was used to manage these device files: when hardware was detected, the kernel would notify `udev` (or some other handler program) by sending event notifications called "uevents" over a netlink socket; whatever handler program was listening to that socket would then create appropriate nodes so the hardware could be used by userspace programs.

This is no longer the case, so `udev` is not nearly as important as it used to be! In modern Linux kernels, the kernel itself maintains a temporary filesystem, the `devtmpfs`, and automatically creates device files within it as hardware is detected and removes them if hardware is disconnected. But `udev` is still helpful: it can set ownership and access permissions as specified by system policy and encoded in configuration files, and creates or removes symbolic links to device files so they can be accessed using names or paths that might be more convenient than the ones assigned by the kernel. `udev` can also run arbitrary commands or scripts when specific hardware devices are connected or removed, which can be handy!

In 2012, the udev code was assimilated into systemd; since CBL doesn't use systemd, and the systemd project team doesn't support the use of udev *without* systemd, CBL can't use udev either. People at the Gentoo project forked udev into the new project "eudev" so that they would have a systemd-independent device management system. In 2021, Gentoo decided to abandon eudev, so it was migrated to a new project outside of Gentoo by people who were interested in continuing to use and maintain it.

Little Blue Linux uses eudev to manage special files. (There are other alternatives, as well, which could be used: busybox contains a program `mdev` that serves the same purpose as `udev`, and there is a variant of `mdev` that runs as a daemon and is available from skarnet.org.)

A test script for eudev is written in perl and has the path `/usr/bin/perl` hard-coded, so we might as well force the final system perl to be built before eudev. As of eudev 3.2.10, the same is true of python — maybe replacing the perl script? I haven't checked.

Dependencies

perl, python.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc --enable-hwdb
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
install -dv /lib/firmware
```

One of the policy decisions made by systemd — and imported as the default policy of `eudev` as well — is that network interfaces should not use the default names defined by the kernel, like `wlan0` or `eth0`, and should instead use names that are thought to be more stable and predictable by the

systemd developers. I don't care for this behavior, so I disable it by overriding the built-in "net-name-slot" rule using an empty file.

Installation commands:

```
touch /etc/udev/rules.d/80-net-name-slot.rules
```

`eudev` uses a hardware database that is compiled from files in its `hwdb.d` directories and should be rebuilt when those files change. We should create the initial version of that database now. package

Installation commands:

```
udevadm hwdb --update
```

The `udevd` program should be run as a daemon process — it listens on the netlink socket mentioned earlier for uevent messages from the kernel and responds to them in accordance with system policy. So we set up an s6-rc service pipeline for it.

The `--debug` option to `udevd` causes the daemon to be *quite* chatty: it produces hundreds of kilobytes of log messages every time the system boots. You may wish to remove that flag until or unless you are trying to figure out why `udevd` is doing (or not doing) something.

Service Pipeline: `udevd` *(in bundle* `rl-default`*)*

Service 1: Longrun `udevd-svc`	
Dependencies	• `setup-dev` • `mount-proc` • `mount-sys`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`export UDEV_LOG err`
	`fdmove -c 2 1`
	`/usr/sbin/udevd --debug`

Service 2: Longrun `udevd-log`	
notification-fd	`3`
Dependencies	• `remount-root-rw`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`s6-log -d3 T s1000000 n10 /var/log/udevd`

Post-installation (as `root`*) commands:*

```
mkdir -p /var/log/udevd
```

The eudev package also provides a program, `udevadm`, that can be used to query or manipulate the device management system. After setting up the daemon, we can use `udevadm` to ask the kernel to emit uevent messages for all the hardware it has already found; that way, the local system policy will be applied to everything that was pre-populated in `/dev` by the kernel.

Since this one-shot service handles hardware that was connected while the computer was turned off, I'm calling it the "coldplug" service — to differentiate it from the services that handle hot-plugged hardware devices.

Service Directory: Oneshot `coldplug` (in bundle `rl-default`)

Dependencies	• `udevd`

Up script	`if { s6-echo "Performing coldplugging" }`
	`if { udevadm trigger --action=change --type=subsystems }`
	`if { udevadm trigger --action=change --type=devices }`
	`if { udevadm settle }`

It appears there are some udev rules in the default distribution that assume there is a `kvm` group, so let's create one.

Post-installation (as `root`) commands:

```
grep -q ^kvm: /etc/group || groupadd -r kvm
```

12.7.26. gdbm

Name	GNU dbm
Version	1.23
Project URL	http://www.gnu.org/software/gdbm
SCM URL	*(unknown)*
Download URL	https://www.gnu.org.ua/software/gdbm/download.html

Historical UNIX systems include a simple key/value database system called `dbm` (for "database manager"). GDBM is the GNU implementation of `dbm`, and supports the APIs of the original `dbm` as well as Berkeley's extended `ndbm` version, which supports having multiple databases open at the same time.

Configuration commands:

```
./configure --prefix=/usr --enable-libgdbm-compat
```

Compilation commands:

```
make
```

By default, the `dbm`-compatible APIs are not built. Some packages may want to use them, though, so we override that option.

Test commands:

```
make check || echo "Exit code $?: continuing anyway"
```

With the current version of tcl or expect or something, one of the GDBM tests crashes.

Installation commands:

```
make install
```

12.7.27. libgpg-error

Name	GnuPG Runtime Library
Version	1.47
Project URL	https://gnupg.org/software/libgpg-error/index.html
SCM URL	*(unknown)*
Download URL	https://gnupg.org/download/index.html#libgpg-error

Originally, libgpg-error was just a library that defined common error values for the various components that make up GnuPG. It has turned into more of a runtime library, including a stream library, mutexes, a base64 decoder, and other miscellaneous logic used throughout the various pieces of GnuPG.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.28. libgcrypt

Name	GNU cryptographic library
Version	1.10.2
Project URL	https://gnupg.org/software/libgcrypt/index.html
SCM URL	*(unknown)*
Download URL	https://gnupg.org/download/index.html#libgcrypt
Dependencies	libgpg-error

Libgcrypt is a general purpose cryptographic library. It was originally part of GnuPG, but was factored out into a standalone project since it can be used by any program that needs cryptographic building blocks.

Libgcrypt can optionally make use of Linux capabilities — specifically, to lock memory pages so they cannot be written to swap. This is important for cryptographic libraries, because if a cryptographic key is written to persistent storage it might be recoverable later on.

Since CBL has support for capabilities, we can enable this.

Configuration commands:

```
./configure --prefix=/usr --with-capabilities
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.29. libksba

Name	GNU Libksba library
Version	1.6.4
Project URL	https://gnupg.org/software/libksba/index.html
SCM URL	*(unknown)*
Download URL	https://gnupg.org/download/index.html#libksba

Libksba is a library of functions that simplify working with X.509 certificates, such as are used in TLS certificates.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.30. libassuan

Name	GNU Libassuan
Version	2.5.6
Project URL	https://gnupg.org/software/libassuan/index.html
SCM URL	*(unknown)*
Download URL	https://gnupg.org/download/index.html#libassuan
Dependencies	libgpg-error

Libassuan implements a protocol used for inter-process communication among many of the components of GnuPG. It was designed to prevent (potentially buggy) programs from compromising secret data; by moving the actual cryptographic work into a server program, and having the program that makes use of the result of that cryptographic work — such as an email client — be a separate client program, it's much easier to demonstrate that there's no way that the client program can leak secret key information.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.31. npth

Name	New GNU portable threads library
Version	1.6
Project URL	https://www.gnupg.org/software/npth/index.html
SCM URL	*(unknown)*
Download URL	*(unknown)*

Pth is a portable threads library; it provides cooperative priority-based scheduling for multiple threads within a program.

nPth is a new implementation of the same Pth API, based on the system's standard threads implementation, that was written as part of the GNU Privacy Guard (gpg) project and is used extensively within modern versions of gpg (version 2.0 and higher).

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.32. pinentry

Name	GNU PIN entry
Version	1.2.1

Project URL	https://gnupg.org/download/index.html
SCM URL	*(unknown)*
Download URL	*(unknown)*

This is a collection of password entry dialog programs, used by GnuPG to obtain passphrase information from users. The simplest of these, a simple TTY version, has no dependencies at all. More commonly, a Curses version (in CBL, using the ncurses library) or GUI version will be used instead.

Only the programs that work with libraries available at configuration time are built. You may want to rebuild and re-install this package if you install one of the GUI libraries for which a pinentry program is provided (GTK, GNOME, or Qt).

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.33. gnupg

Name	The GNU Privacy Guard
Version	2.4.3
Project URL	https://gnupg.org/
SCM URL	*(unknown)*
Download URL	https://www.gnupg.org/download/index.html
Dependencies	libgpg-error, libgcrypt, libksba, libassuan, npth, pinentry
Environment	• `CFLAGS`: `$CFLAGS -fcommon`

The GNU Privacy Guard is an implementation of the OpenPGP standard (defined in RFC4880). It is a

general-purpose public key cryptography system, allowing you to encrypt files such that only a specific set of people can decrypt them, sign files such that anyone with your public key can ensure that only you could have signed them, and stuff like that.

In CBL, GnuPG is considered a core part of the system because it allows you to verify that the source code of many packages is exactly what the package maintainers want it to be. It's very common for "signature" files to be distributed along with package source code files; with GnuPG, you can verify that the package source code has not been modified or corrupted by anyone.

Environment variable: CFLAGS

```
$CFLAGS -fcommon
```

The behavior of GCC has changed in GCC 10 such that "common" sections are no longer used by default to permit duplicate object definitions. This causes build issues in GnuPG, so we override that change to get the old default behavior back.

A number of the tests in `t-gettime.c` fail on the partial CBL system.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.34. groff

Name	GNU troff
Version	1.23.0
Project URL	https://www.gnu.org/software/groff/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/groff/

Groff is the GNU implementation of `troff`, a component of a document processing system originally developed as part of Unix. It's fairly flexible and powerful, and can be used to produce sophisticated documents in a variety of output forms.

In CBL, groff is used primarily to generate man pages. For other sophisticated typesetting, I generally use TeX or LaTeX.

When configuring groff, `PAGE` should be set to the default paper size. Typically this will be `letter` or `a4`.

Configuration commands:

```
PAGE=letter ./configure --prefix=/usr
```

Compilation commands:

```
make
```

There is no automated test suite for groff.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.35. gawk (final-system-components phase)

For an overview of gawk, see gawk.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc
```

Compilation commands:

```
make
```

The test suite for gawk hangs forever, possibly because of some change in glibc 2.27 (prior to which there was a test failure but the suite completed without other incident) or possibly because of some issue in the scaffolding environment. Inspecting the situation, the hanging command is `tr[A-Z][a-z]`; I have no idea what the problem is.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

Some of the documentation in the `doc` directory doesn't get installed by the normal installation routine, but can be helpful if you are trying to understand or use gawk.

Installation commands:

```
mkdir /usr/share/doc/gawk
cp doc/{awkforai.txt,*.{eps,pdf,jpg}} /usr/share/doc/gawk
```

12.7.36. iana-etc

Name	IANA /etc files
Version	20221202a
Project URL	https://www.iana.org/protocols
SCM URL	http://git.freesa.org/freesa/iana-etc.git
Download URL	https://files.freesa.org/
Dependencies	gawk

The Internet Assigned Numbers Authority (IANA) mandates the use of standard numbers for network protocols and services. They're the ones who say that HTTP should use TCP port 80, and HTTPS should use TCP port 443, for example.

Reference files with the number assignments are provided by IANA. These are distributed as XML files; what we need are subsets of the data found in the XML files in a specific format.

The package available on the FreeSA repository contains a script that fetches the current version of those files; it also contains a copy of those files, updated reasonably often; the version number of the package corresponds to the date on which their content was most recently updated. It also has scripts, copied from the Arch Linux PKGBUILD file for *their* version of this package, that produce the `/etc/protocols` and `/etc/services` files we need from those canonical XML files.[2] This blueprint just runs those scripts.

Configuration commands:

```
(none)
```

Compilation commands:

```
./bin/gen-protocols protocol-numbers.xml protocols
./bin/gen-services service-names-port-numbers.xml services
```

Test commands:

```
(none)
```

Installation commands:

```
cp -a -v protocols /etc
cp -a -v services /etc
mkdir -p /usr/share/iana-etc
cp -a -v *.xml /usr/share/iana-etc
```

12.7.37. inetutils

Name	GNU network utilities
Version	2.4
Project URL	https://www.gnu.org/software/inetutils/
SCM URL	https://git.savannah.gnu.org/git/inetutils.git
Download URL	https://ftp.gnu.org/gnu/inetutils/
Dependencies	libcap

This package contains a variety of small Internet Protocol utility programs: `ping`, `tftpd`, `tracepath`, things like that.

Several of the programs in this package — all of the `r` client commands (`rcp`, `rlogin`, `rsh`, possibly others), `ping`, `ping6`, and `traceroute` — require elevated privileges to work. If you want to use them, you can run them as root, install them setuid to root, or grant them *capabilities* as provided by libcap. I prefer the latter approach, so that's how things are set up here.

Dependencies

libcap.

usable by non-root users, you need to make them setuid to root, or use capabilities. This package comes with a script, `setcap-setuid.sh`, that appears to be intended to set binaries from this package up that way. I haven't investigated it to see how exactly it works, or incorporate it into this blueprint, though.

There are a couple of file conflicts with this package — it provides a `hostname` program, which `coreutils` also provides, and a `logger` program that is also found in `util-linux`. We will disable the versions available in this package, because the other ones are already part of LB Linux — if you have some reason to prefer the versions found here, obviously you should disable the other ones and enable these!

Configuration commands:

```
./configure --prefix=/usr --disable-hostname --disable-logger
```

Compilation commands:

```
make
```

A lot of the tests fail during the CBL process — unsurprisingly, since the network is not available during that build and these are network utilities. We can just skip the tests for now.

Test commands:

```
(none)
```

I don't use the `r` client programs, so I don't set any capabilities on them. If you want to use them without `sudo`, you can give them `CAP_NET_BIND_SERVICE` (according to the `README` in this package). I *do* use `ping` and `traceroute`, though, so I give them the privileges needed to use them as a non-root user.

I *believe* that the `setcap` invocation here means "make the CAP_NET_RAW capabiilty (which allows programs to open sockets in "raw" mode) part of the permitted and effective permission set for this program." I haven't read all the relevant documentation, though, so all I'm sure of is that it seems to work.

after-install-as-root: - setcap cap_net_raw+pe /usr/bin/ping - setcap cap_net_raw+pe /usr/bin/ping6 - setcap cap_net_raw+pe /usr/bin/traceroute

Installation commands:

```
make install
```

12.7.38. xz (final-system-components phase)

For an overview of xz, see xz.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.39. grep (final-system-components phase)

For an overview of grep, see grep.

After upgrading to glibc 2.28, the automated test suite for grep started crashing with one unexpectedly-passing test. This might be because some long-standing conformance bugs were fixed in 2.28. It's a good idea to review the test logs to see if they pass clean after upgrading this package or glibc again.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.40. libtool

Name	GNU libtool
Version	2.4.7
Project URL	https://www.gnu.org/software/libtool/
SCM URL	*(unknown)*
Download URL	http://mirrors.kernel.org/gnu/libtool/
Dependencies	grep

Libtool is a part of the GNU build system, along with autoconf and automake; you can read a bit more about it, and find links to resources to learn more, in the section about Autoconf. It lets you use a common set of commands to build and use static and shared libraries across a wide variety of operating systems.

Dependencies

grep.

The path to `grep` is built into the `libtool` and `libtoolize` scripts produced by this package. We need to make sure the final system `grep` is installed already; otherwise, these scripts will find the scaffolding version of `grep`.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check || echo "Exit code $?: continuing anyway"
```

Some of the libtool tests (all pertaining to the `libltdl` library, which provides a common API for dynamically opening shared libraries at runtime) fail. As per usual, we presume that everything is really okay and proceed; check the test logs manually if you're concerned.

Installation commands:

```
make install
```

12.7.41. libxml2

Name	GNOME XML library and toolkit
Version	2.9.12
Project URL	http://xmlsoft.org/
SCM URL	https://gitlab.gnome.org/GNOME/libxml2.git
Download URL	ftp://xmlsoft.org/libxml2/
Dependencies	autoconf, libtool, pkgconf, python

libxml2 is an XML (Extensible Markup Language) parser and toolkit that was originally written for the GNOME project.

As distributed, the libxml2 source does not include a `configure` script, so that has to be generated first.

Configuration commands:

```
autoreconf -i
```

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

The test suite currently requires a separate package. According to the `make check` target, instructions for setting this up can be found at http://www.w3.org/XML/Test/ if you are interested in doing this; I'm not interested enough in running these tests to look into it.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

XML always canonically identifies stylesheets and schemas and things like that using URIs, so it's common for XML documents to include references to stylesheets at web locations like `http://docbook.sourceforge.net/release/xsl-ns/current/manpages/docbook.xsl`. Programs that handle XML documents, like the programs provided in this package and libxslt, will automaticaly fetch files from those canonical locations whenever they find such references. Of course, it's not necessary to fetch files from the Internet if they are already present locally; with this in mind, these programs will also make use of an XML "catalog" file — conventionally located at `/etc/xml/catalog` — if there is one. XML catalog files can contain entries that map Internet locations to local filesystem paths, among other things, so unnecessary Internet lookups can be eliminated.

One of the programs provided by libxml2 is `xmlcatalog`, which can be used to administer XML catalog files. We'll use it to create a catalog file at the conventional location.

Installation commands:

```
mkdir -p /etc/xml
if [ ! -f /etc/xml/catalog ]; \
  then \
  xmlcatalog --noout --create /etc/xml/catalog; \
  fi
```

Since other packages will need to update the catalog file to add references to artifacts they've installed, we make the xmlcatalog program setuid, and put it in the install group — that way, any package user will be able to run `xmlcatalog` to update the main XML catalog file.

Post-installation (as `root`) commands:

```
chgrp install /usr/bin/xmlcatalog
chmod 4750 /usr/bin/xmlcatalog
```

The catalog file is a configuration file, of a sort, so it's nice to track changes to it over time.

Configuration Files

- `/etc/xml/catalog`

12.7.42. libxslt

Name	GNOME XSLT library
Version	1.1.34
Project URL	http://xmlsoft.org/xslt/
SCM URL	https://gitlab.gnome.org/GNOME/libxslt.git
Download URL	ftp://xmlsoft.org/libxslt/
Patches	• `libxslt-1.1.34-fix-tests-1.patch`
Dependencies	autoconf, pkgconf, libxml2

libxslt is an XSLT (Extensible Stylesheet Language Transformations) library that was originally written for the GNOME project. As a library, it provides functionality that lets programs transform XML documents into other things using XSL stylesheets; it also includes a program `xsltproc` that lets you use that functionality from the command line.

The latest release of libxslt has a couple of test failures when using the latest stable libxml2 release, but this issue can be corrected by pulling in some fixes from libxslt mainline development.

Patch:

- libxslt-1.1.34-fix-tests-1.patch

As with libxml2, the source distribution for libxslt does not have a `configure` script, so it must be generated.

Configuration commands:

```
autoreconf -i
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.43. docbook-xsl-nons

Name	DocBook XSL Stylesheets no-namespace
Version	snapshot-2020-06-03
Project URL	https://github.com/docbook/xslt10-stylesheets
SCM URL	https://github.com/docbook/xslt10-stylesheets
Download URL	https://github.com/docbook/xslt10-stylesheets/releases
Dependencies	libxml2, libxslt

The primary blueprint for this package is docbook-xsl. This is an alternate version of the DocBook XSL stylesheets for use with DocBook versions prior to 5, when the DocBook namespace prefix was added.

As with the namespaced version, since we're not building the stylesheets, there's no configuration, compilation, or testing steps.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

The installation for this package is much like the namespaced version; we're simply copying to a different location, and adding a different set of catalog entries.

Installation commands:

```
mkdir -p /usr/share/xml/docbook-xsl-nons-${version}
cp -v -R * /usr/share/xml/docbook-xsl-nons-${version}
xmlcatalog --noout --add "rewriteSystem" \
  "http://docbook.sourceforge.net/release/xsl/${version}" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
  "http://docbook.sourceforge.net/release/xsl/${version}" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteSystem" \
```

```
  "http://docbook.sourceforge.net/release/xsl/current" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
  "http://docbook.sourceforge.net/release/xsl/current" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteSystem" \
  "http://cdn.docbook.org/release/xsl-nons/${version}" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
  "http://cdn.docbook.org/release/xsl-nons/${version}" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteSystem" \
  "http://cdn.docbook.org/release/xsl-nons/current" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
  "http://cdn.docbook.org/release/xsl-nons/current" \
  "/usr/share/xml/docbook-xsl-nons-${version}" /etc/xml/catalog
```

12.7.44. docbook4-xml-dtd

Name	DocBook 4 XML DTD
Version	4.5
Project URL	https://docbook.org/
SCM URL	*(unknown)*
Download URL	https://docbook.org/schemas/4x
Dependencies	libxml2

DocBook is an XML markup language for technical documentation. It lets you create document content in a presentation-neutral form; you can then use programs to transform that content into a variety of formats. It's formally defined by a "schema".

There are a bunch of ways that an XML schema can be presented; one of those, "RELAX NG" (a quasi-acronym for "REgular LAnguage for XML, Next Generation"), is the primary version for modern versions of DocBook. Others are also provided by the OASIS Technical Committee that maintains DocBook, including "W3C XML Schema" and "XML DTD" (which stands for "Document Type Definition).

The current version of DocBook is 5.1, and you can read all about it at `http://docbook.org/` — including, as of May 2017, an ebook that can be downloaded from `http://tdg.docbook.org/`. However, CBL requires a previous version of DocBook for building the documentation for the kmod package. This blueprint is specifically for version 4.5 of the XML DTD variant of DocBook; kmod's documentation is written using the even-more-obsolete version 4.2, but the point-releases are compatible.

The distribution package for this package is a zip file, `docbook-xml-4.5.zip` — if you don't use the package from the CBL repository, you'll need to convert it to the standard packaging structure used by litbuild.

As with other non-program XML artifacts, the DocBook XML DTD consists of files that should be copied to a location on the filesystem, with that location referenced by the main XML catalog file. There's no configuration, build, or test stage for this package.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
mkdir -p /usr/share/xml/docbook-xml-dtd-4.5
cp -v -af docbook.cat *.dtd *.mod ent \
  /usr/share/xml/docbook-xml-dtd-4.5
```

The way we set up the XML catalog for the XML DTD is: we create a separate `docbook` catalog file that references the files from this package, and then we add entries to the main `catalog` file to delegate to the `docbook` catalog file.

It's possible that other packages will have documentation that specifies it needs some other 4.x version of the DocBook XML DTD (4.1, 4.2, 4.3, or 4.4). This version of the DTD will work for any of those packages; if necessary, you can add additional entries to the `docbook` and main XML `catalog` files, specifying that the files for those versions are in the directory where the 4.5 DTD files reside, or you can patch the client package so that the documentation requests the 4.5 DTD.

Installation commands:

```
if [ ! -e /etc/xml/docbook ]; \
  then \
  xmlcatalog --noout --create /etc/xml/docbook; \
  fi
xmlcatalog --noout --add "public" \
  "-//OASIS//DTD DocBook XML V4.5//EN" \
  "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd" /etc/xml/docbook
xmlcatalog --noout --add "public" \
  "-//OASIS//DTD DocBook XML CALS Table Model V4.5//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/calstblx.dtd" /etc/xml/docbook
xmlcatalog --noout --add "public" \
  "-//OASIS//DTD XML Exchange Table Model 19990315//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/soextblx.dtd" /etc/xml/docbook
xmlcatalog --noout --add "public" \
```

```
  "-//OASIS//ELEMENTS DocBook XML Information Pool V4.5//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/dbpoolx.mod" /etc/xml/docbook
xmlcatalog --noout --add "public" \
  "-//OASIS//ELEMENTS DocBook XML Document Hierarchy V4.5//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/dbhierx.mod" /etc/xml/docbook
xmlcatalog --noout --add "public" \
  "-//OASIS//ELEMENTS DocBook XML HTML Tables V4.5//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/htmltblx.mod" /etc/xml/docbook
xmlcatalog --noout --add "public" \
  "-//OASIS//ENTITIES DocBook XML Notations V4.5//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/dbnotnx.mod" /etc/xml/docbook
xmlcatalog --noout --add "public" \
  "-//OASIS//ENTITIES DocBook XML Character Entities V4.5//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/dbcentx.mod" /etc/xml/docbook
xmlcatalog --noout --add "public" \
  "-//OASIS//ENTITIES DocBook XML Additional General Entities V4.5//EN" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5/dbgenent.mod" /etc/xml/docbook
xmlcatalog --noout --add "rewriteSystem" \
  "http://www.oasis-open.org/docbook/xml/4.5" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5" /etc/xml/docbook
xmlcatalog --noout --add "rewriteURI" \
  "http://www.oasis-open.org/docbook/xml/4.5" \
  "file:///usr/share/xml/docbook-xml-dtd-4.5" /etc/xml/docbook
xmlcatalog --noout --add "delegatePublic" "-//OASIS//ENTITIES DocBook XML" \
  "file:///etc/xml/docbook" /etc/xml/catalog
xmlcatalog --noout --add "delegatePublic" "-//OASIS//DTD DocBook XML" \
  "file:///etc/xml/docbook" /etc/xml/catalog
xmlcatalog --noout --add "delegateSystem" \
  "http://www.oasis-open.org/docbook/" \
  "file:///etc/xml/docbook" /etc/xml/catalog
xmlcatalog --noout --add "delegateURI" \
  "http://www.oasis-open.org/docbook/" \
  "file:///etc/xml/docbook" /etc/xml/catalog
```

Configuration Files

- `/etc/xml/docbook`

12.7.45. kmod

Name	kmod
Version	31
Project URL	http://git.kernel.org/cgit/utils/kernel/kmod/kmod.git
SCM URL	*(unknown)*
Download URL	*(unknown)*
Dependencies	pkgconf, xz, zlib, libxslt, docbook-xsl-nons, docbook4-xml-dtd

Linux is a monolithic kernel, but that doesn't mean that the whole kernel is all in one big file.

Almost all parts of the kernel can be built into small modules that can then be loaded into the kernel at runtime. If you know you're always going to want some kernel feature, you can build it into the basic kernel image, but if you might not ever wind up using a kernel feature, or expect to need it only occasionally, you can build it as a module instead, and only load it if you need it for something.

This is handy if you've got some hardware devices that you don't always use — a USB scanner or camera, for example — and you don't want to waste memory on the device drivers for that hardware unless you actually plug it in.

It's also handy for binary GNU/Linux distributions, where the expectation is that users of the system won't be building their own kernels — those distributions can build almost everything as modules, and then use an "early userspace" facility to figure out exactly what modules need to be loaded to get the system working. That complicates the system significantly: early userspaces in Linux depend on an initial ramdisk or ramfs that is loaded by the boot loader along with the Linux kernel, and it has to contain enough initalization logic to figure out what modules need to be loaded, and actually load them, and then mount the actual root filesystem and pivot_root into it and exec the real init process… really, there's a lot going on there! In CBL, we don't need any of that complexity because we *always* build a kernel from source; we just need to make sure that all the kernel features necessary to get the system running are compiled directly into the kernel image.

Still, pretty much every kernel configuration includes at least *some* modules, so we need to be able to load those modules once we have the system booted. That's what the `kmod` package provides: userspace programs that manipulate kernel modules.

Dependencies

docbook-xsl-nons, docbook4-xml-dtd.

The documentation for kmod is written in DocBook XML, using the (very obsolete) DocBook 4.2 release. It's not difficult to provide versions of the XML DTD and XSL stylesheets that kmod currently expects, so that's what we're doing here.

At configuration time, kmod has to be told to enable support for compressed kernel modules, so we do that.

Configuration commands:

```
./autogen.sh
./configure --prefix=/usr --sysconfdir=/etc --with-zlib --with-xz
```

Compilation commands:

```
make
```

The test suite for kmod assumes that a kernel and modules have already been built and installed in the conventional location, which is not the case for CBL — in particular, it expects a `build` script to be present at `/lib/modules/$(uname -r)/build`, but that's a symbolic link to the original host-system path where the kernel source existed. So at this point it's necessary to defer the test suite until later, or skip it entirely.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

Starting with Linux 4.18, the kernel's module installation routine insists on looking for `kmod` programs in the `/sbin` directory, rather than `/usr/bin`. Let's ensure it is visible there.

Note that if `/usr` is a different filesystem than the root filesystem, this command will fail — you will need to make it a symbolic link by adding `-s` to the `ln` command arguments.

Installation commands:

```
ln -f /usr/bin/kmod /sbin/kmod
```

Kmod is designed to be a multi-call binary — there's only one actual program, and its behavior varies depending on what name it's invoked with: for example, when you run it as insmod, it installs a module; when you run it as rmmod, it removes it. For some reason, though, the installation process for kmod does not install links with the expected names, so we have to do that ourselves.

Installation commands:

```
for program in depmod insmod lsmod modinfo modprobe rmmod; \
  do \
  ln -f /usr/bin/kmod /usr/bin/$program; \
  ln -f /usr/bin/kmod /sbin/$program; \
  done
```

12.7.46. less

Name	Less file viewer
Version	643
Project URL	http://www.greenwoodsoftware.com/less/
SCM URL	*(unknown)*
Download URL	http://www.greenwoodsoftware.com/less/
Dependencies	ncurses

Less is a text file viewer. It's often used as a pager at the end of a pipeline — if you run a command that generates thousands of lines of output, you can pipe that output into `less` and it will let you navigate around the output in a variety of helpful ways.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc
```

Compilation commands:

```
make
```

There is no automated test suite for less.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.47. libpipeline

Name	Subprocess pipeline library
Version	1.5.7
Project URL	http://libpipeline.nongnu.org/
SCM URL	*(unknown)*
Download URL	http://download.savannah.nongnu.org/releases/libpipeline/

According to its homepage: "libpipeline is a C library for manipulating pipelines of subprocesses in a flexible and convenient way."

The maintainer of the man-db project discovered that using the basic C library functions like `system` and `popen` to run a collection of subprocesses and link them together into a pipeline (that is, the standard output of each program is fed into the standard input in the next program in the chain) was prone to issues, so he wrote libpipeline to encapsulate all of that stuff.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.48. litbuild

Name	Literate Build tool
Version	1.0.16
Project URL	http://git.freesa.org/freesa/litbuild
SCM URL	http://git.freesa.org/freesa/litbuild
Download URL	https://files.freesa.org/
Dependencies	ruby

Litbuild is the program used to transform the blueprints from Cross-Building Linux into executable scripts and documentation that is intended to be be read.

Like most other ruby programs, litbuild can be installed as a "gem," rather than installed from a source tar file. Here, we are going to install it as a gem using the `gem` command, but we're starting from a source distribution and constructing the gem itself from that source distribution first.

Configuration commands:

```
(none)
```

The rubygems program — which is provided as part of the ruby language distribution — provides a facility to construct a gem from a source package, using a `gemspec` file that tells it what should be packaged that way.

Compilation commands:

```
gem build litbuild.gemspec
```

There are lots of automated tests for litbuild; unfortunately, they can only be run if a bunch of additional ruby packages are installed, and I don't have energy or enthusiasm enough to write blueprints for them at the moment.

Test commands:

```
echo "Skipping tests (would be `rake spec`)"
```

Installation commands:

```
gem install -l litbuild*gem
```

12.7.49. lsof

Name	lsof
Version	4.98.0
Project URL	https://github.com/lsof-org/lsof
SCM URL	https://github.com/lsof-org/lsof.git
Download URL	https://github.com/lsof-org/lsof/releases

`lsof` lists files that are currently opened by Unix processes. (The name stands for "LiSt Open Files".)

By default, `lsof` shows all files that are opened by any process running as any user. That seems like a potential issue, so we'll enable the "security" setting so that only the `root` user is allowed to do so.

Configuration commands:

```
./configure --prefix=/usr --enable-security
```

Compilation commands:

```
make
```

Some of the tests fail for me.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.50. man-db

Name	Manual page viewer
Version	2.12.0
Project URL	http://man-db.nongnu.org/
SCM URL	https://git.savannah.nongnu.org/git/man-db.git
Download URL	http://download.savannah.nongnu.org/releases/man-db/
Dependencies	pkgconf, libpipeline, berkeley-db

man-db is a manual page viewer. It's used (via the `man` command) to read the standard Unix documentation found in manual pages. Unlike the original implementations of `man`, this package uses a Berkeley DB database to speed up the process of finding man pages.

In GNU systems like CBL, the canonical documentation for many programs is in `info` files rather than manual pages, but the manual pages are still a crticial part of the system documentation.

The installation routine for man-db tries to install the `man` program setuid to the user "man". This conflicts with the package-user scheme, so we disable it.

man-db can optionally make use of a large number of external programs for various purposes — web browsers, program source preprocessors, graph preprocessors, and others; run `./configure --help` to see all the `--with` options. If you want man-db to have support for those built in, add the appropriate configure switches. (And, eventually, you'll have to install those programs if they're not part of the CBL build.)

Also by default, man-db sets up a tmpfiles directory entry for use by systemd, which uses the files in that directory to determine what temporary files to clean up, or something. (Because, I guess, any good init system will be in charge of managing temporary files?) CBL does not use systemd, so we skip it.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc --disable-setuid \
  --without-systemdtmpfilesdir
```

Compilation commands:

```
make
```

A number of the automated tests fail. As usual, we proceed with the build and suggest an eventual inspection of the log files and system state.

Test commands:

```
make check || echo "Exit code $?: continuing anyway"
```

TODO migrate man-db.service and man-db.timer to appropriate s6-rc and/or crontab entries, either pre-install or post-install

Installation commands:

```
make install
```

12.7.51. man-pages

Name	Linux Manual Pages collection

Version	6.05
Project URL	https://www.win.tue.nl/~aeb/linux/man
SCM URL	*(unknown)*
Download URL	https://mirrors.edge.kernel.org/pub/linux/docs/man-pages/

Historically, UNIX systems have been documented by online "manual pages," which can be displayed using the `man` program. (You can find out a lot more about the manual pages by running `man man`, once the `man` program is installed.)

Many open-source programs are distributed along with man pages that describe them as part of the source package. However, there is also a distribution of manual pages for Linux — this package — that contains documentation for system calls, library routines, file formats, and things like that. It also contains man pages for programs that are commonly part of GNU/Linux systems but are not distributed with their own man pages as part of the source packages.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Some man pages included in this distribution are also now installed as part of other packages. We can remove the conflicting pages from this package to avoid any problems.

Installation commands:

```
rm -f man5/tzfile.5
rm -f man8/{tzselect,zdump,zic}.8
make install
```

12.7.52. openssh

Name	OpenSSH
Version	9.5p1
Project URL	https://www.openssh.com/
SCM URL	*(unknown)*
Download URL	https://cloudflare.cdn.openbsd.org/pub/OpenBSD/OpenSSH/portable/
Dependencies	libressl, zlib

OpenSSH provides the ability to access remote systems in a secure way, with all traffic between your computer and the remote one encrypted. It's frightfully useful!

We build OpenSSH using LibreSSL as the underlying cryptographic engine.

The OpenSSH daemon runs in a "privilege separation" mode, to make it difficult to exploit any security-critical bugs that may exist in the code. The OpenSSH daemon runs with system privileges, but when a client opens a connection to that daemon, it starts by spawning a process running as a different user with practically no privileges. That unprivileged process then handles all communications over the network to the (potentially malicious) client, making calls back to the privileged parent process using a well-defined interface; if the unprivileged child process is compromised, the damage it can do is strictly limited.

To enable this, we need to add an `sshd` user and group configured the way that OpenSSH expects it to be. This user and group define the unprivileged security context.

Pre-build (as `root`) commands:

```
mkdir -p /var/empty
chown root:sys /var/empty
chmod 755 /var/empty
grep -q ^sshd: /etc/group || groupadd -r sshd
grep -q ^sshd: /etc/passwd || useradd -r -g sshd \
  -c 'sshd privilege separation' -d /var/empty -s /bin/false sshd
```

Configuration commands:

```
./configure --prefix=/usr \
  --with-ssl-dir=/etc/ssl --sysconfdir=/etc/ssh
```

Compilation commands:

```
make
```

Some of the OpenSSH tests apparently assume that the computer is on the network, or otherwise object to the limited CBL userspace.

Test commands:

```
make tests || echo "Exit code $?: continuing anyway"
```

The installation process for OpenSSH generates a set of unique host keys that will identify the server to clients. It also copies a file called `moduli` into the `/etc/ssh` directory; that file contains a bunch of probably-prime numbers used during the key exchange part of the SSH handshake. You can use that `moduli` file without worrying about it — most people do — but if you want to produce one of your own, you can do so; look at the `generate-ssh-moduli` blueprint.

If networking is enabled, the `sshd` daemon should be run. Note that the `run` script for the service

uses an absolute path for the `sshd` program — this is required, starting with OpenSSH 3.9, because `sshd` re-executes itself every time a connection is made. If a non-absolute path is used, `sshd` will complain and won't start.

Service Pipeline: `sshd` *(in bundle* `network-services`*)*

Service 1: Longrun `sshd-svc`	
Dependencies	• `initialize-entropy` • `network-available`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`fdmove -c 2 1`
	`foreground { ssh-keygen -A }`
	`/usr/sbin/sshd -D -e`

Service 2: Longrun `sshd-log`	
notification-fd	`3`
Dependencies	• `remount-root-rw`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`s6-log -d3 T s10000000 n10 /var/log/sshd`

Post-installation (as `root`*) commands:*

```
mkdir -p /var/log/sshd
chown root:root /var/log/sshd
chmod 0755 /var/log/sshd
```

In the servicedir specified above, you might notice a call to `ssh-keygen`. That ensures that the host keys needed by the OpenSSH daemon (to securely and uniquely identify the server to clients) are present. These are generated during the installation process, so the only time that you'll need to generate new host keys is if they've been deleted for some reason, but that can be the case for CBL systems launched in Cloud Computing providers like Amazon Web Services.

Configuration Files

- `/etc/ssh/ssh_config`
- `/etc/ssh/sshd_config`
- `/etc/s6-rc/source/sshd-svc`
- `/etc/s6-rc/source/sshd-log`

Installation commands:

```
make install
```

12.7.53. procps

Name	proc filesystem utilities
Version	4.0.4
Project URL	https://gitlab.com/procps-ng/procps
SCM URL	*(unknown)*
Download URL	https://gitlab.com/procps-ng/procps/tags
Dependencies	libtool, ncurses, pkgconf

GNU/Linux systems have a pseudo-filesystem called `proc`, conventionally mounted at `/proc`. This is called a "pseudo-filesystem" because it's not actually used to store files; rather, it's a view into the state of the computer system exposed by the Linux kernel as a filesystem. The proc filesystem contains a subdirectory for each process currently running on the system, as well as a bunch of files and subdirectories that give a view into other parts of the running system. (For example, if you want to know what CPUs are in the system, you can `cat /proc/cpuinfo`.)

Using the proc filesystem directly is not always the most convenient way to find out what's going on in the system. The procps package contains a bunch of programs that present information from `/proc` in a more helpful and convenient way.

procps provides a version of the `kill` program, which is also provided by util-linux; we're going to prefer the util-linux version. We also specify that `watch` should be built "8 bit clean", which requires ncurses with wide character support. The watch source expects the `ncurses.h` header file to be in a different location when this is enabled, so we have to adjust that expectation.

The procps configuration script also provides an option to select systemd support, so of course we disable it.

Configuration commands:

```
./autogen.sh
./configure --prefix=/usr --disable-kill --without-systemd
```

Compilation commands:

```
make
```

The automated tests fail at this point with an error about insufficient ptys. Just skip them for now.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.54. psmisc

Name	Miscellaneous process utilities
Version	23.6
Project URL	https://gitlab.com/psmisc/psmisc
SCM URL	*(unknown)*
Download URL	https://gitlab.com/psmisc/psmisc/tags
Dependencies	ncurses

The PSmisc package is a collection of small useful utilities that use the proc filesystem to inspect and modify processes:

- `fuser` shows processes that are using files (kind of like `lsof`);
- `killall` lets you send signals to multiple programs based on their name (kind of like `pkill` from procps);
- `prtstat` shows process statistics;
- `pslog` shows the paths to log files that a process has open;
- `pstree` shows process information like `ps`, but showing the process hierarchy explicitly;
- and `peekfd` lets you find out about file descriptors being used by a process.

Configuration commands:

```
./autogen.sh
./configure --prefix=/usr
```

Compilation commands:

```
make
```

The package does not have an automated test suite.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.55. sysfsutils

Name	System Utilities Based on Sysfs
Version	2.1.0

Project URL	http://linux-diag.sourceforge.net/Sysfsutils.html
SCM URL	*(unknown)*
Download URL	https://sourceforge.net/projects/linux-diag/files/sysfsutils/
Patches	• `sysfsutils-2.1.0-update-config-guess-1.patch`

This package contains a library, `libsysfs`, that provides an interface for querying the information about system devices exposed through the `sysfs` filesystem (conventionally mounted at `/sys`); and a program called `systool` that provides a command-line interface to the functionality in `libsysfs`.

`libsysfs` is important primarily because it's a dependency for the `rng-tools`.

Patch:

- sysfsutils-2.1.0-update-config-guess-1.patch

On some servers, the version of `config.guess` distributed with this package is too old to recognize the target triplet. It's easy enough to update it to the version distributed with the modern GCC sources.

Configuration commands:

```
./configure --prefix=/usr --mandir=/usr/share/man
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.56. curl

Name	cURL
Version	8.4.0
Project URL	https://curl.haxx.se/
SCM URL	https://github.com/curl/curl
Download URL	https://curl.haxx.se/download.html
Dependencies	libressl

12.7.56.1. Overview

Curl is a program to transfer data to or from a server using any of a variety of protocols — primarily FTP, SFTP, HTTP, and HTTPS. It's a lot like GNU wget; one important difference is that curl also provides a library, `libcurl`, which can provide similar functionality to other programs that need it.

12.7.56.2. curl (final-system-components phase)

Configuration commands:

```
./configure --prefix=/usr --enable-optimize \
  --with-ca-bundle=/etc/ssl/cert.pem --with-openssl
```

Compilation commands:

```
make
```

Don't be misled by the `--with-openssl` configure directive. It works for BoringSSL and LibreSSL as well as OpenSSL.

Test commands:

```
(none)
```

The automated tests for curl don't work reliably when there is no network available; some tests hang interminably.

Installation commands:

```
make install
```

12.7.57. jansson

Name	Jansson
Version	2.14
Project URL	https://digip.org/jansson/
SCM URL	https://github.com/akheron/jansson
Download URL	https://digip.org/jansson/
Patches	• `jansson-2.14-fix-export-test-1.patch`

Jansson is a C library for manipulating JSON (JavaScript Object Notation) strings. It's used by rng-tools.

The test suite for jansson has an issue when run with a modern version of binutils, because the

output of the `nm` program has changed from what it expected. This is already fixed on the master branch, so we can apply the fix as a patch to this version.

Patch:

- jansson-2.14-fix-export-test-1.patch

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.58. rng-tools

Name	RNG tools
Version	6.16
Project URL	https://github.com/nhorman/rng-tools
SCM URL	https://github.com/nhorman/rng-tools
Download URL	https://github.com/nhorman/rng-tools/releases
Dependencies	sysfsutils, curl (final-system-components phase), jansson
Environment	• `openssl_LIBS`: `$LDFLAGS -lcrypto`

The Linux kernel provides access to random data drawn from an internal entropy pool, populated by random events like the timing of key presses and network packets being received from other systems.

rngd, a daemon program provided in this package, supplements this collection of random data: it monitors other entropy sources — like hardware random number generators found at `/dev/hwrng`, the Trusted Platform Module random number generator found at `/dev/tpm0`, and the RDRAND CPU instruction — and feeds random data from those sources into the entropy pool. This is pretty much always a good idea, if there are any such sources.

Environment variable: openssl_LIBS

```
$LDFLAGS -lcrypto
```

The distribution tarfile does not include the configure script, so we need to create that with the `autogen.sh` script. We also configure without support for PKCS#11, because that adds an additional dependency on the libp11 package. Similarly, the "rtlsdr" feature introduces an unnecessary dependency, so we disable that as well.

Configuration commands:

```
./autogen.sh
./configure --prefix=/usr --without-pkcs11 --without-rtlsdr
```

Compilation commands:

```
make
```

A test sometimes fails in the minimal userspace.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Service Pipeline: `rngd` *(in bundle* `rl-default`*)*

Service 1: Longrun `rngd-svc`	
Dependencies	• `mount-sys`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`fdmove 2 1`
	`/usr/sbin/rngd -f -d`

Service 2: Longrun `rngd-log`	
notification-fd	`3`
Dependencies	• `remount-root-rw`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`s6-log -d3 T s10000000 n10 /var/log/rngd`

Post-installation (as `root`*) commands:*

```
mkdir -p /var/log/rngd
chown root:root /var/log/rngd
chmod 0755 /var/log/rngd
```

Installation commands:

```
make install
```

12.7.59. sudo

Name	su do
Version	1.9.14p3
Project URL	https://www.sudo.ws/
SCM URL	*(unknown)*
Download URL	ftp://ftp.sudo.ws/pub/sudo/
Dependencies	zlib, coreutils

Sudo allows users to run commands as root or as another user. Unlike the `su` program provided in the shadow package, sudo has a focus on configurability and auditability — users or groups can be permitted to run specific commands or any commands, with password required or not, and any command executed via sudo can be logged.

The installation process for sudo always tries to set the UID and GID of files it installs to specific values; by default these are both 0, so the `sudo` program can wind up being setuid to root. This conflicts with the package-users approach. We can get around that easily enough by overridding the UID and GID specified in the top-level `Makefile`. We look those up using the `id` program provided by the GNU coreutils, which makes that a build-time dependency.

Dependencies

coreutils.

Configuration commands:

```
sed -i -e "s@^install_uid =.*@install_uid = $(id -u)@" \
  -e "s@^install_gid =.*@install_gid = $(id -g)@" Makefile.in
```

Sudo can write log messages using the syslog facility or to a log file. We actually want its log messages to be written to an s6-log directory, like other system logs, so we'll write them to a named pipe created at boot time, and set up a supervised s6-log service to grab everything written to that pipe.

Configuration commands:

```
./configure --prefix=/usr --enable-zlib --disable-rpath \
  --with-logging=file --with-env-editor --with-secure-path \
  --with-logpath=/run/fifos/sudo
```

Compilation commands:

```
make
```

There is a failing test in the automated CBL environment — `check_ttyname` encounters unexpected behavior. A something search reveals that this has been seen from time to time by various projects, and it does not seem critically important; as per usual, we allow the build to proceed.

Test commands:

```
make check || echo "exit code $?: continuing anyway"
```

The main configuration file for sudo is `/etc/sudoers`; the installation process crashes if there is no file already present at that path, so we create an empty one.

The default `sudoers` file content is installed as `/etc/sudoers.dist`. In CBL we generally want to start with the default configuration and then make changes based on policy decisions, so we just copy `sudoers.dist` over the empty file we initially created.

Installation commands:

```
touch /etc/sudoers
make install
mv -f /etc/sudoers.dist /etc/sudoers
```

The installation process also insists on having a "run directory," by default `/run/sudo`, and tries to create it if it does not exist. This fails when using the package users scheme because `/run` is not an install directory. We can pre-create the directory as root to work around the issue.

Pre-build (as `root`) commands:

```
mkdir -p /run/sudo
chown sudo:sudo /run/sudo
```

Since `/run` is a temporary filesystem that does not persist across reboots, that directory will vanish when the system is shut down. But `sudo` will create that directory every time it is run, if it doesn't exist, so we don't have to worry about making sure it is re-created as part of system startup.

We do need the `sudo` program to be setuid to root, and it's necessary (or at least desirable) for some other files and directories to be owned by UID 0 as well.

Post-installation (as `root`) commands:

```
chown root /usr/bin/sudo
chmod u+s /usr/bin/sudo
chown root /usr/libexec/sudo/sudoers.so
chown root /etc/sudoers
chown root /etc/sudoers.d
chmod 755 /etc/sudoers.d
```

```
chown -R root /var/db/sudo
chown root:root /etc/sudo.conf
```

Configuration Files

- `/etc/sudoers`
- `/etc/sudoers.d`
- `/etc/sudo.conf`
- `/etc/s6-rc/source/create-sudo-fifo`
- `/etc/s6-rc/source/sudo-log`

The sudo program generally logs via syslog, but can be told to write its logs to a file instead. We don't actually *want* to write to a file, because we want to direct all those messages to an s6-log service; to facilitate this, we can write them to a FIFO in a canonical location, and have the s6-log process read its standard input from that FIFO.

There's a minor infelicity with this approach: since each invocation of `sudo` is a different process and will close the log file when it terminates, the s6-log process will keep finding its standard input has been closed and will shut itself down each time. The service supervisor will therefore need to spawn a new s6-log every time someone uses a sudo command. That's a little unfortunate but not a big deal — essentially, one extra process will be spawned every time `sudo` is used to run a command.

We always have a tmpfs mounted at `/run`, so that's a convenient place to put the sudo log FIFO (along with FIFOs for any other command that wants to write log messages to a file location).

Service Directory: Oneshot `create-sudo-fifo`

Dependencies	• `setup-run`
Up script	`if { s6-echo "Creating FIFO for sudo logs" }` `s6-mkfifo -m 0600 /run/fifos/sudo`

Now we need an s6-log process that reads from the FIFO and writes to a log directory.

Service Directory: Longrun `sudo-log` *(in bundle* `rl-default`*)*

notification-fd	`3`
Dependencies	• `remount-root-rw` • `create-sudo-fifo`

Run script	`#!/bin/execline -P`
	`redirfd -r -nb 0 /run/fifos/sudo`
	`emptyenv`
	`s6-log -d3 T s1000000 n10 /var/log/sudo`

Post-installation (as `root`) commands:

```
mkdir -p /var/log/sudo
```

Starting in version 1.8.29, sudo expects there to be a file `/etc/environment` that can contain environment variable settings. This file is canonically set up by the Pluggable Authentication Module (PAM) package, and used by the `pam_env` module within that package, but since PAM is not built as part of CBL the file does not automatically exist. It's irritating to see a warning message every time you run `sudo`, so we'll ensure there is a file at that location.

Post-installation (as `root`) commands:

```
touch /etc/environment
```

12.7.60. xml-parser

Name	Perl XML-Parser module
Version	2.46
Project URL	https://github.com/toddr/XML-Parser
SCM URL	https://github.com/toddr/XML-Parser.git
Download URL	https://github.com/toddr/XML-Parser/tags
Dependencies	expat, perl

The XML::Parser package is a module that provides a perl interface to the `expat` XML parser.

This uses the standard perl module build process, so we have to override the default configure and test commands.

Configuration commands:

```
perl Makefile.PL
```

Compilation commands:

```
make
```

Test commands:

```
make test
```

Installation commands:

```
make install
```

Once the CBL system is complete, litbuild and the CBL blueprints can still be used to install new versions of all the packages that are already part of the system. They can also be used to install additional packages — many of the blueprints in the CBL repository are for packages that aren't part of the base system at all, and will only be used if you explicitly ask litbuild to produce scripts or build documentation for them (or for something that depends on them).

12.7.61. bash (final-system-components phase)

For an overview of bash, see bash.

Right after we install this package, we will delete the symbolic links we created previously (and create a new `sh` link that points to the final system `bash`). In order to be able to do that, we need to change the links to be owned by the new "bash" package user. The `-h` option for chown causes the ownership of the symbolic link *per se* to be changed, rather than the file it references.

Pre-build (as `root`) commands:

```
chown -h bash /bin/bash /bin/sh
```

We again disable the bash `malloc` routine. We also tell bash to use the readline library we've already installed.

Configuration commands:

```
./configure --prefix=/usr --without-bash-malloc --with-installed-readline
```

Compilation commands:

```
make
```

Test commands:

```
make tests
```

Installation of the final system `bash` is fussy! The `make install` target will crash unless there is a shell available at `/bin/sh`, so we need to leave that in place until after installing `bash`. But `/bin/bash` is a symbolic link to the scaffolding `bash`, and we need to remove that so that when the final system `bash` is copied there it won't simply overwrite the scaffolding version.

Installation commands:

```
rm -f /usr/bin/bash
```

There's another issue, as well: the package users wrapper scripts declare that they use the interpreter `/bin/bash`, and because `/bin` is a symbolic link to `/usr/bin`, there can't be a `/bin/bash` until this package is actually installed.

We can work around that with a kludge: before we run the `install` Makefile target to install this package properly, we'll copy the freshly-built `bash` program to its correct home in `/usr/bin`. If the installation routine copies it again, well, that's fine.

Installation commands:

```
cp bash /usr/bin
make install
```

Now that the real bash is finally installed, we can get rid of the `sh` symbolic link and recreate it as a link to it.

Installation commands:

```
rm -f /usr/bin/sh
ln -s bash /usr/bin/sh
```

12.7.62. ed

Name	GNU standard text editor
Version	1.19
Project URL	https://www.gnu.org/software/ed/
SCM URL	*(unknown)*
Download URL	http://ftpmirror.gnu.org/ed/

12.7.62.1. Overview

Ed is the standard text editor.

(See `http://repo.freesa.org/ed.txt` or `https://www.gnu.org/fun/jokes/ed-msg.html`)

12.7.62.2. ed (final-system-components phase)

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.63. bc

Name	GNU basic calculator
Version	1.07.1
Project URL	https://www.gnu.org/software/bc/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/bc/
Dependencies	ed, readline

12.7.63.1. Overview

`bc` (which, according to Wikipedia, is an acronym for "basic calculator") is an arbitrary-precision calculator — the documentation calls it a "numeric processing language," which is probably accurate, but mostly I use it as a calculator program.

"Arbitrary-precision" means that you're not limited to standard 32- or 64-bit integer math, or IEEE standard types of floating-point arithmetic, or anything like that. You can ask it "Hey, what's six to the sixth power, to the sixth power?" and it will obligingly give you a couple screens full of numbers.

12.7.63.2. bc (final-system-components phase)

Dependencies

readline.

Configuration commands:

```
./configure --prefix=/usr --with-readline
```

Compilation commands:

```
make
```

There is a nonstandard way to run the automated test suite for bc. You have to inspect the results manually to see if anything is wrong, though: it always exits with a 0 status code.

Test commands:

```
echo 'quit' | ./bc/bc -l Test/checklib.b
```

Installation commands:

```
make install
```

12.7.64. cmake

Name	CMake
Version	3.27.7
Project URL	https://cmake.org/
SCM URL	*(unknown)*
Download URL	https://cmake.org/download/
Dependencies	curl, expat, zlib, bzip2, xz

12.7.64.1. Overview

CMake is a configuration tool that generates Makefiles, or automated build files for other systems like ninja; it's an alternative to the GNU autotools-based build system. With CMake, rather than running a `configure` script with a bunch of options, you run `cmake` with a bunch of `-D` configuration directives to produce `Makefile`s, and (if you're not doing an in-place build) conclude with a relative or absolute path to the source directory. If you want to see what options are available, you can use an interactive interface like the `ccmake` curses-based UI to explore them. Any options you change in the `ccmake` session can then be specified with `-D` directives when running `cmake` later on.

Packages using the GNU build system can be told to use particular compiler or linker directives by setting environment variables like CFLAGS and LDFLAGS. I don't know what the equivalent for CFLAGS is in cmake, but to use specific options when linking programs (the equivalent of LDFLAGS), you can use the directives `CMAKE_EXE_LINKER_FLAGS` when building programs. Similar directives exist for other targets: `CMAKE_STATIC_LINKER_FLAGS` is used when building *static libraries*, `CMAKE_SHARED_LINKER_FLAGS` when building *shared libraries*, and `CMAKE_MODULE_LINKER_FLAGS` when building *modules* (probably kernel modules? I'm not actually sure.)

Another helpful tip: the Makefiles produced by cmake generally only talk about what they're doing, they don't echo the commands that are being run. You can run `make VERBOSE=1` to change this behavior.

The download page on cmake.org has a bunch of files available for each release. The tarfile you need is the file in the `Source distributions` section for `Unix/Linux Source`. If want to verify that it has not been corrupted or modified, you can also find in the `Summary files` section a SHA-256 checksum file, and a GPG signature for that checksum file.

12.7.64.2. cmake (final-system-components phase)

CMake uses a bunch of third-party libraries, and includes versions of them in its source distribution. Obviously, that's not the way we want to do things for CBL. Unfortunately, even if we set up all of those libraries ahead of time, the CMake bootstrap refuses to find at least one of them

(librhash); since that's the first one I tried to set up as a standalone library package, I'm not going out of my way to provide additional dependency libraries just for CMake.

Dependencies

curl, expat, zlib, bzip2, xz.

Configuration is also a little bizarro — even if directories like `docdir` are set with absolute paths, it appears that the CMake configuration system sets them up to be under the `prefix` location, for whatever reason.

Configuration commands:

```
./bootstrap --prefix=/usr --docdir=/share/doc/cmake \
  --system-curl \
  --system-expat \
  --no-system-jsoncpp \
  --system-zlib \
  --system-bzip2 \
  --system-liblzma \
  --no-system-libarchive \
  --no-system-librhash \
  --no-system-libuv
```

Compilation commands:

```
make
```

On my CBL system, three tests fail: CTestTestUpload, CTestTestStopTime, and RunCMake.string. I'm not concerned enough to investigate what's going on with those.

Test commands:

```
make test || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.65. tcl (final-system-components phase)

For an overview of tcl, see tcl.

Build Directory	`unix`

We want to run automated test suites once the CBL system is complete, obviously, so we will need tcl on the final system just as we do during the initial build.

Build Directory

`unix`

Configuration commands:

```
./configure --prefix=/usr --mandir=/usr/share/man \
  --infodir=/usr/share/info
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
make install-private-headers
```

12.7.66. expect (final-system-components phase)

For an overview of expect, see expect.

Configuration commands:

```
./configure --prefix=/usr --with-tcl=/usr/lib \
  --with-tclinclude=/usr/include
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.67. dejagnu (final-system-components phase)

For an overview of dejagnu, see dejagnu.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.68. diffutils (final-system-components phase)

For an overview of diffutils, see diffutils.

By default, diffutils uses `ed` as the default text editor for `sdiff`. On CBL systems, we prefer `vim` as the default editor; feel free to adjust this to taste.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
sed -i 's@\(^#define DEFAULT_EDITOR_PROGRAM \).*@\1"vim"@' lib/config.h
make
```

The `test-strftime` test fails because two of the expected UTC date strings are offset by a few seconds. As always, inspect the test logs and determine whether the results are satisfactory.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.69. util-linux (final-system-components phase)

For an overview of util-linux, see util-linux.

This is a fairly normal package-user build. If python is available when it is installed, a python `libmount` library gets installed, so it's a good idea to make sure python is installed first.

Dependencies

python.

Configuration commands:

```
./configure --enable-write
```

Compilation commands:

```
make
```

Many of the util-linux tests will be skipped any time the suite is run by a non-privileged user. If you want to run the most thorough test suite, run them as root.

One of the tests fails because the `openpty` function does not work properly in the partial-system environment. In fact, it fails in such a way that putting in a guard clause doesn't work; the build still aborts entirely. As with other packages whose test suite is problematic in the partial environment, we skip the tests entirely for now.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.70. e2fsprogs (final-system-components phase)

For an overview of e2fsprogs, see e2fsprogs.

Build Directory	`build`

This is configured similarly to the scaffolding version.

Configuration commands:

```
../configure --enable-elf-shlibs \
  --disable-libblkid --disable-libuuid --disable-fsck --disable-uuidd \
  --without-crond-dir
```

Compilation commands:

```
make
```

At one test (`f_pre_1970_date_encoding`) sometimes fails for me.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
make install-libs
```

12.7.71. elfutils

Name	ELF utilities
Version	0.189
Project URL	https://sourceware.org/elfutils/
SCM URL	*(unknown)*
Download URL	https://sourceware.org/elfutils/ftp/
Dependencies	curl

12.7.71.1. Overview

According to its homepage, the elfutils are a collection of utilities and libraries to read, create and modify ELF binary files, find and handle DWARF debug data, symbols, thread state and stacktraces for processes and core files on GNU/Linux.

Depending on the configuration settings and architecture in use, the Linux kernel build process sometimes makes use of the `libelf` library that is part of this package, to build a program that is used to analyze object files during the kernel compilation process. It is therefore a possible dependency for the kernel. It may only be needed for `X86_64` kernels.

This should not be confused with the `libelf` project found at `http://www.mr511.de/software/english.html`; that's an entirely different and unrelated library.

12.7.71.2. elfutils (final-system-components phase)

Configuration commands:

```
CFLAGS="$CFLAGS -Wno-error=packed-not-aligned" ./configure \
  --prefix=/usr --sysconfdir=/etc --disable-debuginfod
```

Compilation commands:

```
make
```

Some of the tests do not reliably succeed.

Test commands:

```
make check || echo "exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.72. file (final-system-components phase)

For an overview of file, see file.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.73. findutils (final-system-components phase)

For an overview of findutils, see findutils.

Configuration commands:

```
./configure --prefix=/usr --localstatedir=/var/lib/locate
```

Compilation commands:

```
make
```

The `test-strftime` test fails because two of the expected UTC date strings are offset by a few seconds. As always, inspect the test logs and determine whether the results are satisfactory.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.74. flit

Name	Flit Python build system
Version	3.9.0
Project URL	https://flit.pypa.io/en/stable/
SCM URL	https://github.com/pypa/flit
Download URL	https://github.com/pypa/flit/tags

12.7.74.1. Overview

Flit is a build and packaging system for Python packages and modules, designed to make it easy to publish them on PyPI.

12.7.74.2. flit (flit_core phase)

Building the full flit package requires `build` and possibly `installer`, which bring in additional dependencies. Some of those required pacakges depend on `flit_core` — luckily, they do not depend on the full `flit` program — so we need to start by building that.

flit_core has only one problematic dependency, a TOML parser called `tomli`. To avoid a cyclic dependency, there is a version of tomli bundled with flit_core.

Configuration commands:

```
(none)
```

Compilation commands:

```
pushd flit_core
python -m flit_core.wheel
popd
```

Test commands:

```
(none)
```

Installation commands:

```
pushd flit_core
python bootstrap_install.py dist/flit_core-*.whl
popd
```

12.7.75. packaging

Name	Packaging
Version	23.2
Project URL	https://pypi.org/project/packaging/
SCM URL	https://github.com/pypa/packaging
Download URL	https://pypi.org/project/packaging/#files
Dependencies	flit (flit_core phase)

12.7.75.1. Overview

Packaging is a python module that contains APIs related to software packaging.

12.7.75.2. packaging (final-system-components phase)

Dependencies

flit (flit_core phase).

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/packaging*whl
```

12.7.76. tomli

Name	TOML library for python
Version	2.0.1

Project URL	https://pypi.org/project/tomli/
SCM URL	https://github.com/hukkin/tomli
Download URL	https://pypi.org/project/tomli/#files
Dependencies	flit (flit_core phase)

TOML is an acronym for "Tom's Obvious, Minimal Language." It's a syntax for configuration files commonly used for python package metadata files. `tomli` is a TOML parser.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/tomli*whl
```

12.7.77. pyproject_hooks

Name	pyproject-hooks
Version	1.0.0
Project URL	https://pypi.org/project/pyproject_hooks/
SCM URL	https://github.com/pypa/pyproject-hooks
Download URL	https://pypi.org/project/pyproject_hooks/#files
Dependencies	flit (flit_core phase), tomli

pyproject_hooks is a low-level python library that contains wrappers to call build backend programs for projects that use a `pyproject.toml` file to describe their build process and dependencies.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/pyproject*whl
```

12.7.78. build

Name	Build library for python wheels
Version	1.0.3
Project URL	https://pypi.org/project/build/
SCM URL	https://github.com/pypa/build
Download URL	https://pypi.org/project/build/#files
Dependencies	flit (flit_core phase), packaging, pyproject_hooks

Build is a python library for constructing "wheel" distribution files for python projects.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/build*whl
```

12.7.79. setuptools

Name	setuptools
Version	68.2.2

Project URL	https://pypi.org/project/setuptools/
SCM URL	https://github.com/pypa/setuptools
Download URL	https://pypi.org/project/setuptools/#files
Dependencies	python

12.7.79.1. Overview

setuptools is one of the original tools in the python build ecosystem. It has historically been bundled with python (or possibly pip, which amounts to the same thing), so it has not been necessary to install it separately; that changed in recent versions of python, but it's still required to build other packages like meson.

12.7.79.2. setuptools (final-system-components phase)

The build process is typical for old-style python packages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

12.7.80. docutils

Name	Docutils
Version	0.20.1
Project URL	http://docutils.sourceforge.net/
SCM URL	http://svn.code.sf.net/p/docutils/code/trunk
Download URL	https://sourceforge.net/projects/docutils/files/docutils/
Dependencies	setuptools

Docutils is a text processing system for transforming plaintext documentation into formats like HTML and LaTeX.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

12.7.81. installer

Name	Installer library for python wheels
Version	0.7.0
Project URL	https://pypi.org/project/installer/
SCM URL	https://github.com/pypa/installer
Download URL	https://pypi.org/project/installer/#files
Dependencies	flit (flit_core phase)

Installer is a low-level python library for installing "wheel" distribution files. It's being installed here in accordance with the bootstrap suggestions in flit.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/install*whl
```

12.7.82. certifi

Name	TLS CA Certificates
Version	2023.7.22
Project URL	https://pypi.org/project/certifi/
SCM URL	https://github.com/certifi/python-certifi
Download URL	https://pypi.org/project/certifi/#files

Certifi is the Mozilla bundle of TLS CA certificates.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

12.7.83. urllib3

Name	urllib3
Version	1.26.18
Project URL	https://urllib3.readthedocs.io/en/latest/
SCM URL	git://github.com/urllib3/urllib3
Download URL	https://github.com/urllib3/urllib3/releases

This is an HTTP client library for Python, providing thread safety, connection pools, proxy support, and other handy features.

As is typical for Python programs, there aren't any configuration or test stages.

There is a newer version of urllib3 available — v2.0 — which requires some work by clients to upgrade. Normally, I'd rename this version of the package to `urllib3v1` or something; but the only reason there's a blueprint of urllib3 at all is because it is a required dependency for other packages, and those packages can't use the modern version of urllib3 yet, so I'm not taking the time.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

12.7.84. idna

Name	Python IDNA library
Version	3.4
Project URL	https://pypi.org/project/idna/
SCM URL	https://github.com/kjd/idna
Download URL	https://pypi.org/project/idna/#files
Dependencies	flit (flit_core phase)

The idna package provides support for the Internationalized Domain Names in Applications (IDNA) protocol in python.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/idna*whl
```

12.7.85. charset-normalizer

Name	Python character set detector
Version	3.3.2
Project URL	https://pypi.org/project/charset-normalizer/
SCM URL	https://github.com/Ousret/charset_normalizer
Download URL	https://pypi.org/project/charset-normalizer/#files

charset-normalizer is a library that makes it easy to read text from an unknown charset encoding.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

12.7.86. requests

Name	Python HTTP library
Version	2.31.0
Project URL	https://requests.readthedocs.io/en/latest/
SCM URL	https://github.com/psf/requests
Download URL	https://pypi.org/project/requests/#files
Dependencies	certifi, urllib3, idna, charset-normalizer

Requests is a simple HTTP library.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

As with other python packages, requests tries to re-install files that are owned by other packages.

Pre-build (as `root`) commands:

```
for wrapper in /usr/bin/normalizer; \
  do \
  mv $wrapper ${wrapper}.MOVED; \
  done
```

Post-installation (as `root`) commands:

```
for wrapper in /usr/bin/normalizer.MOVED; \
  do \
  mv $wrapper ${wrapper%.MOVED}; \
  done
```

12.7.87. tomli-w

Name	TOML writer library for Python
Version	1.0.0
Project URL	https://pypi.org/project/tomli-w/
SCM URL	https://github.com/hukkin/tomli-w
Download URL	https://pypi.org/project/tomli-w/#files
Dependencies	flit (flit_core phase)

TOML is an acronym for "Tom's Obvious, Minimal Language." It's a syntax for configuration files commonly used for python package metadata files. `tomli-w` is a TOML writer.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/tomli*whl
```

12.7.88. flit (final-system-components phase)

For an overview of flit, see flit.

Dependencies

build, docutils, installer, requests, tomli-w.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/flit*whl
```

12.7.89. gettext (final-system-components phase)

For an overview of gettext, see gettext.

One of the tests — `lang-gawk` — fails, perhaps because gawk is a later version than the tests expect. The issue is that the test expects the string "EUR remplace FF." but instead gets "FF is replaced by EUR."

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.90. pcre2

Name	Perl Compatible Regular Expressions
Version	10.42
Project URL	https://www.pcre.org/
SCM URL	https://github.com/PCRE2Project/pcre2
Download URL	https://github.com/PCRE2Project/pcre2/releases
Dependencies	zlib, bzip2, readline

PCRE is a library of functions that implement regular expression pattern matching using the same syntax as Perl 5. It's used by a bunch of other packages to provide regular expression capabilities.

There are two versions of the PCRE library: PCRE and PCRE2. The former is the original version, and is still used by many packages. It has now been declared end-of-life and is no longer maintained.

This version, PCRE2, was first released in 2015; the project maintainers urge people who want to use PCRE to use this version.

There are several optional dependencies that provide enhanced capabilities in PCRE; these are already part of CBL, so we go ahead and enable them.

Dependencies

zlib, bzip2, readline.

Configuration commands:

```
./configure --prefix=/usr --enable-jit --enable-pcre2-16 \
  --enable-pcre2-32 --enable-pcre2grep-libz --enable-pcre2grep-libbz2 \
```

```
  --enable-pcre2test-libreadline
```

Compilation commands:

```
make
```

The tests don't work right in the partial CBL build, possibly because of missing dependency flags.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.91. xmlto

Name	XML-to-any converter
Version	0.0.28
Project URL	https://pagure.io/xmlto/
SCM URL	*(unknown)*
Download URL	http://releases.pagure.org/xmlto/
Dependencies	libxslt

xmlto is a front-end wrapper script that can run other programs to convert XML documents into a variety of output formats — man pages, HTML, XHTML, PDF, and others. It is basically a convenience for people who don't want to have to remember or look up how to invoke programs like xsltproc, FOP, and passivetex.

This project seems to be more-or-less orphaned. As of December 2022, the only activity since 2015 has happened in forked versions; one of those, by Thomas Kuehne, was merged to master a year ago but there are two outstanding pull requests from earlier this year and those have just been sitting without any response or activity. I wouldn't suggest using it for anything — just figure out the correct incantation to invoke xsltproc, or whatever, directly.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.92. docbook-xsl

Name	DocBook XSL Stylesheets
Version	snapshot-2020-06-03
Project URL	https://github.com/docbook/xslt10-stylesheets
SCM URL	https://github.com/docbook/xslt10-stylesheets
Download URL	https://github.com/docbook/xslt10-stylesheets/releases
Dependencies	libxml2, libxslt

DocBook is an XML markup language for technical documentation. It lets you create document content in a presentation-neutral form; you can then use programs to transform that content into a variety of formats. It's formally defined by a "schema," with several different versions available.

This package — DocBook XSL — is a set of XSLT stylesheets. These stylesheets define rules by which DocBook XML documents can be transformed into output formats intended to be used by people: HTML (single-page and section-per-page), XHTML, XSL-FO (an intermediate form that can then be transformed by other programs into PDF or other formats), man pages, WebHelp, and probably dozens of other things.

As of May 2017, you can learn a lot more about DocBook XSL by reading a book available online at `http://www.sagehill.net/docbookxsl/index.html`.

It appears that the only releases being produced for this package are "snapshot" releases created through a continuous integration system; the most recent numbered release was 1.79.2, produced in December of 2016 when the project migrated from sourceforge.net to github. Rather than continuing to use that version, I use the latest snapshot version in CBL.

There are two different distributions of XSL stylesheets available from the docbook-xsl project: the standard `docbook-xsl` tar file has a DocBook namespace prefix present in element names within the stylesheets and is intended for use with DocBook 5. A second form without the namespace prefix, intended for use with DocBook 4, is distributed as `docbook-xsl-nons`.

In addition to those two distributions — which are found in `zip` files for each release on the download page — there's a source distribution found in a normal gzip-compressed tar file, which contains a Makefile that can presumably be used to produce the compiled stylesheets. As of December 2022, I can't get it to work, so I'm just using the stylesheet distributions.

Since we're not building the stylesheets, there's no configuration, compilation, or testing steps.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
(none)
```

There's no automated installation process for this package, either. We're just going to copy the package files to a reasonable system location.

Installation commands:

```
mkdir -p /usr/share/xml/docbook-xsl-${version}
cp -v -R * /usr/share/xml/docbook-xsl-${version}
```

After installing the files, update the XML catalog so that they will be easy to find. There are two different sets of URLs we're specifying here: one, with a `docbook.sourceforge.net` hostname, will handle documents that are using the old canonical stylesheet URLs from version 1.79.1 and earlier; the other, at `cdn.docbook.org`, is the modern location for these stylesheets.

Installation commands:

```
xmlcatalog --noout --add "rewriteSystem" \
  "http://docbook.sourceforge.net/release/xsl-ns/${version}" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
  "http://docbook.sourceforge.net/release/xsl-ns/${version}" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteSystem" \
  "http://docbook.sourceforge.net/release/xsl-ns/current" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
  "http://docbook.sourceforge.net/release/xsl-ns/current" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteSystem" \
  "http://cdn.docbook.org/release/xsl/${version}" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
  "http://cdn.docbook.org/release/xsl/${version}" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteSystem" \
  "http://cdn.docbook.org/release/xsl/current" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
xmlcatalog --noout --add "rewriteURI" \
```

```
  "http://cdn.docbook.org/release/xsl/current" \
  "/usr/share/xml/docbook-xsl-${version}" /etc/xml/catalog
```

12.7.93. git (final-system-components phase)

For an overview of git, see git.

Dependencies

zlib, gettext, openssh, libressl, curl (final-system-components phase), expat, pcre2.

This is a nearly-complete build of the git version control system, including optional components and documentation. Since there is no GUI environment in the base CBL system, the `gitk` and `git-gui` components are not built here. (The info version of the documentation is also not built.)

The documentation for git is maintained in AsciiDoc format; this can be processed either by the package of that name or, by using an additional build flag, by the newer Asciidoctor package. There are a variety of packages that are needed to produce the documentation, depending on what output formats are desired — according to the `INSTALL` file:

- AsciiDoc or Asciidoctor is always required. For HTML documentation, nothing else is needed.
- For man-page output, `xmlto`, `xsltproc`, and the DocBook XSL stylesheets are needed.
- For info output, `docbook2X` and `texinfo` are needed.
- For PDF output, `dblatex` is needed.

Dependencies

asciidoctor.

I find it useful enough to have the documentation available as man pages that I go through the work of setting up `xmlto` here — the other packages are already part of the basic CBL process, so it's not that far out of our way to get the man pages. The other documentation targets are too much trouble, so I don't worry about them.

Dependencies

xmlto, docbook-xsl.

The xmlto package is old and appears to be orphaned but is still in use in the git documentation build process.

It would be worthwhile to update the documentation build process for git to eliminate the xmlto dependency — xmlto appears to be just a wrapper around `xsltproc` and the docbook XSL stylesheets, so those could be used directly instead — but that's more work than I've been able to put in to the challenge as yet. It appears that others have done some work in this direction; a search of the git mailing list archive shows that a set of patches were submitted to generate man-page output directly using Asciidoctor, without going through docbook, but that patch set has apparently not been merged into mainline git as of release 2.38.

The build system is designed to use the Python AsciiDoc package, but can be told to use Asciidoctor instead with the `USE_ASCIIDOCTOR` flag. I'm not sure whether that flag needs to be set at configure

time, build time, or both. It doesn't hurt anything to set it in both locations. It actually seems like it's important to specify it at installation time, as well, which doesn't make a huge amount of sense to me, but whatever.

Configuration commands:

```
USE_ASCIIDOCTOR=YesPlease ./configure --prefix=/usr --sysconfdir=/etc \
  --with-libpcre2 --with-curl --with-expat
```

Compilation commands:

```
USE_ASCIIDOCTOR=YesPlease make all
USE_ASCIIDOCTOR=YesPlease make doc
```

Some of the git tests fail in the partial environment.

Test commands:

```
make test || echo "Exit code $?: continuing anyway"
```

The installation routine for git has something weird going on: it creates a tar file with locale data, then tries to expand that tar file into the standard filesystem location — which promptly fails because tar tries to modify directory ownership and permissions to match what is recorded in the tar file, and package users are not allowed to do this.

To work around this, we initially install with a `DESTDIR` set to a new directory; then we copy everything from that directory to the final system locations.

Installation commands:

```
tempgitdir=$(mktemp -d)
USE_ASCIIDOCTOR=YesPlease make DESTDIR=$tempgitdir install
USE_ASCIIDOCTOR=YesPlease make DESTDIR=$tempgitdir install-doc
USE_ASCIIDOCTOR=YesPlease make DESTDIR=$tempgitdir install-html
pushd $tempgitdir
cp -d -f -R -v --parents usr /
popd
rm -rf $tempgitdir
```

12.7.94. pcre

Name	Legacy Perl Compatible Regular Expressions
Version	8.45
Project URL	https://www.pcre.org/
SCM URL	*(unknown)*

Download URL	https://sourceforge.net/projects/pcre/files/
Patches	• pcre-8.45-sljit_mips-label-statement-fix.patch
Dependencies	zlib, bzip2, readline

12.7.94.1. Overview

PCRE is a library of functions that implement regular expression pattern matching using the same syntax as Perl 5. It's used by a bunch of other packages to provide regular expression capabilities.

There are two versions of the PCRE library: PCRE and PCRE2. This blueprint is for PCRE, also known as PCRE1, which is still commonly used by other packages. It has been end-of-lifed and is no longer maintained.

Patch:

- pcre-8.45-sljit_mips-label-statement-fix.patch

In some scenarios, a syntax error in a MIPS-specific file causes problems. I don't remember exactly where I found this fix, but it worked.

12.7.94.2. pcre (final-system-components phase)

There are several optional dependencies that provide enhanced capabilities in PCRE; these are already part of CBL, so we go ahead and enable them.

Dependencies

zlib, bzip2, readline.

Configuration commands:

```
./configure --prefix=/usr --enable-jit --enable-pcre16 \
  --enable-pcre32 --enable-utf --enable-unicode-properties \
  --enable-pcregrep-libz --enable-pcregrep-libbz2 \
  --enable-pcretest-libreadline
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.95. ninja

Name	Ninja
Version	1.11.1
Project URL	https://ninja-build.org/
SCM URL	https://github.com/ninja-build/ninja
Download URL	https://github.com/ninja-build/ninja/releases
Dependencies	python

12.7.95.1. Overview

Ninja is a relatively new build tool, designed with the expectation that the files used to drive it are generated by other, higher-level build systems. Its primary design goal is to be as fast as possible.

12.7.95.2. ninja (final-system-components phase)

Configuration commands:

```
(none)
```

There is a bootstrap process that uses a python script to compile a ninja program just by compiling all non-test source code files, and then uses that program to rebuild itself.

Compilation commands:

```
./configure.py --bootstrap
```

Test commands:

```
./ninja ninja_test
./ninja_test
```

There is no installation routine for ninja; the only thing that needs to be done to install the program is to copy the `ninja` program to one of the standard system binary directories.

There are emacs and vim editor files that can facilitate editing ninja build files, but I'm not installing those here.

Installation commands:

```
cp ninja /usr/bin
```

12.7.96. meson

Name	The Meson Build System

Version	1.2.3
Project URL	http://mesonbuild.com/
SCM URL	https://github.com/mesonbuild/meson
Download URL	https://github.com/mesonbuild/meson/releases
Dependencies	python, setuptools, ninja

12.7.96.1. Overview

Meson is a build system implemented in python 3. Its main goals are to be very fast and user-friendly.

Meson is a high-level build system that, when used on Unix-ish systems, runs on top of the Ninja build tool. (According to its homepage, meson can also generate project files for Microsoft Visual Studio and the MacOS Xcode development environment.)

Dependencies

python, setuptools, ninja.

12.7.96.2. meson (final-system-components phase)

As is typical for Python programs, there is no configuration stage.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

The meson package does include some automated tests, but they do not work for me — probably because of missing dependencies. I don't care enough to track down how to get the tests running.

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

12.7.97. glib

Name	GLib
Version	2.78.1

Project URL	https://wiki.gnome.org/Projects/GLib
SCM URL	*(unknown)*
Download URL	https://download.gnome.org/sources/glib
Patches	• `glib-2.78.1-replace-distutils-with-packaging-1.patch`
Dependencies	libffi, pcre, pkgconf, python, meson, cmake, packaging, util-linux, pcre

12.7.97.1. Overview

GLib is a collection of functions used widely in other software projects. It was originally part of the GIMP Toolkit library ("GTK" or "GTK+"); some of the functions in the GIMP Toolkit were useful outside of a GUI context, so the project maintainers decided to split those functions into separate libraries. The result of that refactoring eventually resulted in GLib.

GLib provides implementations of several useful data structures, string utilities, concurrency primitives, object-orientation (through the GLib Object System), a virtual filesystem (through GIO), and similar low-level features.

As of GLib version 2.60, the build system has changed from the GNU build system to meson.

Dependencies

meson.

And as of version 2.70, it apparently *also* needs CMake. (Or, at least, emits error messages that suggest it wants CMake if it can't find other dependencies, like pcre.) I don't really understand why — I mean, how many different build systems can you need at the same time? — but here we are.

Dependencies

cmake.

As you may recall from the python section, the python package ecosystem is currently in a state of disarray: packages like setuptools have been deprecated, and support for them is vanishing, but many downstream projects still depend on them. GLib is an example of this: it uses distutils, which has been deprecated and removed from the core python package. The specific feature needed by GLib is `distutils.version`, and the recommended alternative to this is the `packaging` package. To get GLib working with the latest stable python, we'll add that as a dependency, and then patch GLib to use it.

Dependencies

packaging.

Patch:

- glib-2.78.1-replace-distutils-with-packaging-1.patch

12.7.97.2. glib (final-system-components phase)

Dependencies

util-linux, pcre.

Configuration commands:

```
meson setup --prefix /usr --buildtype release --sysconfdir /etc _build
```

Compilation commands:

```
ninja -C _build
```

Some of the tests still fail. I haven't found this important enough to spend effort addressing.

Test commands:

```
ninja -C _build test || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
ninja -C _build install
rm -rf .cache .dbus-keyrings .local
```

12.7.98. nettle

Name	GNU Nettle cryptographic library
Version	3.9.1
Project URL	https://www.lysator.liu.se/~nisse/nettle/
SCM URL	https://git.lysator.liu.se/nettle/nettle
Download URL	https://ftp.gnu.org/gnu/nettle/
Dependencies	gmp, m4

Nettle is a low-level cryptographic library, like libgcrypt. It's present in CBL because some packages depend on it.

This is one of the packages that runs `install-info` in a problematic way.

Pre-build (as `root`) commands:

```
chown nettle:install /usr/share/info/dir
```

Post-installation (as `root`) commands:

```
chown root:install /usr/share/info/dir
```

```
chmod 664 /usr/share/info/dir
```

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.99. libtasn1

Name	GNU Libtasn1
Version	4.19.0
Project URL	https://www.gnu.org/software/libtasn1/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/libtasn1/

GNU Libtasn1 provides a library and some utility programs for parsing and otherwise dealing with Abstract Syntax Notation One (ASN.1) and Distinguished Encoding Rules (DER). These are used TLS certificates and in other things related to cryptography; some of the CBL packages are more useful if libtasn1 is available.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.100. gnutls

Name	GnuTLS Library
Version	3.8.1
Project URL	https://gnutls.org
SCM URL	*(unknown)*
Download URL	https://www.gnupg.org/ftp/gcrypt/gnutls/
Dependencies	pkgconf, nettle, libtasn1

12.7.100.1. Overview

GnuTLS is, like OpenSSL and LibreSSL, an implementation of the SSL and TLS protocols and related things. It's a part of the GNU project.

Sometimes — as with version 3.6.7.1 — the gnutls distribution file expands to a directory that doesn't match the expected name. In those cases, of course the tar file needs to be repackaged using the standard convention.

Since the GnuTLS package does not conflict with LibreSSL or OpenSSL, it is installed into the standard `/usr` location, where it is easy for other packages to find it.

If desired, the libunistring library can be installed separately rather than using the one bundled with GnuTLS. In most cases I prefer to install standalone versions of libraries; in this case, since I don't otherwise need libunistring, I just use the one that comes with GnuTLS.

If PKCS #11 support is desired, the p11-kit package should be installed before GnuTLS, and the configure directive about it can be removed.

GnuTLS provides support for secure DNS if the libunbound package at http://unbound.net/ is available. If that matters to you, you might want to set that up first.

12.7.100.2. gnutls (final-system-components phase)

Configuration commands:

```
./configure --prefix=/usr --with-included-unistring \
  --without-p11-kit --enable-ssl3-support --disable-ssl2-support \
  --enable-openssl-compatibility
```

Compilation commands:

```
make
```

The test suite sometimes hangs in the partial environment, for reasons I haven't discovered.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.101. gzip (final-system-components phase)

For an overview of gzip, see gzip.

Some of the gzip tests fail. These usually aren't very important ones, but you can inspect the test logs after the build is complete.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.102. kbd (final-system-components phase)

For an overview of kbd, see kbd.

Configuration commands:

```
./configure --prefix=/usr --disable-vlock --enable-optional-progs
```

Compilation commands:

```
make
```

The documentation doesn't get installed by default, so we have to do that ourselves.

Test commands:

```
make check
```

Installation commands:

```
make install
mkdir /usr/share/doc/kbd
cp -R -v docs/doc/* /usr/share/doc/kbd
```

12.7.103. libslirp

Name	libslirp
Version	4.7.0
Project URL	https://gitlab.freedesktop.org/slirp/libslirp
SCM URL	https://gitlab.freedesktop.org/slirp/libslirp
Download URL	https://gitlab.freedesktop.org/slirp/libslirp/-/tags
Dependencies	glib, meson, ninja

12.7.103.1. Overview

libslirp is a user-mode networking library. It can be used by programs like QEMU to provide networking services without any involvement from the kernel.

The functionality provided by this package is used by user-mode networking in QEMU. The code necessary to support this feature was previously provided as a part of the QEMU package; with QEMU release 7.2.0, however, it was removed from QEMU and must be provided separately if user-mode networking is needed.

12.7.103.2. libslirp (final-system-components phase)

Configuration commands:

```
meson setup --prefix=/usr --libdir /usr/lib _build
```

Compilation commands:

```
ninja -C _build
```

Test commands:

```
(none)
```

Installation commands:

```
ninja -C _build install
```

12.7.104. linux (final-system-components phase)

For an overview of linux, see linux.

It's finally time to build the final system kernel! Since this will be used for the actual Little Blue Linux, we'll configure it with a little more care than we have previously, so that all the features we're likely to need are available.

Dependencies

bc, kmod.

As mentioned earlier, the kernel build process requires a TLS library.

Dependencies

libressl.

For `x86_64` builds (and possibly other architecture builds, as well), the kernel build process compiles some programs that it uses to validate object code produced later in the build process. At least one of these programs requires the `libelf` library, which is part of the elfutils package.

Dependencies

elfutils.

The kernel build this time is similar to the one we did with the cross-toolchain. We start with the kernel configuration we had before; because we enabled the `IKCONFIG` and `IKCONFIG_PROC` options, we can get it from the `/proc` filesystem.

Configuration commands:

```
make mrproper
cat /proc/config.gz | gzip -d > .config
make olddefconfig
```

This time we *do* want the kernel to auto-mount the `devtmpfs` at `/dev` while booting — that saves us the trouble of doing it, and this time we will actually have a `/dev` directory at boot time!

Configuration commands:

```
./scripts/config --enable DEVTMPFS_MOUNT
```

12.7.104.1. Configuration Settings For Containers

Virtual machines — such as are provided by the qemu package — have been around since the 1970s. If you want to emulate other CPU architectures on your computer, or run multiple operating

systems of different types, then virtual machines are definitely the solution for you! But in many cases, the goal is simply to split a large physical computer into some number of smaller virtual machines so that the workload on each can run independently and isolated from the others. This is intrinsic to the "cloud computing" model that has become so popular since 2006 or so.

If you just want to isolate some of the processes on a computer from others, so that each group of processes can act as though they have a computer to themselves, full system virtual machines can be overkill. Each virtual machine has a full operating system kernel, which imposes a pretty significant amount of overhead. To avoid this overhead, some additional sets of features were added to the Linux kernel:

1. *namespaces* allow a set of processes to run in isolation from other processes, so that each set sees a different view of what is happening on the computer;
2. *control groups* allow restrictions to be imposed on sets of processes so that there are well-defined limits on what system resources each set is allowed to use;
3. *seccomp* (for "secure computing") allows the kernel to restrict the system calls that some sets of processes are allowed to invoke; and
    - *selinux* (for "security enhanced linux") allows labels to be attached to things like processes and files, and allows rules to be defined that restrict how labeled things interact with each other.

Together, these features allow a set of processes to be run in a lightweight virtualized environment, partitioned away from other processes on the same computer. When a set of processes is run in this way, they are said to be running in a 'container."

This set of techniques has become popular enough that an entire ecosystem of tools have grown up around containers: LXC, Docker, Kubernetes, and many others.

We want to be able to run containers on Little Blue Linux (although the userspace components that do this are not part of the base system), so we need to enable a bunch of additional kernel features.[3]

For namespaces:

Configuration commands:

```
./scripts/config --enable PID_NS
./scripts/config --enable USER_NS
```

For control groups:

Configuration commands:

```
./scripts/config --enable PAGE_COUNTER
./scripts/config --enable MEMCG
./scripts/config --enable BLK_CGROUP
./scripts/config --enable DEBUG_BLK_CGROUP
./scripts/config --enable CGROUP_WRITEBACK
./scripts/config --enable CFS_BANDWIDTH
```

```
./scripts/config --enable RT_GROUP_SCHED
./scripts/config --enable CGROUP_PIDS
./scripts/config --enable CGROUP_RDMA
./scripts/config --enable CGROUP_HUGETLB
./scripts/config --enable CGROUP_DEVICE
./scripts/config --enable CGROUP_PERF
./scripts/config --enable CGROUP_DEBUG
./scripts/config --enable BPF_SYSCALL
./scripts/config --enable CGROUP_BPF
./scripts/config --enable BLK_DEV_THROTTLING
./scripts/config --enable NET_CLS_CGROUP
./scripts/config --enable CGROUP_NET_PRIO
```

For security:

Configuration commands:

```
./scripts/config --enable SECURITY_SELINUX
./scripts/config --enable SECURITY_APPARMOR
```

We also enable the use of "overlay" filesystems, which allow new filesystems to be layered on top of an existing filesystem; this allows multiple filesystems to share an underlying base filesystem without duplicating its contents.

Configuration commands:

```
./scripts/config --enable OVERLAY_FS
```

Some additional networking features are often helpful in allowing containers to communicate with each other, and with the outside world.

Configuration commands:

```
./scripts/config --enable VETH
./scripts/config --enable VXLAN
./scripts/config --enable BRIDGE
./scripts/config --enable NETFILTER_ADVANCED
./scripts/config --enable BRIDGE_NETFILTER
./scripts/config --enable VLAN_8021Q
./scripts/config --enable BRIDGE_VLAN_FILTERING
./scripts/config --enable NETFILTER_XT_MATCH_IPVS
./scripts/config --enable IP_NF_TARGET_REDIRECT
./scripts/config --enable IP_VS
./scripts/config --enable IP_VS_NFCT
./scripts/config --enable IP_VS_PROTO_TCP
./scripts/config --enable IP_VS_PROTO_UDP
./scripts/config --enable IP_VS_RR
./scripts/config --enable CRYPTO
./scripts/config --enable CRYPTO_SEQIV
```

```
./scripts/config --enable XFRM
./scripts/config --enable XFRM_USER
./scripts/config --enable XFRM_ALGO
./scripts/config --enable INET_ESP
./scripts/config --enable IPVLAN
./scripts/config --enable MACVLAN
./scripts/config --enable DUMMY
./scripts/config --enable NF_NAT_FTP
./scripts/config --enable NF_CONNTRACK_FTP
./scripts/config --enable NF_NAT_TFTP
./scripts/config --enable NF_CONNTRACK_TFTP
```

There are some filesystems that can be used for container storage, so we might as well enable them as well.

Configuration commands:

```
./scripts/config --enable BTRFS_FS
./scripts/config --enable BTRFS_FS_POSIX_ACL
./scripts/config --enable DM_THIN_PROVISIONING
./scripts/config --enable EXT4_FS
./scripts/config --enable EXT4_FS_POSIX_ACL
./scripts/config --enable EXT4_FS_SECURITY
```

Containers are not part of the basic CBL process, because they are not needed as part of a minimal self-hosted system. A set of userspace tools that allow the use of containers can be built using the `containers` blueprints, as described in the [containers] appendix.

12.7.104.2. Other Configuration Settings

The kernel has support for detecting some types of buffer overflow attacks. This is a good idea if you ever might run malicious or untrusted code on your computer.

Configuration commands:

```
./scripts/config --enable CC_STACKPROTECTOR_REGULAR
```

Linux also supports virtual machine acceleration on CPUs that provide those features. As with the x32 ABI earlier, this is architecture-specific; when the options are irrelevant, the `olddefconfig` Makefile target will simply remove them.

Configuration commands:

```
./scripts/config --module KVM
./scripts/config --module KVM_INTEL
./scripts/config --module KVM_AMD
./scripts/config --module VHOST
./scripts/config --module VHOST_NET
```

You might want to run `make nconfig` and browse through all the options to customize the kernel further. Minimally, it's a good idea to ensure that your kernel will include support for all your hardware! But this is pretty good for a basic starting point.

Configuration commands:

```
make olddefconfig
```

Now we can build the kernel and modules.

Compilation commands:

```
make all
make Image.gz
```

Test commands:

```
(none)
```

To install modules, it's easiest to use the Makefile target. This automatically installs the modules to a subdirectory of `/lib/modules`, named with the kernel version and `LOCALVERSION` — which is where the module utilities will look for them.

Installation commands:

```
make modules_install
```

To install the kernel itself, we can again simply copy files where we want them instead of using the `install` target.

We can get the kernel release name by looking at the subdirectory that was just installed under `/lib/modules`. (There may be some easier way to do this, but if there is, I don't know it.)

Installation commands:

```
export KERNELRELEASE=$(ls -1t /lib/modules | head -n 1)
export KERNELPATH=$(find . -name Image.gz -a -type f)
cp -v $KERNELPATH /boot/kernel-$KERNELRELEASE
cp -v .config /boot/config-$KERNELRELEASE
lzip -9 /boot/config-$KERNELRELEASE
cp -v System.map /boot/System.map-$KERNELRELEASE
depmod -e -F /boot/System.map-$KERNELRELEASE -a $KERNELRELEASE
```

12.7.105. lzo

Name	LZO
Version	2.10

Project URL	http://www.oberhumer.com/opensource/lzo/
SCM URL	*(none)*
Download URL	http://www.oberhumer.com/opensource/lzo/

12.7.105.1. Overview

LZO is a compression library, like gzip and bzip2 and so on. It decompresses very very quickly; that's apparently its main benefit.

12.7.105.2. lzo (final-system-components phase)

LZO is installed in Little Blue Linux because it's used by lrzip.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.106. lz4

Name	LZ4
Version	1.9.4
Project URL	http://lz4.github.io/lz4/
SCM URL	*(unknown)*
Download URL	https://github.com/lz4/lz4/releases/

12.7.106.1. Overview

LZ4 is a compression program optimized for speed, both when compressing and when decompressing data. The homepage for LZ4 claims it can compress at 400 megabytes per second per core, and decode at rates of multiple gigabytes per second per core.

I often use lz4 for things like database backups, where compression time is a major factor in the total time to complete a backup — switching to lz4 from gzip caused the backup time to go from

about 5 hours to around 40 minutes.

12.7.106.2. lz4 (final-system-components phase)

lz4 doesn't use the GNU build system, although its Makefile follows the GNU conventions.

Configuration commands:

```
(none)
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make prefix=/usr install
```

12.7.107. lrzip

Name	Long Range Zip
Version	0.651
Project URL	https://github.com/ckolivas/lrzip
SCM URL	*(unknown)*
Download URL	*(unknown)*
Dependencies	lzo, lz4

12.7.107.1. Overview

LRZip is a compression program that is optimized for compression of large files. Often, it can compress better than `lzip` for files that are larger than about 100 megabytes in size, because it does a long-distance redundancy reduction pass before doing conventional compression. This requires a great deal of memory, especially for compression, but that might not be an issue in your particular circumstance.

This package relies on compression and decompression logic from other libraries.

Dependencies

lzo, lz4.

12.7.107.2. lrzip (final-system-components phase)

LRZip is not really necessary, but I use it to compress the QEMU disk images when I package up Little Blue Linux builds — see the `qemu-to-qemu-build` blueprint for more details.

Configuration commands:

```
./autogen.sh
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.108. lzip (final-system-components phase)

For an overview of lzip, see lzip.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.109. make (final-system-components phase)

For an overview of make, see make.

This is an almost completely standard build using the GNU build system. Really, it would be surprising if it weren't!

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

The automated tests for make are implemented in Perl, and rely on a test driver program present in the project sources. In perl 5.26, this breaks as described at https://lists.gnu.org/archive/html/bug-make/2017-03/msg00040.html. To work around the issue, we can set an environment variable as described there.

Some tests sometimes fail with timeout errors, as well; we can proceed anyway as usual.

Test commands:

```
PERL_USE_UNSAFE_INC=1 make check || echo "exit code $?, proceeding anyway"
```

Installation commands:

```
make install
```

12.7.110. patch (final-system-components phase)

For an overview of patch, see patch.

This is a standard GNU build system build.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.111. patchelf (final-system-components phase)

For an overview of patchelf, see patchelf.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Since this version of `patchelf` will be part of the final system, we should run the tests. If there are failures, though, it is as safe to proceed as usual.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

12.7.112. pixman

Name	pixman library
Version	0.42.2
Project URL	http://www.pixman.org/
SCM URL	git://anongit.freedesktop.org/pixman
Download URL	https://www.cairographics.org/releases/

12.7.112.1. Overview

Pixman is a library for pixel manipulation. It is used by the Cairo graphics library and the X server; it's also a required dependency when building the QEMU emulator.

12.7.112.2. pixman (final-system-components phase)

This is a perfectly standard GNU build system package.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

12.7.113. popt (final-system-components phase)

For an overview of popt, see popt.

One of the tests doesn't reliably pass.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?, continuing anyway"
```

Installation commands:

```
make install
```

12.7.114. pth (final-system-components phase)

For an overview of pth, see pth.

Configuration commands:

```
./configure --prefix=/usr --mandir=/usr/share/man
```

Compilation commands:

```
make -j1
```

Test commands:

```
make -j1 test
```

Installation commands:

```
make -j1 install
```

12.7.115. qemu

Name	QEMU
Version	8.1.2
Project URL	http://www.qemu.org/
SCM URL	*(unknown)*
Download URL	https://www.qemu.org/download/#source
Dependencies	python, glib, libslirp, pixman, zlib

12.7.115.1. Overview

QEMU is a computer emulator and virtualizer. It's pretty important in the CBL process, so I'm going to spend some time expanding on what that means — as with everything else in this book, feel free to skip ahead if you're not interested enough to get into the details; it won't hurt my feelings.

12.7.115.2. About Virtual Machines

Let's start with the basics: Generally, in the realm of computing, the word *virtual* means that something doesn't really exist, but it can be used as though it does exist, and *physical* means that something actually does exist.

A common example of this is memory: the actual physical RAM in your computer can be used to store data. Since RAM is relatively scarce on most systems, it's often the case that people don't have enough memory in their computers to store all the data needed by all the programs they want to run. When that happens, all modern operating systems provide the ability to use *virtual memory* as well as actual *physical memory*. When a program asks the operating system for some memory, and the operating system knows that all the RAM in the computer has already been allocated, it finds some chunks of RAM that haven't been accessed recently and writes the data from them out to a hard drive or similar persistent storage device, and then provides the chunks of RAM that have just been freed up to the requesting program. When and if a program needs data that was written out to persistent storage this way, the operating system finds *other* chunks of RAM that haven't been accessed in a while, writes *those* chunks out to the storage volume, reads the chunks of data that are now needed and stores them in the freed-up RAM, and provides them to the program that needs

them.[4]

The data that was written to persistent storage to free up physical RAM is called "virtual memory" because it can be accessed like physical memory but doesn't really exist. The only drawback to this approach is that persistent storage is several orders of magnitude slower than RAM, so when you are actively using programs that, collectively, need more memory than you have in your computer, you will sometimes find that everything gets really slow.

That's all really a digression. The important thing to understand is that *physical* resources actually exist as hardware, and *virtual* resources do not: some program or operating system is just pretending that they exist.

A virtual machine is like virtual memory, but it is a whole pretend *computer* rather than a quantity of pretend RAM. Pretending that a comptuer exists when it does not is pretty complicated, so (unlike virtual memory, which is a feature provided by the operating system) there is a whole separate program that does the work of pretending that the computer exists. That program is QEMU.[5]

12.7.115.3. About Emulation

I've spoken elsewhere about CPU architectures — different types of processors have different instruction sets and numbers of registers and so on, so programs compiled to run on Intel or AMD CPUs won't run on ARM or MIPS or POWER or Sparc CPUs. QEMU itself can be compiled on any kind of system, and it includes programs that pretend to be all sorts of computers, with all sorts of CPU architectures, so if you're on an Intel laptop computer you can use QEMU to run an Intel-architecture virtual machine, or you can run a 64-bit ARM virtual machine, or a 32-bit MIPS virtual machine, or whaever else you want. If you move to a Chromebook with an ARM CPU (running Linux or some other full-featured operating system rather than ChromeOS), you can use QEMU to run the same virtual machines on *that* physical computer.

The process of converting the instructions from the set used on one CPU to the set used on a different CPU is called *emulation*. So when QEMU is pretending to be a computer with a *different CPU architecture* than the physical computer where QEMU is running, that's called an "emulated virtual machine." Emulation is computationally difficult — for every instruction in the programs being run in an emulated VM, QEMU needs to translate it into one or more instructions that run on the physical computer's CPU, and then execute each of *those* instructions — so emulated VMs are pretty slow!

Conversely, when QEMU is pretending to be a computer with *the same* CPU architecture as the physical computer, that's simply called a "VM." In this case, no emulation needs to be done, so every CPU instruction that needs to run in a virtual-machine process can simply be run on an actual physical CPU, without any conversion or translation. That means that these processes are not nearly as slow as emulated-VM processes — there's only a minor overhead added by the virtualization — and in many cases CPUs have features that allow *accelerated* virtualization, so there is essentially no performance loss when running a program in a virtual machine compared with running it directly on the physical computer. (All modern Intel and AMD CPUs provide such a feature; Intel calls it "Intel VT" and AMD calls it "AMD-V," but in either case it just means that QEMU can run programs in non-emulated virtual machines about as fast as those programs run *outside* of QEMU. This is the most common use of QEMU — to provide cloud-style virtual machines on

powerful hypervisor servers.

12.7.115.4. How QEMU Fits Into All This

That's not what we're doing here, though. Instead, we use QEMU so that we can do CBL builds — which will, by definition, involve two distinct CPU architectures, one for the host side of the build and a different one for the target side — when we don't have a spare computer with a different machine architecture lying around.

The default CBL configuration will run the host side of the build on the physical computer where you're running it, and will then use QEMU to execute the target side of the build. An alternative process, the "QEMU-to-QEMU" build configuration, uses QEMU virtual machines for both the host side and target side of the builds — typically, one of those builds will be emulated, but in principle both sides could be running on *different* emulated CPUs.

Since QEMU provides the same virtualized hardware everywhere, it provides a lot of flexibility about what types of builds we can perform. It also lets us be a lot more confident that there won't be any weird idiosyncracies with the target system that will prevent the normal CBL process from working!

All that said, if you have multiple physical computers with different types of CPUs, you can perform the entire CBL process without using QEMU at all! In which case you can ignore this entire section.

12.7.115.5. qemu (final-system-components phase)

Dependencies

pixman, zlib.

In order for CBL to be natively self-hosting, we need to have QEMU available in the final target system. As a pleasant consequence, we'll then be able to do new CBL builds on the resulting system, targeting *other* architectures (including the original, native host system architecture), on the base CBL system without worrying about the Trustworthy Host-System Programs.

Configuration commands:

```
./configure --prefix=/usr --cc=gcc --localstatedir=/var \
  --enable-gnutls --enable-gcrypt --enable-kvm --enable-slirp
```

Compilation commands:

```
make
```

The automated test suite is problematic in the initial target system. It is *also* sometimes problematic in the final Little Blue Linux system, so QEMU is not rebuilt with tests as part of Rebuild the Packages Whose Tests Have Not Been Run.

The issue occurs when the target system is being emulated, rather than being an actual computer or native-architecture VM; in that case, there are two levels of emulation happening — for example, in an amd64-to-aarch64 CBL build running on an amd64 computer, all binaries in the target system

use aarch64 CPU instructions, so QEMU generates equivalent amd64 CPU instructions and runs those. When running the QEMU tests, all of the emulation targets are exercised to make sure they work properly, so there are tests that launch a QEMU Sparc-architecture virtual machine and exercise it. When that happens in the CBL target system, it means that QEMU on the target system generates and runs aarch64 instructions that are equivalent to the sparc CPU instructions in the target binaries. Then the QEMU running the aarch64 target system takes those aarch64 instructions, generates amd64 instructions for them, and runs *those*. That's too much translation! It's horribly, horribly slow. I don't know how long the QEMU tests usually take to run, but I got bored after waiting several hours for the very first tests to complete.

If you want to run the automated tests for the QEMU package — which, again, is only recommended in a real computer or a native-architecture virtual machine — you can run the `rebuild-untested-qemu` phase of this package.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

12.7.116. rsync (final-system-components phase)

For an overview of rsync, see rsync.

Dependencies

zlib, lz4.

As before, we disable features that rely on packages that are not part of CBL.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc --with-included-popt=no \
  --with-included-zlib=no --disable-xxhash --disable-zstd
```

Compilation commands:

```
make
```

Test commands:

```
make check || echo "Exit code $?, continuing anyway"
```

Installation commands:

```
make install
```

12.7.117. sed (final-system-components phase)

For an overview of sed, see sed.

Configuration commands:

```
./configure --prefix=/usr --docdir=/usr/share/doc/sed
```

Compilation commands:

```
make
make html
```

Starting with sed 4.5, I get an error in the `inplace-selinux.sh` test; it complains that "CONFIG_HEADER" is not defined. Everything else works, and I'm not worrying about selinux until I have a basic system working, so for now we can just ignore the error and proceed.

Test commands:

```
make check || echo "exit code $?, proceeding anyway"
```

Installation commands:

```
make install
make install-html-am
```

A few scripts that were built and installed before the final system sed is set up have the scaffolding sed path baked into them. Now that we have the real sed available, we can modify those scripts so they know where to find it.

Post-installation (as `root`) commands:

```
pushd /usr/bin
grep -l /scaffolding/bin/sed * | while read FILE; \
  do \
  sed -i -e 's@/scaffolding/bin/sed@/usr/bin/sed@g' $FILE; \
  done
popd
```

12.7.118. shadow (final-system-components phase)

For an overview of shadow, see shadow.

Dependencies

libxcrypt.

This is built pretty much the same as the scaffolding version.

The shadow package provides `groups` and `nologin` programs. There are also versions of these programs provided by GNU coreutils (for `groups`) and util-linux (for `nologin`); we only want one of each, so we disable the ones provided by this package. (This is a completely arbitrary decision on my part — I have no reason to think that the other implementations of those programs are superior to the ones provided here.)

The man-pages package includes a couple of pages that are also distributed as part of the shadow package; generally, in these cases, we prefer the version distributed along with the package.

Pre-build (as `root`) commands:

```
rm -f /usr/share/man/man3/getspnam.3 \
  /usr/share/man/man5/passwd.5
```

Some configuration files in `/etc` were created by the scaffolding version of this package, and are owned by root. The installation process for shadow will want to install them again, so they need to be owned by the package user during installation.

Pre-build (as `root`) commands:

```
chown shadow:shadow /etc/login.defs \
  /etc/limits \
  /etc/login.access
```

Configuration commands:

```
sed -i src/Makefile.in -e 's/groups$(EXEEXT) //' \
  -e 's/= nologin$(EXEEXT)/= /'
find man -name Makefile.in -exec \
  sed -i -e 's/man1\/groups\.1 //' -e 's/man8\/nologin\.8 //' '{}' \;
./configure --prefix=/usr --sysconfdir=/etc \
  --with-group-name-max-length=32 --without-libbsd
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

The default password hashing method is DES, which is very weak and only allows passwords up to 8 characters. For CBL we switch to a SHA-512 hash.

Installation commands:

```
sed -i -e 's@#\(ENCRYPT_METHOD \).*@\1SHA512@' /etc/login.defs
```

It's also convenient to create home directories for new users automatically.

Installation commands:

```
sed -i -e 's@#\(CREATE_HOME \).*@\1YES@' /etc/login.defs
```

Using the package-users scheme, UIDs and GIDs starting at 9999 are reserved for package users, so we need to adjust `login.defs` to restrict normal users to have UIDs only from 1000 to 9998 rather than the default 60000 upper bound. A more reasonable place to do this is in the package-users installation, but the installation here overwrites anything we do there.

Installation commands:

```
sed -i -e '/^[UG]ID_MAX/s@60000@ 9998@' /etc/login.defs
```

The files we chown'ed to the package user earlier should be owned by root again.

Post-installation (as `root`) commands:

```
chown root:root /etc/login.defs \
  /etc/limits \
  /etc/login.access
```

Some of the programs distributed in the shadow package need to be setuid root. The presumption made in CBL is that the programs that shadow installs as setuid should all be setuid root — if that isn't the policy you want to use, modify this as you see fit!

Post-installation (as `root`) commands:

```
find /usr/bin /usr/sbin -user shadow -a -perm -4000 | \
  while read PROGRAM; \
  do \
  chown root $PROGRAM; \
  chmod 4755 $PROGRAM; \
  done
```

And we actually want to use shadow passwords, so we need to run the conversion programs provided by this package:

Post-installation (as `root`) commands:

```
test -f /etc/shadow || pwconv
test -f /etc/gshadow || grpconv
```

The shadow password utilities allow for *subordinate* user and group identifiers. These allow normal (non-root) users to use the `newuidmap` and `newgidmap` commands to configure user IDs and group IDs, respectively, within a user namespace.[6] These programs use the files `/etc/subuid` and `/etc/subgid` to determine specifically how users are allowed to do this. These file are not currently created by the shadow package's installation routine, so we should ensure they exist.

Post-installation (as `root`) commands:

```
touch /etc/subuid /etc/subgid
```

Many of the files under `/etc` should be tracked in the configuration file repository — including all of the files we've already adjusted.

Configuration Files

- `/etc/default/useradd`
- `/etc/group`
- `/etc/gshadow`
- `/etc/login.defs`
- `/etc/passwd`
- `/etc/shadow`
- `/etc/subgid`
- `/etc/subuid`

12.7.119. tar (final-system-components phase)

For an overview of tar, see tar.

In the latest version of tar, one of the automated tests fails: `difflink.at`, a test that involves hard-linked symbolic links. The expected output is apparently `a/z: Not linked to a/y` but instead the tar command emits `a/y: Not linked to a/z`. I don't think that's worrisome enough to abort the build. As with other packages where test failures are ignored, it's a good idea to review the test logs to see whether you have other, more problematic, issues.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

The documentation doesn't get installed by default, so we need to do that separately.

Installation commands:

```
make install
make -C doc install-html
```

12.7.120. texinfo (final-system-components phase)

For an overview of texinfo, see texinfo.

Dependencies

perl, pth, ncurses.

The test suite hangs, so we will skip it for now. Texinfo is in the Rebuild the Packages Whose Tests Have Not Been Run section, so it's easy to rebuild it with the full test suite once the system is fully built.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

Some components of texinfo should be installed into a directory that will be used by TeX and METAFONT (conventionally, this is called `texmf`, which is an abbreviation of "TeX and METAFONT") if those programs will ever be installed on the system. In CBL, we presume that TeX *may* eventually be installed, so we go ahead and install those components.

(TeX and METAFONT are the main programs in a typesetting system written by Donald Knuth, initially so that his magnum opus *The Art Of Computer Programming* could be typeset properly.)

Installation commands:

```
make TEXMF=/usr/share/texmf install-tex
```

12.7.121. vim (final-system-components phase)

For an overview of vim, see vim.

The test suite for vim is really noisy — a lot of terminal controls are used. The tests also do not work properly when run non-interactively, as far as I can tell — at least, I have not been able to get them to work. If you want to run the automated tests for vim, wait until the base system is complete and running, and then manually configure and build (using the configure_commands and compile_commands functions); then unset `MAKEFLAGS` to disable parallelism, and then *manually* run `make scripttests` (which will conclude with `TEST FAILURE` if there were errors and `ALL DONE` if not) and then `make unittests` (which will compile some test programs and then run them; if the test programs run successfully, you'll see the line `passed` after each of the test programs is executed).

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

A lot of people who have been using Unix systems for a long time habitually invoke it as `vi` rather than `vim`. To make life slightly easier for those people, we set up a symbolic link.

Installation commands:

```
make install
ln -s vim /usr/bin/vi
```

We also need to create a basic configuration file. The only thing we specify in it by default is "nocompatible" mode, which makes `vim` behave less like the original `vi` editor in a number of useful ways. You might want to make additional changes to suit your preferences!

Installation commands:

```
echo "set nocompatible" > /etc/vimrc
```

12.7.122. yoyo

Name	yoyo program runner
Version	0.99.5
Project URL	https://gitlab.com/freesa/yoyo

SCM URL	https://gitlab.com/freesa/yoyo.git
Download URL	https://gitlab.com/freesa/yoyo/-/releases

12.7.122.1. Overview

Sometimes programs crash or hang due to ephemeral or transient issues. In situations where this happens — one specific case is the target side of the Cross-Building Linux process, when running QEMU to perform the final system build — it can make sense to retry the program some number of times. That's what `yoyo` does.

yoyo runs the rest of its own command line as a new subprocess. If the subprocess terminates with an error status, yoyo restarts it. If the subprocess appears hung — in other words, all threads are sleeping, and both user and system CPU time is increasing very slowly if at all, over the course of several minutes — yoyo terminates and restarts it.

After yoyo has restarted a hung or crashed process five times, it gives up and terminates itself.

12.7.122.2. yoyo (final-system-components phase)

Configuration commands:

```
(none)
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
cp -a -v build/yoyo /usr/bin
```

[1] If `pwd` is not present where Perl expects to find it, the Perl `cwd` function will fail and this will cause the build to crash with a message about being unable to find the current working directory. Another workaround to this problem is just to change the code that calls `cwd` to call the Perl function `getcwd` instead, but I prefer just adding a dependency to patching the source code.

[2] As far as I can tell, there is no license for the PKGBUILD file that prohibits this type of use of the scripts it contains. Please let me know if that's incorrect!

[3] Some of the container projects maintain scripts that examine the current kernel configuration and identify changes that are needed to support containers: in the LXC repository at https://github.com/lxc/lxc there's a script `lxc-checkconfig.in`, and the moby repository at https://github.com/moby/moby contains a script `check-config.sh`.

[4] This process is often called "swapping" because chunks of data are being swapped back and forth between RAM and persistent storage. The area of the persistent storage volume used for this purpose is sometimes called "swap space" or a "swap file." If you've ever wondered what those things are, now you know!

[5] There are other programs that serve a similar purpose, like VirtualBox and VM Ware and probably others, but the only program I'm going to use here or talk about is QEMU.

[6] You can `man 7 user_namespaces` to read more about that, if you're interested.

Chapter 13. Complete the CBL System

We have all the system components in place now, so we can set up the s6-based init system that will launch userspace.

13.1. Set Up Networking For the Target System

Networking is a really complicated topic and there is no way to cover it comprehensively here! We're only going to cover a few of the absolute basics: what kind of networking hardware the target system has, and how to configure that hardware so the target system is accessible over the network.

The default assumption made by CBL is that the target system will perceive itself as having a *wired* Ethernet interface; it will have a *dynamically-configured* IP address; and that networking should be enabled at the conclusion of the CBL build. If any of those isn't the case, override the `ENABLE_TARGET_NETWORK` parameter — the network-related services will still be defined, but not made part of the `rl-default` standard system run-level bundle.

13.1.1. Network Hardware

There are a few types of network hardware you might be dealing with.

13.1.1.1. Wired Ethernet

This is the simplest situation you can possibly be in. To get the target system on the network, all you need to do is ensure that a driver for your network hardware is compiled into the Linux kernel.[1] Easy-peasy!

13.1.1.2. Wireless

If your target system has only a wireless network interface, you need additional software that is not part of the base CBL system. The simplest set of packages you can add is:

- wireless_tools — provides programs like `iwconfig` that control wireless network hardware. Hosted at https://hewlettpackard.github.io/wireless-tools/Tools.html
- wpa_supplicant — allows your computer to connect to WPA and WEP-encrypted wireless networks. Hosted at https://w1.fi/wpa_supplicant/

As of this writing, this is beyond the scope of CBL.

13.1.1.3. QEMU Emulated

The term "network hardware" may be misleading, because there might not be any hardware involved! If the target system is a QEMU virtual machine, all the hardware it perceives is actually being simulated by the QEMU program. In this case, you configure what kind of hardware QEMU will simulate by using command-line arguments to the `qemu-system` program. Then all you have to do is ensure that the Linux kernel running on the virtual machine has support for whatever hardware you've told QEMU to simulate.

The way that I like to set this up — because it's the easiest thing for me to think about — is to give the virtual machine a TAP interface that is connected to a bridge interface on the host system where the virtual machine is running.

This is all described more fully in [setup-virtual-network], which you can also use on the host system to construct an entire virtual network usable by any number of virtual machines simultaneously.[2]

You might also be interested in the [enable-virtnet-internet] blueprint, which sets up the virtual network so that the virtual machines connected to it can reach the actual network the host system is on, and thereby reach the Internet. But while working on CBL — especially when writing or testing package blueprints — I don't do that. There are a lot of build systems that automatically fetch source code or binaries for other programs or libraries from locations on the Internet. Since my goal with CBL is to make it screechingly obvious what code is present in the system, and to enable the entire system to be built from a known repository of source code without anything extra being needed, this (normally helpful and convenient) technique is exactly counter to my purposes. Therefore, while I like having the CBL system be able to reach the host system (so I can fetch software packages from it, for example), I explicitly do *not* want it to be able to reach the Internet.

13.1.2. Configuring Networking For the Target System

Once you have a functioning network interface (real or emulated), the other thing you need to do is ensure that interface has an IP address. There are two ways you can do this: *static* and *dynamic*.

13.1.2.1. Static Network Configuration

If you know an available IP address on the local network, you can simply assign that IP address to the interface. This is most conveniently done at system startup time, so if you want to do this you can set up an s6-rc service definition directory, maybe called `eth0-network`, dependent on `mount-sys` and `mount-proc`, with an `up` script that does something like:

```
if { s6-echo "Starting network" }
if { ip link set eth0 up }
ip addr add ${TARGET_IP_ADDRESS} dev eth0
```

and a corresponding `down` script something like:

```
if { s6-echo "Shutting down network" }
if { ip addr flush dev eth0 }
ip link set eth0 down
```

Then make that service the active one in the `network` bundle so that it runs automatically as part of the `network-services` bundle, and so that other network-related services will depend on it.

13.1.2.2. Dynamic Network Configuration

In most cases, static networking isn't the best option — these days, computers are more often

portable than not, and even when they're always in the same place they may have multiple routes to the Internet. That means they often wind up on different networks at different times, which in turn means that dynamic network configuration is usually the best approach. This is done with the Dynamic Host Configuration Protocol (DHCP).

There are several different programs that implement the client side of DHCP. I have used `dhclient`, provided as part of the DHCP package from the Internet Systems Consortium, and `dhcpcd` by Roy Marples; both work fine. CBL uses `dhcpcd` by default because it is slightly more congruent with the way that supervised services are set up in CBL: both can be told not to background themselves, but `dhclient` always logs its activities via the `syslog` function, while `dhcpcd` can be told to write log messages to its standard error instead. (By default it *also* writes them to syslog, but this can be redirected to `/dev/null`).

13.1.3. libmnl

Name	Minimalistic netlink library
Version	1.0.5
Project URL	https://www.netfilter.org/projects/libmnl
SCM URL	https://git.netfilter.org/libmnl/
Download URL	https://netfilter.org/projects/libmnl/downloads.html

This library provides low-level functions to help userspace programs interact with the Linux kernel via netlink sockets; it takes care of plumbing details like constructing, validating, and parsing netlink headers. It doesn't try to hide any of the details of actually *using* netlink sockets, so you still wind up writing some pretty low-level code if you use this library.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

13.1.4. Set Up Network Availability Service

Some startup services — both long-running daemon processes and oneshot initialization

processes — need access to the network to function. Long-running processes are typically set up to keep retrying until they succeed, but — especially for oneshots — it's inconvenient for those services simply to depend on a `network` service. When that resolves to a DHCP client (as is the default configuration on CBL systems) the service will be regarded as being up almost instantly — as soon as the DHCP client program is running — but the network interface won't actually be configured until some time later. The delay may only be a few seconds, but it can be inconvenient for dependent services to fail until the network is actually available.

This blueprint installs a very simple program, `await-default-route`, and a oneshot s6-rc service that runs it. The program simply waits until there is a default IPv4 network route — that is, a route that instructs the kernel that network packets intended to be sent to other computers should be sent through a specific network interface. If there is a default route, then presumably the network is accessible through it! If there is no default route, `await-default-route` will block until that situation changes.

Any other service that needs to be able to reach external services over the network can depend on the `network-available` oneshot service that is set up here, and the s6-rc service manager will defer starting those until the `network-available` service completes.

We could implement this really easily by writing a script that just dumps the route table and looks to see if there's a default route, but for that to work the script would have to "poll" — that is, check to see if there is a default route; if there is not, then sleep a little while and check again, and keep doing that until there is a default route.

Polling is often really easy, but it's never the *best* thing to do: it wastes time and energy. Any time you can use an interrupt or notification-based mechanism, you should. That's what we're going to set up here: a tiny program that will be notified of all changes to the route table and blocks until it is told about the creation of a default route.

It uses a small library that eliminates some of the drudgery of using netlink.

Dependencies

libmnl.

The source for this program is only a couple thousand characters in size. It doesn't really make sense to package it in a tarfile, along with a Makefile and README and license text and so on, so we'll just write it out here, then compile and install it using litbuild directives.

The only drawback of this approach is that the program will wind up being owned by root rather than by a package user. It's easy enough to change that if you wish — just add commands that create a package user and so on.

I based this program on two example programs that are included in the libmnl package: `rtnl-route-event` and `rtnl-route-dump`. Both of those are in the public domain. Like the rest of the code in CBL, though, this modified program is licensed under the GNU GPL.

As usual for C programs, this one starts by including header files that define the functions and macros it calls.

File /tmp/await-default-route.c:

```
#include <stdlib.h>
#include <time.h>
#include <libmnl/libmnl.h>
#include <linux/rtnetlink.h>
```

Rather than fetching data from a netlink socket and processing it directly, we're going to use "callback" functions: we'll pass function references to `mnl_cb_run` and trust that they'll be invoked appropriately.

Here we are setting up two callback functions. The first one, `route_dump_cb`, will be used to detect a default route that already exists when the program is run, if there is one — that way, it can terminate immediately rather than waiting (possibly forever) for a new default route to be created.

A default route has "destination length" of 0 — I'm not sure exactly what "destination length" means, but by inspecting route table entries I observe that the only 0-length entry is the default route — and has other common characteristics: the default route is in the main routing table (which has ID 254), is a unicast route (type 1), and has universal scope (ID 0).

If this function is called with a payload that indicates any other route table entry, it just returns `MNL_CB_OK`. The program simply keeps running when it gets that result.

File /tmp/await-default-route.c (continued):

```
static int route_dump_cb(const struct nlmsghdr *nlh, void *data)
{
    struct rtmsg *rm = mnl_nlmsg_get_payload(nlh);
    if (rm->rtm_dst_len == 0 && rm->rtm_table == 254 &&
            rm->rtm_type == 1 && rm->rtm_scope == 0) {
        exit(EXIT_SUCCESS);
    }
    return MNL_CB_OK;
}
```

The other callback we define will handle the more typical case, where no default route exists at the time the program is executed. It will inspect route table change events. Each of these either adds or deletes a route table entry; we ignore all deletion events and check creation events with the same logic as the earlier callback.[3]

File /tmp/await-default-route.c (continued):

```
static int route_event_cb(const struct nlmsghdr *nlh, void *data)
{
    struct rtmsg *rm = mnl_nlmsg_get_payload(nlh);
    if (nlh->nlmsg_type == RTM_DELROUTE)
        return MNL_CB_OK;
    if (rm->rtm_dst_len == 0 && rm->rtm_table == 254 &&
            rm->rtm_type == 1 && rm->rtm_scope == 0) {
        exit(EXIT_SUCCESS);
```

```
    }
    return MNL_CB_OK;
}
```

Now we can write the main routine for the program. The variables we need to define are two netlink sockets — one to dump currently-existing routes, the other to track change events — along with buffers that they'll use to store data. There are also some other handy variables we need as well — I haven't done any sort of analysis for what they're for, they're just used in the example programs I based this on.

The buffer size, `8192L`, is a kludge to get this program to be compliant with the C89 standard. It should really be defined as being `MNL_SOCKET_BUFFER_SIZE`, but that leads to problems when compiling with `-std=c89` because it counts as a "variable length array" (which is not soemthing that C89 supports).

Looking at the definition of `MNL_SOCKET_BUFFER_SIZE` in `/usr/include/libmnl/libmnl.h`, it's always defined to be `8192L` or smaller. So we're just declaring it to be as big as it might possibly need to be.

File /tmp/await-default-route.c (continued):

```
int main(int argc, char *argv[])
{
    struct mnl_socket *event_nl, *dump_nl;
    char buf[8192L];
    struct nlmsghdr *nlh;
    struct rtmsg *rtm;
    unsigned int seq, portid;
    int ret;
```

We need to initialize and bind the event socket. I'm pretty sure that as soon as we do this, the kernel will start sending change events to this socket. By setting it up *first*, we can ensure that there's no window between the time we inspect the *already defined* route entries and the time we start inspecting *changes* to route entries — if there were such a window, there would be a race condition bug that could cause us to miss the creation of a default route.

That might all be completely obvious to everyone but me, but that's the kind of fiddly detail that I find easy to overlook. Concurrent programming is hard.

File /tmp/await-default-route.c (continued):

```
event_nl = mnl_socket_open(NETLINK_ROUTE);
if (event_nl == NULL) {
    perror("mnl_socket_open");
    exit(EXIT_FAILURE);
}
if (mnl_socket_bind(event_nl,
        RTMGRP_IPV4_ROUTE,
        MNL_SOCKET_AUTOPID) < 0) {
    perror("mnl_socket_bind");
```

```
    exit(EXIT_FAILURE);
}
```

Now that we are set up to capture all important change events, we can initialize and bind the dump socket. This is more complicated than setting up the event socket because, instead of just registering interest in change events, we have to compose a message we can send to the kernel to ask it for the current set of route table entries.

File /tmp/await-default-route.c (continued):

```
nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = RTM_GETROUTE;
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
nlh->nlmsg_seq = seq = time(NULL);
rtm = mnl_nlmsg_put_extra_header(nlh, sizeof(struct rtmsg));
rtm->rtm_family = AF_INET;
dump_nl = mnl_socket_open(NETLINK_ROUTE);
if (dump_nl == NULL) {
    perror("mnl_socket_open");
    exit(EXIT_FAILURE);
}
if (mnl_socket_bind(dump_nl, 0, MNL_SOCKET_AUTOPID) < 0) {
    perror("mnl_socket_bind");
    exit(EXIT_FAILURE);
}
portid = mnl_socket_get_portid(dump_nl);
```

Now we can ask for the route table entries and parse through them.

File /tmp/await-default-route.c (continued):

```
if (mnl_socket_sendto(dump_nl, nlh, nlh->nlmsg_len) < 0) {
    perror("mnl_socket_sendto");
    exit(EXIT_FAILURE);
}
ret = mnl_socket_recvfrom(dump_nl, buf, sizeof(buf));
while (ret > 0) {
    ret = mnl_cb_run(buf, ret, seq, portid, route_dump_cb, NULL);
    if (ret <= MNL_CB_STOP) {
        break;
    }
    ret = mnl_socket_recvfrom(dump_nl, buf, sizeof(buf));
}
if (ret == -1) {
    perror("error");
    exit(EXIT_FAILURE);
}
```

That's it for the pre-existing route table entries. If one of them was a default route, the callback

function will have terminated the program already, so if we've gotten to this point we know we need to start looking at the change events. If we run out of change events, the program will block until a new one arrives — which is the whole point of the program!

File /tmp/await-default-route.c (continued):

```
ret = mnl_socket_recvfrom(event_nl, buf, sizeof(buf));
while (ret > 0) {
    ret = mnl_cb_run(buf, ret, 0, 0, route_event_cb, NULL);
    if (ret <= MNL_CB_STOP) {
        break;
    }
    ret = mnl_socket_recvfrom(event_nl, buf, sizeof(buf));
}
if (ret == -1) {
    perror("error");
    exit(EXIT_FAILURE);
}
```

We shouldn't make it here — the intention of this program is that it continues to inspect route change events until it sees a new default route being created — but it's always possible that something will go wrong. If it does, close both sockets and exit with a failure code.

File /tmp/await-default-route.c (continued):

```
    mnl_socket_close(dump_nl);
    mnl_socket_close(event_nl);
    exit(EXIT_FAILURE);
}
```

That's it for the program! Let's compile and install it. We can just compile it directly to the standard program location, rather than compiling it in place and then copying it there.

Commands:

```
gcc -std=c89 -Wall -Wextra -Wpedantic -Werror -Wno-unused-parameter \
  -g -O2 -lmnl -o /usr/bin/await-default-route \
  /tmp/await-default-route.c
```

We also need to set up the oneshot service that will run the program. This is what other services should depend on.

Service Directory: Oneshot `network-available`

Dependencies	• `network`
Up script	`/usr/bin/await-default-route`

13.1.4.1. Complete text of files

/tmp/await-default-route.c

```
#include <stdlib.h>
#include <time.h>
#include <libmnl/libmnl.h>
#include <linux/rtnetlink.h>
static int route_dump_cb(const struct nlmsghdr *nlh, void *data)
{
    struct rtmsg *rm = mnl_nlmsg_get_payload(nlh);
    if (rm->rtm_dst_len == 0 && rm->rtm_table == 254 &&
            rm->rtm_type == 1 && rm->rtm_scope == 0) {
        exit(EXIT_SUCCESS);
    }
    return MNL_CB_OK;
}
static int route_event_cb(const struct nlmsghdr *nlh, void *data)
{
    struct rtmsg *rm = mnl_nlmsg_get_payload(nlh);
    if (nlh->nlmsg_type == RTM_DELROUTE)
        return MNL_CB_OK;
    if (rm->rtm_dst_len == 0 && rm->rtm_table == 254 &&
            rm->rtm_type == 1 && rm->rtm_scope == 0) {
        exit(EXIT_SUCCESS);
    }
    return MNL_CB_OK;
}
int main(int argc, char *argv[])
{
    struct mnl_socket *event_nl, *dump_nl;
    char buf[8192L];
    struct nlmsghdr *nlh;
    struct rtmsg *rtm;
    unsigned int seq, portid;
    int ret;
    event_nl = mnl_socket_open(NETLINK_ROUTE);
    if (event_nl == NULL) {
        perror("mnl_socket_open");
        exit(EXIT_FAILURE);
    }
    if (mnl_socket_bind(event_nl,
            RTMGRP_IPV4_ROUTE,
            MNL_SOCKET_AUTOPID) < 0) {
        perror("mnl_socket_bind");
        exit(EXIT_FAILURE);
    }
    nlh = mnl_nlmsg_put_header(buf);
    nlh->nlmsg_type = RTM_GETROUTE;
    nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
    nlh->nlmsg_seq = seq = time(NULL);
```

```
    rtm = mnl_nlmsg_put_extra_header(nlh, sizeof(struct rtmsg));
    rtm->rtm_family = AF_INET;
    dump_nl = mnl_socket_open(NETLINK_ROUTE);
    if (dump_nl == NULL) {
        perror("mnl_socket_open");
        exit(EXIT_FAILURE);
    }
    if (mnl_socket_bind(dump_nl, 0, MNL_SOCKET_AUTOPID) < 0) {
        perror("mnl_socket_bind");
        exit(EXIT_FAILURE);
    }
    portid = mnl_socket_get_portid(dump_nl);
    if (mnl_socket_sendto(dump_nl, nlh, nlh->nlmsg_len) < 0) {
        perror("mnl_socket_sendto");
        exit(EXIT_FAILURE);
    }
    ret = mnl_socket_recvfrom(dump_nl, buf, sizeof(buf));
    while (ret > 0) {
        ret = mnl_cb_run(buf, ret, seq, portid, route_dump_cb, NULL);
        if (ret <= MNL_CB_STOP) {
            break;
        }
        ret = mnl_socket_recvfrom(dump_nl, buf, sizeof(buf));
    }
    if (ret == -1) {
        perror("error");
        exit(EXIT_FAILURE);
    }
    ret = mnl_socket_recvfrom(event_nl, buf, sizeof(buf));
    while (ret > 0) {
        ret = mnl_cb_run(buf, ret, 0, 0, route_event_cb, NULL);
        if (ret <= MNL_CB_STOP) {
            break;
        }
        ret = mnl_socket_recvfrom(event_nl, buf, sizeof(buf));
    }
    if (ret == -1) {
        perror("error");
        exit(EXIT_FAILURE);
    }
    mnl_socket_close(dump_nl);
    mnl_socket_close(event_nl);
    exit(EXIT_FAILURE);
}
```

13.1.5. Set up the network bundle

The service that actually enables networking may be different in different cases — the presumption made by CBL is that the network is dynamically configured using the `dhcpcd` long-running service, but it might use a different DHCP program, like `dhclient`, or be statically configured with the `ip`

command, or have a more complex script that sets up networking differently depending on some runtime context.

The idiom that s6-rc uses for such variable behavior is, as with many other things, bundles: you define a bundle with the name of one of the possible services in its `contents.d` directory, and if you want to switch to a different alternative later, you change the bundle's `contents.d` (and then rebuild the s6-rc database and switch to the new one as described in the section Construct the s6-rc service database). Other things that need the service to be configured and available, but don't care how exactly that happens, should declare the bundle as their dependency, rather than the actual service provider.

In CBL we define a `network` bundle for this purpose. This is, you might notice, unnecessary complexity — we could just make other network services depend directly on the `dhcpcd` service — but this idiom of using bundles to select from any of a number of alternative service providers is important enough that I think it's important to have an example of it in the base CBL system.

Commands:

```
cd /etc/s6-rc/source
mkdir -m 0755 -p network
echo bundle > network/type
mkdir -m 0755 -p network/contents.d
```

There is another benefit of doing this: it lets us define the entire `network-services` bundle as depending on the `network` service — if we do that, we don't have to declare that each individual service in the `network-services` bundle depends on `network`. In the default CBL configuration I'm not opting to do that, because I think it's a good idea to be explicit about dependencies for all services; that's a stylistic preference, though, so if you have a different preference you might want to change your configuration.

While setting up an AWS AMI, I encountered another problem: some services need to be able to access external resources to run successfully, and marking these dependent on the `network` service is problematic because, when using `dhcpcd` (as CBL does by default), the service is considered to be started as soon as the `dhcpcd` program is running — which is generally at least a few seconds before it configures a network interface and brings it online.

It's possible to write all of the services that actually require the network to keep retrying until they succeed, but that's not an elegant approach. I prefer to set up an additional service that will simply block until the network is actually available.

Dependencies

Set Up Network Availability Service.

13.1.6. dhcpcd

Name	DHCP Client Daemon
Version	10.0.4

Project URL	https://roy.marples.name/projects/dhcpcd
SCM URL	https://github.com/NetworkConfiguration/dhcpcd
Download URL	https://github.com/NetworkConfiguration/dhcpcd/releases
Dependencies	Set up the network bundle

The dhcpcd package provides a client for the Dynamic Host Configuration Protocol. This operates as a long-running service that communicates with a DHCP server, obtains a lease on an IP address, and configures network interfaces and other related aspects of the system as directed by the DHCP server and local system policy.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc
```

Compilation commands:

```
make
```

Test commands:

```
make test
```

Installation commands:

```
make install
```

The main configuration file for dhcpcd is `/etc/dhcpcd.conf`. We don't have to declare it as a configuration file here, but it's a good idea to do so when there's a good chance you'll modify it later as part of local system policy; if you decide to install the maradns package to set up a local DNS cache, that blueprint will modify dhcpcd.conf.

Configuration Files

- `/etc/dhcpcd.conf`
- `/etc/s6-rc/source/dhcpcd-svc`
- `/etc/s6-rc/source/dhcpcd-log`
- `/etc/s6-rc/source/create-dhcpcd-fifo`

The default mode of operation for dhcpcd (when run as a daemon) is to remain in the foreground until an IP address lease is obtained, and then to fork itself into the background. This is not desirable in a system that uses process supervision, as CBL does with the s6 supervision suite, so we disable it with `--nobackground`.

We also prefer to avoid using syslog, so we redirect log output to a log FIFO with the `--logfile` directive; that means that the s6-log process needs to redirect its standard input from that fifo.

Service Pipeline: `dhcpcd` *(in bundle* `network`*)*

Service 1: Longrun `dhcpcd-svc`	
Dependencies	• `create-dhcpcd-fifo`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`/usr/sbin/dhcpcd --nobackground --logfile /run/fifos/dhcpcd eth0`

Service 2: Longrun `dhcpcd-log`	
notification-fd	`3`
Dependencies	• `remount-root-rw` • `create-dhcpcd-fifo`

Run script	`#!/bin/execline -P`
	`redirfd -r -nb 0 /run/fifos/dhcpcd`
	`emptyenv`
	`s6-log -d3 T s10000000 n10 /var/log/dhcpcd`

Post-installation (as `root`*) commands:*

```
mkdir -p /var/log/dhcpcd
```

As with other programs that log to a static filesystem location, we need to create the FIFO for it to use.

Service Directory: Oneshot `create-dhcpcd-fifo`

Dependencies	• `setup-run`
Up script	`if { s6-echo "Creating FIFO for dhcpcd logs" }` `s6-mkfifo -m 0600 /run/fifos/dhcpcd`

In order for DHCP to work, there obviously must be a DHCP server running on the local network. The [setup-virtual-network] blueprint mentioned earlier will set up a DHCP server on the host system and configure it to provide IP addresses to any virtual machine that wants one.

If you want to set up a DHCP server that's *not* restricted to just serving IP addresses to virtual machines on a virtual network, you can adapt the [setup-virtual-network] blueprint to do that.

13.2. Configure the system initialization framework

13.2.1. A Review Of The Way System Initialization Works

As that section header suggests, you've already read a lot of this stuff. Feel free to skim or skip ahead!

The job of the Linux kernel is to initialize hardware, mount the root filesystem, and then run a program, conventionally called `init`, that is in charge of starting all the userspace processes.

The `init` program is always executed with process identifier (PID) 1, and is treated differently from all other processes by the kernel: it is not allowed to terminate. [4] If the process running as PID 1 ever exits, the kernel immediately crashes. So it's important that whatever program you have running with PID 1 be very stable!

There are a variety of programs that have been designed to serve as `/sbin/init` for GNU/Linux systems. One of these, the `sysvinit` program, was the most common one used in GNU/Linux distributions for many years — it's still used today in a few distributions, like Slackware and Gentoo. (As you may recall, CBL also uses sysvinit as the `init` program for the initial scaffolding-based target system.)

`sysvinit` has many shortcomings, though, so there are several other projects that have been developed as alternatives to it.

13.2.2. An Editorial About Systemd

One of these, called "systemd," deserves special attention because it has been adopted by the majority of GNU/Linux distributions — not necessarily because the people managing those distributions prefer it, but simply because most distributions are variants or derivatives of either Debian or Redhat, and both of *those* distributions now use systemd. It is definitely the path of least resistance for GNU/Linux systems as of this writing (in 2019) — so much so that I'm going to take a few paragraphs here to discuss the two main reasons that the CBL process avoids systemd entirely, and why it uses the init framework that it does. If you're not interested in that, feel free to skip ahead — even if you tell me you've done that, it won't hurt my feelings.

The first is that I regard systemd as a festival of nightmarishly-bad design. There are any number of discussions of the design merits of systemd on the Internet — if you are interested, and can't find any, you can start here — and you can make up your own mind about the matter; I'm not going to try to persuade you here.

The other is simplicity. You'll recall that one of the main goals of CBL is to make every part of the resulting system as simple and clear as possible, and systemd is very large — it does *a lot*. The documentation page for systemd currently (as of this writing) introduces it as:

> `systemd` is a suite of basic building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system. `systemd` provides aggressive parallelization capabilities, uses socket and D-Bus activation for starting services, offers on-demand starting of daemons, keeps track of processes using Linux control groups, maintains mount and automount points, and implements an elaborate transactional dependency-based service control logic. `systemd` supports SysV and LSB init scripts and works as a replacement for sysvinit. Other parts include a logging daemon, utilities to control basic system configuration like the hostname, date, locale, maintain a list of logged-in users and running

> containers and virtual machines, system accounts, runtime directories and settings, and daemons to manage simple network configuration, network time synchronization, log forwarding, and name resolution.
>
> — What is this?, section of the systemd documentation page

That last sentence, especially, shows what I'm talking about when I say that systemd does a lot.

I also find systemd to be opaque and difficult to understand — it's hard to learn how it does things. I tried to find a clear description of the mechanics of how the systemd boot process works, for example, but gave up after half an hour. I suspect the only clear and complete documentation is the source code itself, which is not really accessible to non-programmers and is much too large for me to dig into comfortably.[5]

Bringing those points together in a single sound bite, I offer this (with a tip of the hat to Kevin Chadwick, who quoted this in a Debian email forum thread debating the merits of switching to systemd):

> I conclude that there are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies. And the other is to make it so complicated that there are no obvious deficiencies.
>
> — Sir Tony Hoare, Turing award lecture from 1980

As a separate issue, unrelated to the technical points, it also appears to me that a lot of systemd's prevalence is due to bullying or strong-arm tactics by the people who write or support it, and I object to bullying. That's one of the *minor* reasons that Little Blue Linux doesn't use systemd.

If you decide that you'd really like a systemd-based system, that's cool! But in that case, you'll likely find the most valuable thing about CBL is the narrative it offers around what the basic components of a GNU/Linux operating system are, and how they fit together; it's almost certainly easier to use some other distribution than to integrate systemd into Little Blue Linux. As of this writing, there are even options that allow you to build a complete GNU/Linux system from source code, as the CBL process does, but using systemd as the init system: both Linux From Scratch and Cross Linux From Scratch have systemd options.

In CBL, we set up the s6 suite of packages as the PID 1 init system; I find that the combination of s6 and s6-rc provides all of the advantages that systemd purportedly has over sysvinit, without any of its drawbacks. I also find that it permits the final system to be very simple, transparent, and easy to extend — which you'll recall as one of the primary design goals for Little Blue Linux.

Again considering the project documentation as an indication of how clear or obscure the program behavior is, consider the "Early preparation" section of the s6-linux-init program documentation page-- it describes the entire sequence of operations performed by the program, almost down to the individual system call level!

Tying this back to Sir Hoare's quote — the `systemd` program that is intended to run as PID 1 is 1915 kilobytes on a debian-derived system I just looked at, which puts systemd deep into "so-

complicated-that" territory. In contrast, the programs in the s6 suite of programs described here are at least plausibly in the "so-simple-that" category: the `s6-svscan` program, which runs as PID 1, is around 23 kilobytes in size, and the largest program in the entire suite of software is `s6-rc-compile`, weighing in at 43 kilobytes.

This section walks through the setup of the s6 init framework, with a narrative describing each step in the process; as always, you can learn much more about how this all fits together, and other options you can use, by consulting the documentation on the various components.

13.2.3. Overview Of The S6 Init System

The way that the s6 suite works as a full init system is a little unconventional. To discuss that, I'm going to introduce some terminology used in the s6 documentation.

There are really three stages that a computer goes through, in the lifetime of the operating system from boot time through system halt:

1. Stage one is the process of bringing the system to its normal running state, and involves mounting filesystems, configuring network interfaces, launching all the long-running processes that are supposed to be running, and things of that sort.
2. Stage two is where most time is spent. In this stage, the machine is simply running in its normal state. During this phase, the only thing that needs to be done is to monitor the long-running processes that are supposed to be running, and restart them if they crash or otherwise terminate. It may also be necessary or useful to adjust the running state of the system by stopping or starting services, as temporary changes or persistently across reboots.
3. Stage three is triggered when the system is being shut down. In this stage, all of the long-running processes are terminated (hopefully gracefully, so they can close any files they have open and so on), filesystems are unmounted, and everything is generally put into a state in which it is ready for an eventual reboot.

In most init systems (including sysvinit and systemd), the `/sbin/init` program executed by the kernel handles all three of those stages. When using the s6 suite of software, though, each stage is handled by a *different program*.

As mentioned earlier, the process run by the kernel as PID 1 is not allowed to terminate, and the system will crash if it does. However, PID 1 *is* allowed to use the `exec` system call, which *changes the program that is being run by an existing process*, without terminating that process.

So the way the CBL process sets things up is: the program run by the kernel, `/sbin/init`, only needs to ensure the system is in a specific, well-defined state, in which the s6 process supervisor and s6-rc service manager are running properly — this takes care of stage one of the system lifecycle. Then it tells the `s6-rc` program to bring the system to its normal running state.

While the system is running, s6 and s6-rc can be used to change the system state in a predictable and consistent way — stopping or restarting services, for example, or defining additional services and adding them to the normal running state of the server.

Then, when the system is eventually shut down (and powered off or rebooted), s6-rc is used to bring the system down to a minimal state. Then the s6 process supervisor again uses the `exec`

system call to replace itself with a script that performs the actual power off or reboot action.

This looks more complicated than the conventional "one init program for all stages" approach, if you only consider the number of programs that are involved from boot to shutdown; however, if you look at how complicated each of those programs actualy is, you'll find that the total amount of code is *substantially smaller* than the monolithic approach — and when you're talking about the stability of software, small and simple programs will win every time and on every axis.

13.2.4. Setting Up The Init Framework

To get started, we use the `s6-linux-init-maker` program, provided by the s6-linux-init package; it generates a directory of scripts and support files to put in the root filesystem. That set of scripts and support files becomes the basis for the `/sbin/init` initial PID 1 process.

The s6 init process uses locally-managed scripts in a specific location, `/etc/s6-linux-init/current`. If that location already exists, the initialization framework may already be set up — in which case, we don't need to do this and should bail out.

This setup is normally only done at the very end of the CBL process, though, so if we've gotten to this point in the process and there is already a `current` init directory, things are probably in a weird partially-configured state and could use some manual attention! Therefore, we bail out here with a non-zero exit code, causing the automated build process to abort entirely. If that's not necessary and the build should proceed normally, you can replace the `exit 1` here with `exit 0`.

Commands:

```
if [ -e /etc/s6-linux-init/current ]; \
  then \
  echo "s6 init directory already exists, terminating."; \
  exit 1; \
  fi
```

In CBL, we use a lot of the default settings for `s6-linux-init-maker`. However:

1. The default settings for the catch-all logger is to use TAI64N timestamps to record when messages are logged. I don't like TAI64N timestamps, so I tell it to use ISO-8601 timestamps instead.[6]
2. By default, messages sent to the catch-all logger (a process that captures all the log messages that are not caught by *other* log processes) are only written to its log directory. We specify `-1` so that these messages are also written to the system console — that makes it easier to see what's going on at a glance.
3. To help diagnose issues with the boot process, we specify the `-G` option; this enables a getty process on tty1 very early. If your target system uses a serial console, the getty process should be run on the serial device instead — `/dev/ttyS0` or `/dev/ttyAMA0` or however it shows up — so you should adjust this command appropriately. You should also get rid of the `getty-ttyS` service definition in Construct the s6-rc service database if you do that, because it does not make sense to run two getty processes on the same terminal device!
4. The desired running state of the system is identified by any arbitrary string. This defaults to

`default`. For reasons we'll discuss in a little while, we call the running state of the system `rl-default` and specify that with the `-D` option.

Commands:

```
s6-linux-init-maker -1 -t 2 -G '/sbin/agetty tty1' -D rl-default /tmp/s6i
mv /tmp/s6i /etc/s6-linux-init/current
```

The `s6-linux-init-maker` program creates a number of wrapper scripts that simply invoke other programs that are part of the s6-linux-init package — including an `init` script that should be run by the kernel as PID 1. We should copy or link these to `/sbin` so that the boot process will find them.

Commands:

```
cp -a /etc/s6-linux-init/current/bin/* /sbin/
```

13.2.5. Writing The System-Specific Init Scripts

The programs that are part of s6-linux-init, and the scripts written by `s6-linux-init-maker`, assume that there are three scripts present on the system, in `/etc/s6-linux-init/current/scripts`. An initial template for these, with some instructions about what they should contain, is provided by the s6-linux-init installation process — these templates are found in `/etc/s6-linux-init/skel`. We could simply replace those templates with our version of these scripts, but I prefer to leave the `skel` directory alone and modify the scripts in the live location. (As long as your changes to these scripts are tracked in the configuration file repository, it really doesn't matter where you adjust them.)

Two of these scripts, `rc.init` and `rc.shutdown`, are expected to start up and shut down the system, respectively — just as you'd expect from the names.

The third, `runlevel`, is expected to bring the system to a specific run-time state. The name of the script is a callback to the sysvinit package, which uses the term "runlevel" to identify the various machine states that might be desired by the system administrators. (Besides the typical running state of a server, it might for example be desirable to define a runlevel without networking enabled, or a single-user runlevel used for emergency maintenance or system recovery.)

`rc.init` execs into the `runlevel` script to bring the system to the defined initial state, and `runlevel` can also be used to change the system state using the `telinit` command (which is provided as part of s6-linux-init for compatibility with the sysvinit package). I don't use `telinit` for that purpose, since I don't mind using the `s6-rc` and `s6-svc` commands directly, but it's available if you'd like to use it to manage your own systems.

We could start with the scripts installed by [s6-linux-init-maker] and adjust them as needed, but I prefer to start from scratch. Here we'll write our own versions of these scripts in `/tmp` and then move them into place, overwriting the template versions copied from `skel`.

13.2.5.1. `rc.init`

The first of these scripts, `rc.init`, is run by `s6-linux-init` after it launches the `s6-svscan` supervisor process. Its responsibility is to bring the system to its normal running state.

In our case this is really easy to do, because we're actually going to have the s6-rc service manager do all the hard work! `s6-rc` will tell the s6 process supervisor to run all necessary one-time scripts and start long-running daemon processes, all in the right order.

That means that the only thing `rc.init` needs to do is initialize s6-rc (which is done, reasonably enough, by the `s6-rc-init` program) and then exec into the `runlevel` script, with an argument that identifies the normal running state of the system. That argument is provided as a command-line argument to `rc.init`.

s6-rc supports the idea of runlevels using its "bundle" mechanism, which can also be used for other purposes in addition to identifying a defined running state for the system. [7] The convention used in CBL for identifying *runlevel* bundles, as distinct from other service bundles, is to give them the prefix `rl-`. The default state is, hopefully unsurprisingly, `rl-default` — we specified this while running `s6-linux-init-maker`.

File /tmp/rc.init:

```
#!/bin/execline -p
if { s6-rc-init /run/service }
importas rl 1
/etc/s6-linux-init/current/scripts/runlevel $rl
```

13.2.5.2. runlevel

The runlevel script executed by `rc.init` simply executes the `s6-rc` program to tell the service manager to change the system state.

The command line arguments here are:

1. the `change` command tells `s6-rc` to modify the system state, rather than printing out information about its service database or the current system state;
2. `-v2` tells `s6-rc` to be more verbose than usual, and print an informational message for every state transition it performs;
3. `-b` uses a locking mechanism to ensure that the shutdown process will wait for any other processes using the s6-rc service database to complete — which will of course not be the case when the system is being booted, but might be the case if the state is changed subsequently using the `telinit` program;
4. `-u` specifies that services should be brought up;
5. The `-p` option is added as well, to tell `s6-rc` to "prune" the set of running services to exactly what is specified in the named runlevel service — again, this is meaningless at system boot time, but if the `telinit` program is used to change the system runlevel at some other time, some services that are normally running should be taken down; and
6. The command-line argument given to the `runlevel` script is passed through to `s6-rc`, to specify what service should be brought up. This can technically be any service defined in the s6-rc service database, but really ought to be a service bundle that defines a runlevel.

File /tmp/runlevel:

```
#!/bin/execline -P
importas rl 1
s6-rc -v2 -b -u -p change $rl
```

13.2.5.3. `rc.shutdown`

Another script, `/etc/rc.shutdown`, is responsible for performing the first stage of shutting down the system. This is equally simple, because all we have to do is tell the service manager to shut down everything that it manages.

The differences in this invocation compared to the one in `runlevel` are:

1. `-d` tells `s6-rc` to shut things down rather than starting them up; and
2. `-a` specifies that `s6-rc` should operate on all active services, rather than expecting a selection of services on the command line.

File /tmp/rc.shutdown:

```
#!/bin/execline -P
s6-rc -v2 -b -d -a change
```

After running `rc.shutdown`, the shutdown procedure simply tells the top-level s6 process (`s6-svscan`) to terminate. When `s6-svscan` receives that instruction, it runs its `finish` script — also generated by `s6-linux-init-maker` — which terminates all remaining processes, unmounts disk partitions, and performs the final poweroff, reboot, or halt operation.

13.2.6. Installing the scripts

All of those scripts should be copied into the scripts directory where s6-linux-init will expect to find them, and should of course be executable. We should also make sure they are in the configuration file repository — really, we might as well track all of the files produced by `s6-linux-init-maker`.

Commands:

```
mv /tmp/{rc.init,rc.shutdown,runlevel} /etc/s6-linux-init/current/scripts
chmod 775 /etc/s6-linux-init/current/scripts/*
cfggit add /etc/s6-linux-init/current
cfggit as-default -m 'Add s6 init scripts'
```

13.2.7. Complete text of files

13.2.7.1. /tmp/rc.init

```
#!/bin/execline -p
if { s6-rc-init /run/service }
```

```
importas rl 1
/etc/s6-linux-init/current/scripts/runlevel $rl
```

13.2.7.2. /tmp/rc.shutdown

```
#!/bin/execline -P
s6-rc -v2 -b -d -a change
```

13.2.7.3. /tmp/runlevel

```
#!/bin/execline -P
importas rl 1
s6-rc -v2 -b -u -p change $rl
```

13.3. Build and Install Entropy Adder (target-system phase)

For an overview of addentropy, see Build and Install Entropy Adder.

For the final system, `addentropy` can be compiled using the normal system compiler, and the binary can live in the standard system binary directory for programs that might be used during the boot process.

Commands:

```
gcc -std=c89 -Wall -Wextra -Wpedantic -Werror \
  -Wno-unused-parameter -g -O2 \
  -o /sbin/addentropy /tmp/addentropy.c
```

13.4. Construct the s6-rc service database

The service manager for CBL systems, s6-rc, is responsible for running one-time scripts to initialize parts of the system — say, for example, remounting the root filesystem read/write — and for telling the s6 process supervisor to start or stop long-running daemon processes. It operates on a service *database*, which by default resides at `/etc/s6-rc/compiled`. The service database is compiled from one or more "source" directories, which each have subdirectories that describe services. These service subdirectories are pretty simple, and are documented at `/usr/share/doc/s6-rc/s6-rc-compile.html`

For convenience, we'll summarize that document here.

An s6-rc service directory can be any of three types: "oneshot", "longrun", or "bundle". A "oneshot" service is basically a script that should modify the system state in some way (e.g., remount the root filesystem so it is writable). A "longrun" service is a daemon process like `sshd`. A "bundle" service isn't really a service at all, it's just a way of labeling groups of services so that they can all be

brought up or down without listing them all individually. Every service directory needs to have a file called `type` that specifies one of those three types, and other contents depending on what type of service it is:

Bundle directories must have a directory called `contents.d`, which must contain directory entries (typically empty files) that each name a service that should be part of the bundle. The bundle is essentially an alias for that list of services — which can themselves be bundles.

Other service directories may contain a subdirectory called `dependencies.d`, which works just like the `contents.d` directory for bundles but defines the other services that are required to be up in order for the service to function. (Examples: networking has to be working in order for `sshd` to run; everything that writes to the root filesystem is dependent on the root filesystem being writable.)

Oneshot directories must have a file called `up`, which must contain a single Unix command line and will be run by the execline interpreter when the service is brought up. These services may also have a file called `down`, which is similar to `up` but is run when the service is brought down.

Longrun directories must have a `run` file, which will be used as an s6 `run` script for that service, and may also contain a bunch of other files and directories that are common to s6 service directories.

To make it easy to define service directories along with the packages that are run as service, litbuild provides a directive `servicedir` that can be used to define an s6-rc service.

There's also a directive `service-pipeline` that defines a pipeline of services that can be managed as a unit. In CBL, this is most often used for logging services: a long-running daemon process runs as a service, and anything it writes to its standard error stream is written to a file by a separate logging service.

As usual, we keep things pretty simple in the basic CBL system — we set up only one source directory, `/etc/s6-rc/source`, and a couple of "runlevel" bundles that define the standard running states of the system. We created that basic structure back in the initial target system setup script, because any blueprint might define one or more service directories. In this section, all we're doing is setting up the service directories that are not tied to other packages or blueprints, and using the s6-rc programs to create the compiled service database.

Commands:

```
cd /etc/s6-rc/source
```

Everything in the s6-rc source directory should be tracked in the configuration file repository.

Configuration Files

- `/etc/s6-rc/source`
- `/etc/s6-rc/source/syslogd-svc`
- `/etc/s6-rc/source/syslogd-log`
- `/etc/s6-rc/source/klogd-svc`
- `/etc/s6-rc/source/klogd-log`

- /etc/s6-rc/source/setup-run
- /etc/s6-rc/source/set-hostname
- /etc/s6-rc/source/mount-proc
- /etc/s6-rc/source/mount-sys
- /etc/s6-rc/source/setup-dev
- /etc/s6-rc/source/mount-cgroups
- /etc/s6-rc/source/fsck-disks
- /etc/s6-rc/source/mount-disks
- /etc/s6-rc/source/swap
- /etc/s6-rc/source/remount-root-rw
- /etc/s6-rc/source/getty-tty2
- /etc/s6-rc/source/getty-tty3
- /etc/s6-rc/source/getty-tty4
- /etc/s6-rc/source/getty-tty5
- /etc/s6-rc/source/getty-tty6
- /etc/s6-rc/source/getty-ttyS
- /etc/s6-rc/source/loopback-interface
- /etc/s6-rc/source/initialize-entropy

The only additional complexity here is that networking is optional for the target system. All services that relate to enabling or using the network are in a separate `network-services` bundle, which is only part of the `rl-default` bundle if `ENABLE_TARGET_NETWORK` is set to `true` (which is its default value).

Commands:

```
if [ true = 'true' ]; \
  then \
  touch rl-default/contents.d/network-services; \
  fi
```

Now that we're done with the network configuration, let's get on with the other low-level system initialization tasks.

There are a bunch of one-shot initialization tasks to do. To keep everything as simple as possible — and to parallelize everything that can be run in parallel — we set up each task as a separate one-shot service.

The `/run` directory is used for transient system state data. It's a temporary filesystem that gets mounted at boot time by the `/sbin/init` script produced by `s6-linux-init-maker`. We want `/run` to contain a couple of standard directories that we'll use in other service scripts.

The idiom you see here, of wrapping all the commands in `if` blocks, or all commands but the last

one, is pretty common with execline scripts — the idea is, if any of the commands fails, the script immediately terminates. (This works basically like bash when you set the `-e` option, but since execline doesn't have any place to store mutable state or options there's no way for it to support that directly.)

Service Directory: Oneshot `setup-run`

Up script	`if { s6-echo "Setting up /run" }`
	`if { mkdir /run/lock }`
	`if { mkdir -m 1777 /run/shm }`
	`if { s6-mkdir -p -m 0755 /run/fifos }`

Service Directory: Oneshot `set-hostname` *(in bundle* `rl-default`*)*

Up script	`if { s6-echo "Setting hostname" }` `s6-hostname cblviton.freesa.org`

Service Directory: Oneshot `mount-proc`

Dependencies	• `setup-run`
Up script	`if { s6-echo "Mounting /proc" }` `s6-mount -o nosuid,noexec,nodev -t proc proc /proc`

Service Directory: Oneshot `mount-sys`

Dependencies	• `setup-run`
Up script	`if { s6-echo "Mounting /sys" }` `s6-mount -o nosuid,noexec,nodev -t sysfs sys /sys`

The `/dev` directory starts out with only the device files placed there by the kernel; we need to set up some additional mounts that are expected to be under `/dev` and create some symbolic links that are expected to exist there by various programs.

The `/dev/pts` filesystem is used for pseudoterminals — device files that act like terminals, for use by terminal emulators like xterm and remote-access programs like ssh.

The other filesystem we're mounting here, `/dev/shm`, is used for various types of inter-process communication; "shm" stands for "shared memory," one type of inter-process communication facilitated by having this location available.

Service Directory: Oneshot `setup-dev`

Up script	`if { s6-echo "Setting up /dev" }`
	`if { mkdir -p -m 0755 -- /dev/pts }`
	`if { s6-mount -t devpts -o noexec,nosuid,gid=0,mode=0600 devpts /dev/pts }`
	`if { mkdir -p -m 0755 -- /dev/shm }`
	`if { s6-mount -t tmpfs -o noexec,nosuid,nodev,mode=1777 none /dev/shm }`
	`if { s6-ln -s /proc/self/fd /dev/fd }`
	`if { s6-ln -s /proc/kcore /dev/core }`
	`if { s6-ln -s loop0 /dev/loop }`
	`if { s6-ln -s ram0 /dev/ram }`
	`if { s6-ln -s rtc0 /dev/rtc }`
	`if { s6-ln -s fd/0 /dev/stdin }`
	`if { s6-ln -s fd/1 /dev/stdout }`
	`if { s6-ln -s fd/2 /dev/stderr }`

There are a number of control-group filesystems that some programs expect to be mounted in a specific structure. Here, we automatically mount the control-group filesystems known by the current kernel by parsing the `/proc/cgroups` file. This could be considered fragile, since any change in the format of `/proc/cgroups` has a chance to break this service script, but the format of files in `/proc` is pretty stable.

This script uses the `pipeline` program provided as part of the execline language, which sets up something equivalent to bash pipelines. The script below is the execline equivalent of:

```
cat /proc/cgroups | s6-tail -n +2 | s6-cut -f1 | s6-grep -vF -- devices
```

In other words: read the `/proc/cgroups` file, discard the first line, discard everything after the first tab character of each line, and discard the "devices" line. The result is a list of cgroup names. This is then provided to the `forstdin` program, which runs the rest of the script as a loop with one line of the cgroup name list as an environment variable `i`.

Service Directory: Oneshot `mount-cgroups` *(in bundle* `rl-default`*)*

Dependencies	• `mount-proc` • `mount-sys`

Up script	`if { s6-echo "Mounting cgroup filesystems" }`
	`if { s6-mount -wt tmpfs cgroup /sys/fs/cgroup }`
	`redirfd -r 0 /proc/cgroups`
	`pipeline { s6-tail -n +2 }`
	`pipeline { s6-cut -d"\t" -f1 }`
	`pipeline { s6-grep -vF -- devices }`
	`forstdin -d"\n" -- i`
	`importas -u i i`
	`if { s6-mkdir /sys/fs/cgroup/${i} }`
	`s6-mount -t cgroup -o ${i} -- cgroup /sys/fs/cgroup/${i}`

We wrote an `fstab` file back when we were starting up the target in preparation for the target side of the build. Let's set up the services that use it. Note that the environment for startup scripts defines `PATH` as just including the `/usr/bin` directory — that means that programs in `sbin` directories

must specify the full path, or change `PATH` appropriately (in execline, this is typically done with the `export` program).

For the `fsck` service, we need to do the latter, since `fsck` will try to run other programs like `fsck.ext4` and it must be able to find those. (Because of the way that the execline language works, we have to either do this first, before the `if` command that follows it, or *also* specify the full path to `fsck` — working out exactly why this is the case is a useful exercise in understanding how execline works!)

Service Directory: Oneshot `fsck-disks`

Dependencies	• `setup-dev`

Up script	`export PATH /usr/bin:/usr/sbin`
	`if { s6-echo "Checking filesystems" }`
	`fsck -r -A -n`

Service Directory: Oneshot `mount-disks` (in bundle `rl-default`)

Dependencies	• `fsck-disks`
Up script	`if { s6-echo "Mounting filesystems" }` `mount -n -a`

Service Directory: Oneshot `swap` (in bundle `rl-default`)

Dependencies	• `setup-dev`

Up script	`export PATH /usr/bin:/usr/sbin`
	`if { s6-echo "Enabling swap devices" }`
	`swapon -a -e -v`

Service Directory: Oneshot `remount-root-rw` (in bundle `rl-default`)

Dependencies	• `fsck-disks`
Up script	`if { s6-echo "Remounting root filesystem read-write" }` `mount -o remount,rw /`

The C standard library provides a function, `syslog`, that many userspace applications use to log messages; these are then made available on a domain socket, `/dev/log`, which is *typically* read by a daemon program called `syslogd`; syslogd then writes the log messages to any of a variety of output locations, which can be configured however the system administrator wishes.

In CBL, we prefer to have each daemon process write to an individual logging service, rather than using the syslog function — when possible, we run daemon processes with whatever options cause them to write their log messages to standard error, then set up a service pipeline that sends their standard error to an `s6-log` process. Some programs have no such option and insist on using the `syslog` function, though, so we also set up a pair of lightweight s6-managed services that do the same thing that `syslogd` normally does — but, for simplicity, writing everything to a single s6-log-

managed directory, rather than writing to different locations based on the syslog "facility" to which the message is assigned.[8]

Aside from writing to log files, syslogd can also be configured to send log messages across a network, or dispose of them in other ways. In CBL, though, we strive for the simplest possible configuration, and in this case that means having no syslog daemon at all. s6 provides a program specifically for handling syslog messages.

Service Pipeline: `syslogd` *(in bundle* `rl-default`*)*

Service 1: Longrun `syslogd-svc`	
notification-fd	`3`
Dependencies	• `setup-dev`

Run script	`#!/bin/execline -P`
	`fdmove -c 2 1`
	`emptyenv`
	`s6-envuidgid nobody`
	`s6-socklog -d3 -U -t3000`

Service 2: Longrun `syslogd-log`	
notification-fd	`3`
Dependencies	• `remount-root-rw`

Run script	`#!/bin/execline -P`
	`s6-setuidgid syslog`
	`emptyenv`
	`s6-log -d3 T s1000000 n10 /var/log/syslogd`

Commands:

```
mkdir -p /var/log/syslogd
chown syslog:syslog /var/log/syslogd
```

Similarly to `syslog`, messages logged by the Linux kernel using the `printk` function are made available in userspace through a file in the `proc` filesystem called `kmsg`. Often, a daemon program called `klogd` is used to write those messages to a log file. Once again, CBL uses a pair of lightweight s6 services for this purpose, and directs the log messages to an s6-log directory.

As soon as we turn on klogd, we can disable printing kernel messages to the console with `dmesg --console-off`. Kernel messages will then only show up in `/var/log/klogd/current`.

Service Pipeline: `klogd` *(in bundle* `rl-default`*)*

Service 1: Longrun `klogd-svc`

Dependencies	• `mount-proc`

Run script	`#!/bin/execline -P`
	`if { dmesg --console-off }`
	`fdmove -c 2 1`
	`redirfd -r 0 /proc/kmsg`
	`emptyenv`
	`ucspilogd`

Service 2: Longrun `klogd-log`	
notification-fd	`3`
Dependencies	• `remount-root-rw`

Run script	`#!/bin/execline -P`
	`s6-setuidgid klog`
	`emptyenv`
	`s6-log -d3 T s1000000 n10 /var/log/klogd`

Commands:

```
mkdir -p /var/log/klogd
chown klog:klog /var/log/klogd
```

We usually want a few getty processes — these provide the login prompt that lets you use virtual terminals to interact with your system. We don't start one on tty1, because that's done earlier in the boot process (as the "early getty" in Configure the system initialization framework).

Service Directory: Longrun `getty-tty2` *(in bundle* `rl-default`*)*

Run script	`#!/bin/execline -P` `/sbin/agetty tty2`

Service Directory: Longrun `getty-tty3` *(in bundle* `rl-default`*)*

Run script	`#!/bin/execline -P` `/sbin/agetty tty3`

Service Directory: Longrun `getty-tty4` *(in bundle* `rl-default`*)*

Run script	`#!/bin/execline -P` `/sbin/agetty tty4`

Service Directory: Longrun `getty-tty5` *(in bundle* `rl-default`*)*

Run script	`#!/bin/execline -P` `/sbin/agetty tty5`

Service Directory: Longrun `getty-tty6` *(in bundle* `rl-default`*)*

Run script	`#!/bin/execline -P` `/sbin/agetty tty6`

In addition to the getty processes on virtual terminals, we should run a getty process on the serial device. This will only be used if there is a terminal connected to the serial port, but if the system is a virtual machine run through QEMU with `-nographic` (as I often do), the standard input and output of the QEMU process will be tied to the simulated serial port.

In this case, we're also passing the string `tty` as the `TERM` argument to `agetty` — this sets the TERM environment variable to a very dumb terminal, which prevents the shell from printing control sequences that more intelligent types of terminal understand. This avoids potential problems when running a virtual machine with `-nographic` and controlling it using a program like `expect`, as (for example) the `automated-qemu-to-qemu-build` blueprint does.

Service Directory: Longrun `getty-ttyS` *(in bundle* `rl-default`*)*

Run script	`#!/bin/execline -P` `/sbin/agetty ttyAMA0 tty`

It's always a good idea to set up the loopback network interface. This is not a real interface at all; it's a fiction invented by the kernel that allows programs on the computer to communicate with each other as though through a network, even without any actual network hardware.

Service Directory: Oneshot `loopback-interface` *(in bundle* `rl-default`*)*

Dependencies	• `mount-sys` • `mount-proc`

Up script	`export PATH /usr/bin:/usr/sbin`
	`if { s6-echo "Configuring the loopback interface" }`
	`if { ip addr add 127.0.0.1/8 dev lo }`
	`ip link set lo up`

When the `rngd` daemon works, it's the best way to ensure that the computer's entropy pool is always initialized. But on some machines, `rngd` doesn't have any source of unpredictable data; for these cases, the `addentropy` program can be used to save some random data at system shutdown time and read it into the entropy pool again at boot time.

Dependencies

Build and Install Entropy Adder (target-system phase).

Commands:

```
dd if=/dev/urandom of=/var/cache/saved_entropy bs=512 count=2
chmod 600 /var/cache/saved_entropy
```

Service Directory: Oneshot `initialize-entropy` (in bundle `rl-default`)

Dependencies	• `coldplug`
Down script	`if { s6-echo "Caching entropy from current pool:" }` `dd if=/dev/urandom of=/var/cache/saved_entropy bs=512 count=2`

Up script	`export PATH /usr/bin:/usr/sbin`
	`if { s6-echo "Initializing entropy pool from cache:" }`
	`redirfd -r 0 /var/cache/saved_entropy`
	`addentropy`

That's a pretty bare-bones set of services, but it's good enough to start with!

s6-rc actually operates on a *compiled* service database, so we need to create that.

Don't get confused by the `-v2` and `-v1` stuff in these commands! The `-v2` argument to `s6-rc-compile` and, later, `s6-rc-update` makes those commands more verbose than they are by default. The `-v1` suffix to the `compiled` directory is a version number that will be incremented when we make a change to the service database in the running target system.

Commands:

```
s6-rc-compile -v2 /etc/s6-rc/compiled-v1 /etc/s6-rc/source
ln -s /etc/s6-rc/compiled-v1 /etc/s6-rc/compiled
```

13.4.1. Modifying the Service Database

Once the system is live and running, the basic process for modifying the service database is:

- modify the service definitions in `/etc/s6-rc/source` however you like (and add those changes to the configuration file repository);
- run `s6-rc-compile -v2 /etc/s6-rc/compiled-vN /etc/s6-rc/source`, where `N` is the next version number;
- run `s6-rc-update -v2 /etc/s6-rc/compiled-vN` (using the full path of the new compiled database) so that s6-rc switches its *active* service database to the new one;
- if new services were added to the `rl-default` bundle, run `s6-rc -u change rl-default` to activate the new services; and, finally,
- if everything looks satisfactory, recreate the `/etc/s6-rc/compiled` symbolic link so it points to the new compiled directory, with a command like `ln -s -f -n /etc/s6-rc/compiled-vN /etc/s6-rc/compiled`.

Using a versioned database directory is a good idea because `s6-rc` expects to have exclusive ownership of whatever active service database it's told to use — you must not make any changes to that directory structure. By compiling into a new database directory every time and switching to it, you can do a basic sanity check on the service database and then configure the system to use it at the next boot by changing the `/etc/s6-rc/compiled` symbolic link; and if it turns out to have problems, you can revert to an earlier version of the service database by changing where the

`compiled` symlink again. I usually keep a half-dozen or so compiled service databases around — they use less than a megabyte each of storage.

If `s6-rc-update` can't figure out what needs to be done to modify the running state of the system — for example, if you've changed the name of a service — you can provide a "conversion file" that resolves the ambiguities. You can read all about that in the `s6-rc-update` documentation.

After the `s6-rc-update` command completes successfully, you can remove the old `compiled` directory: `s6-rc-update` changes the *active* or *live* database directory, so the old one is no longer in use by s6-rc and can be destroyed. On the other hand, if you think you might want to switch back to the previous service database, you can keep it around!

13.4.2. Manipulating the system state

The `s6-rc` program lets you start and stop programs and subsystems. You can stop a daemon process and its associated s6-log process by running `s6-rc -d change`, passing the service pipeline name as an argument, and restart it with `s6-rc -u change`, again with the service pipeline name.

You can also get a list of all services that s6-rc considers to be active with `s6-rc -a list`.

If you want to send a signal to a supervised process without restarting it — say, for example, that you'd like to send a `SIGHUP` to an nginx process to cause it to reload its configuration file — you can do that with the `s6-svc` program. The s6 scan directory containing the service directories is located at `/run/service`, so sending a `SIGHUP` to nginx is as simple as `s6-svc -h /run/service/nginx-svc`.

13.4.3. Adjustments that you might want to make to the service database

The set of services you need on your target system might vary from the default services set up for CBL.

If your target system is an emulated virtual machine and won't ever have a console, you will probably want to skip all the `getty` processes. You might also want to change the boot `console` directive, and the early `getty` set up by s6-linux-init, to a serial device that will actually be accessible!

Similarly, if your target system lacks any kind of hardware random number generator, the `rngd` service won't be useful for you, and you should get rid of it.

Most of the other services are really pretty important, so it's a bad idea to remove them unless you know for sure it's the right thing to do.

And, of course, once you have your target system doing something useful, you'll probably want to add more services to one bundle or another!

Before this point, the CBL build process is pretty cleanly restartable! If something fails, you get dropped to a bash shell and can look at log files, figure out if it was a transient issue or something more significant, optionally clean up whatever partially-completed work is remaining from the script that failed, and then `exec /scaffolding/target-side-build.sh` to resume the build process.

That stops working well at this point, though! If the automated build fails once you get here, the

most reliable way to proceed is by running commands or scripts manually.

The boot loader needs to be installed and configured, unless the system will be booted through some other mechanism (for example, by using QEMU as a boot loader).

13.5. Set up the system boot loader

When running CBL systems as QEMU virtual machines, QEMU itself can act as the boot loader — it loads the Linux kernel and transfers control to it. That's a little bit messy, because the kernel has to be available outside of the virtual machine disk image. A tidier approach is to set up a boot loader in the virtual machine disk image itself so that the virtual machine can boot the same way an actual computer does.

You also need to set up a boot loader if you want to run CBL on a real computer rather than a virtual machine! In that case, the approach you need to follow depends on the type of computer you're using.

For Intel-architecture computers, there are two options: you can set up the boot loader in the master boot record of a storage device, as the primary boot loader for the computer; or, alternatively, you can set up the CBL boot loader in the first sector of a *partition* on a storage device, and use an existing primary boot loader on the computer to chain-load the CBL boot loader.

(Actually, there is also a third option: you can skip the boot loader for CBL entirely, and configure an existing primary boot loader to boot CBL rather than the operating system it currently boots. If you're setting up a CBL system on a computer that already has some other GNU/Linux system installed on it, this is probably the easiest thing to do.)

There are a variety of boot loaders available for Linux, and each one is installed by a separate blueprint. The blueprints distributed with CBL are for GRUB (the GNU Grand Unified Boot Loader); "manual", which doesn't actually do anything at all; and "rpi4", which installs the Raspberry Pi Model 4 boot loader. A configuration parameter can be used to select which boot loader to use.

13.5.1. Set up some non-standard boot loader

Setting up a boot loader for a GNU/Linux system can be a complicated topic — there is a pretty standard process for x86-architecture desktop or server systems, but there are so many different variations when your target environment is an embedded system or alternate architecture that there's no way to have an exhaustive set of blueprints that cover every contingency.

This option is provided when that's the situation you're in: you need to do something to set up a boot loader, but CBL has no blueprint that describes that process.

Alternatively, you may not need to set up a boot loader at all! If you are using QEMU to run your target CBL system, for example, QEMU can act as a boot loader perfectly well, so you don't need to do anything at all to get the minimal target system running.

Beyond that, though, you will need to set things up yourself — compile and install a boot loader, or (if your target system already has a boot loader) add a configuration stanza for your CBL system, or whatever.

Once you have things working, you could write a new blueprint that describes the process you used to get your CBL system running!

We should also create a non-root user and permit that user to run privileged commands via sudo.

13.6. Create a Non-Root User Account

It's really bad practice to use the root account for everything. (Really, after setting up a system, it's not *great* practice to use the root account for *anything*, but that's not a topic I want to get into in depth here.) Accordingly, CBL systems always create a non-root user that has the ability to run privileged commands under `sudo`. There are various CBL configuration parameters that allow this account to be customized — `LOGIN` and `LOGIN_FULL_NAME` — which you might want to override in your configuration file.

Users are added with the `useradd` program. The file `/etc/login.defs` controls a lot of its default behavior; the only options we're going to specify explicitly are `-c` to set the user's full name as the value of the comment field, `-G` to add the user to the `wheel` group, and `-m` to create a home directory for the user. And any time we create an account, we need to commit the change to the configuration file repository.

When creating the home directory, `useradd` will copy everything from the `/etc/skel` directory into the new home directory. It's convenient to have a `.ssh` directory automatically created with the correct permissions, so we can set that up.

Commands:

```
mkdir -p /etc/skel
mkdir -p -m 700 /etc/skel/.ssh
cfggit add /etc/skel
```

We want the user to have a password, because otherwise it won't be possible to ssh into the CBL system as that user — we could reconfigure `sshd` to permit unauthenticated logins, but that's poor practice. Here, we simply set the password to be the string `lbluser`. That's not very secure, so you might want to adjust the command to suit your tastes.

Commands:

```
grep -q ^lbl: /etc/passwd || useradd -c \
  'A Little Blue User' -G wheel -m lbl
echo 'lbl:lbluser' | chpasswd
cfggit stageall
cfggit as-default -m 'Add a non-root user'
```

The `sudo` program is controlled by the file `/etc/sudoers`. By default, `sudoers` contains a directive that also loads all the files in the directory `/etc/sudoers.d`; it's more convenient to put directives in files there than to edit the `sudoers` file in place, so that's what we do here. The directive we add tells `sudo` that members of the `wheel` group are allowed to run any command as any user.

CAUTION

The way that `sudo` uses the sudoers file (and files included into it) is a little counter-intuitive. If a command line matches more than one directive in the sudoers file, the *last matching directive* is the one that determines how `sudo` will work, regardless of how specific or general the directives are. That means that if you want to allow members of the `wheel` group to run any command by entering their password — the default — and also allow them to run some commands *without* passwords, you have to ensure the `NOPASSWD` directives are *later in the sudoers file*.

This convention — using the group name `wheel` to indicate which users are allowed to run commands with extra privilege — originated in the TENEX operating system, which has a special "wheel bit" that can be set on the accounts of "big wheel" users.

I'm a little bit tempted to tweak that convention for CBL and use a different name for the group — like maybe `burgundy`, since it's for users who are kind of a big deal. But no…

Commands:

```
mkdir -p /etc/sudoers.d
echo '%wheel ALL=(ALL) ALL' > /etc/sudoers.d/00-wheel
cfggit add /etc/sudoers.d/00-wheel
cfggit as-default -m 'Add sudoers directive for wheel'
```

All users can have individual personal bash configuration files — `.profile` or `.bash_profile` for login shells and `.bashrc` for non-login shells — and it's common for those to be pretty tricked out. The `bash` shell program also executes the commands found in system-wide configuration files. The `bash` manual page describes all this in the `INVOCATION` section, so if you want to know how this works in detail you can look there.

Let's go ahead and write those system-wide files. Here we are staying pretty minimalistic.

File /etc/profile:

```
# /etc/profile: system-wide .profile for Bourne-compatible shells
export PATH=/usr/bin
export PS1='$ '
if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      source $i
    fi
  done
  unset i
fi
source /etc/bash.bashrc
```

The `profile` we wrote automatically sources all of the files it finds in `/etc/profile.d` that end in `.sh`. Some packages, like gawk, create scripts there.

A lot of packages install files that help bash to complete commands usefully — for example, the kmod package installs a completion script for the `kmod` program; if you load it, you can type `kmod s` and hit the tab key and the shell will complete the command as `kmod static-nodes`. Very handy if you use the command line a lot and prefer to type as little as possible. We can load all those scripts in the systemwide `bash.bashrc` script — we have to do that in non-login shells as well as login shells because the completion rules aren't inherited by subprocesses.

File /etc/bash.bashrc:

```
# /etc/bash.bashrc: system-wide .bashrc for Bourne-compatible shells
for i in /usr/share/bash-completion/completions/*
do
    source $i
done
unset i
```

Configuration Files

- `/etc/profile`
- `/etc/profile.d`
- `/etc/bash.bashrc`

13.6.1. Complete text of files

13.6.1.1. /etc/bash.bashrc

```
# /etc/bash.bashrc: system-wide .bashrc for Bourne-compatible shells
for i in /usr/share/bash-completion/completions/*
do
    source $i
done
unset i
```

13.6.1.2. /etc/profile

```
# /etc/profile: system-wide .profile for Bourne-compatible shells
export PATH=/usr/bin
export PS1='$ '
if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      source $i
    fi
  done
  unset i
fi
source /etc/bash.bashrc
```

And, last but not least, we can get rid of references to the `/scaffolding` directory in the package-users shell initialization scripts.

13.7. Prepare To Tear Down The Scaffolding

In our construction metaphor, the programs and libraries we built in the `/scaffolding` area were explicitly intended to be present only temporarily, until the building itself — the CBL target system — as complete enough to stand on its own. That's now the case, so we can prepare the sysetm to have the scaffolding removed entirely.

We're not actually going to delete the `/scaffolding` directory quite yet! We *could* do that, because it's no longer necessary, but at this point it is still in use: the system that's currently running was booted from the `/scaffolding` userspace and therefore the `init` program running in it is the one found in `/scaffolding`. It's convenient to be able to tell that `/init` process to reboot the system, but the program you use to do that also lives in `/scaffolding`. If we delete it, we won't be able to run it.[9]

We'll still want the CBL blueprints to be available after removing the scaffolding. These can reside pretty much anywhere; by convention, I put them in the `root` user's home directory.

Commands:

```
mv /scaffolding/cbl /root
```

Going forward, we can just use `/tmp` as a staging directory for new versions of source code and patches, so we can reconfigure the root user's environment so that will be done.

Commands:

```
sed -i -e 's@/scaffolding/materials@/tmp@g' /root/.bash_profile
```

We also won't need to set all the CBL parameters to specific parameters for every root shell any longer — we only need the ones we wrote to `.bash_profile` originally.

Commands:

```
sed -i -e '/source.*cblrc/d' /root/.bash_profile
```

We also want to get rid of references to the scaffolding in the package users environment.

Commands:

```
sed -i -e '/scaffolding/d' /etc/pkgusr/bash_profile
cfggit stageall
cfggit as-default -m 'Remove scaffolding from pkgusr profile'
```

The string "/scaffolding/" actually still appears in some of the programs and libraries on the system, in ELF sections related to debugging, but that won't cause any issues. If you don't plan to debug

programs using `gdb`, you can remove symbols from the programs and libraries using the `strip` program provided by binutils; this saves a few megabytes and removes those last lingering references to the `/scaffolding` directory. If you want to do that, you may find that the `strip-binaries` blueprint is helpful.

The configuration file repository now has all of the commits that it will have for the initial CBL system, so this is a convenient time to run the `gc` process to pack up all the loose objects in that repository.

Commands:

```
cfggit gc
```

13.8. Next Steps: Maintaining Your Little Blue Linux System

Now that you have a base LB Linux system, you might be considering how to make use of it. The base system is really only useful as a foundation to build on: it's self-hosting, so you can use it to bootstrap new GNU/Linux systems, and it has plenty of development tools you can use to construct other packages or programs, but it doesn't really *do* anything.

There are a few options for how to build additional functionality into your LB Linux systems; we'll discuss some of them here.

13.8.1. Write litbuild blueprints

This is our preferred approach, of course! Any time we add additional software to a LB Linux system, we do it by writing new litbuild blueprints, in the same style as the blueprints that are part of the basic CBL process; then we use the litbuild program to produce scripts for those blueprints and run them to build and install the software.

The blueprints we write like that are included in the CBL repository, just as the blueprints for the basic system are. For example, the freesa.org mail server is a Little Blue Linux system with the `freesaorg` blueprint added to it; that section adds and configures the Dovecot IMAP server and Postfix MTA. You can browse through all the blueprints in the CBL repository; if any of them seem interesting or useful, it's pretty trivial to use them — as-is, or with enhancements or modifications to suit your own taste.

If you write your own blueprints to describe how to build and configure additional software components, please consider submitting those as patches to the CBL project! The more pieces of the free software ecosystem that have clear and comprehensible build stories, the better off we all are.

13.8.2. Add new packages as package users

If it's too much trouble to write litbuild blueprints, you can still use the package-users framework to set up new packages. Just use the `add_package_user` script to create a new package user, and then update the `options` file as necessary to document the build process for that package.

Again, please consider submitting those `options` files to the CBL project, especially when the process of building a package is complicated or not straightforward. That way, we'll have an easier time of writing blueprints for those packages if we ever wind up needing them.

13.8.3. Just install whatever you want

LB Linux systems are, of course, GNU/Linux systems; you can simply ignore litbuild and package-users and all the other weird CBL stuff and install whatever you like. The normal GNU/Linux convention of installing stuff into `/usr/local` or into `/opt` directories works perfectly well on LB Linux systems! Binary packages, when those are provided, will work the same on LB Linux as on any other distribution.

We don't manage our own LB Linux systems that way — we *like* having package users and a git repository for all our configuration files and the ability to see which package is responsible for a file just by running `ls -l` — but it won't hurt our feelings if you decide to ignore all that and simply use Little Blue Linux because it's lightweight and has fairly modern versions of all the standard software components that are found on GNU/Linux systems.

13.9. The End of the Process

The Cross-Building Linux process is complete, and we now have a complete GNU/Linux system that we built entirely from source code!

What would you like to do now?

If you configured the CBL build to let you install a boot loader manually, this is a good time to do that. (Alternatively, if you're going to run the target system as a virtual machine in QEMU, you can let QEMU itself serve as your boot loader — but in that case you'll need to copy the linux kernel from `/boot` to the host system.)

If you configured the CBL build not to enable the standard networking configuration (wired network, DHCP), you'll probably want to set up whatever networking you *will* want to use.

After booting the final system, you can clean up the scaffolding area by running, as root,

```
rm -rf /scaffolding
```

And you can also use litbuild to rebuild the packages whose test suites were skipped, with Rebuild the Packages Whose Tests Have Not Been Run.

You can save some space in the final system by stripping debug symbols from programs and libraries using `strip`. (CBL includes a blueprint `strip-binaries` for this purpose.)

You may want to build other packages, like the X Window System and a web browser. The basic system built here is only really suitable as a development server.

[1] You also have to ensure that the kernel has support for TCP/IP networking and various other things, but everything you need is enabled by default so you generally don't have to worry about it.

[2] Since the virtual network is IPv4 it's not really *any number* of virtual machines. In the default configuration you can have only

up to 253 virtual machines running simultaneously on the host server.

[3] That logic could easily be factored into a separate function to increase the clarity of the program, but there's no real point in fussing about the clarity of *this* program: I'm certainly not expecting it to be a long-term maintenance project.

[4] Also, `init` is assigned ownership of all orphaned processes, but that's not important right now.

[5] As of systemd version 242, the latest version as of August 2019, the `src` directory of the distribution is over twenty megabytes in size.

[6] TAI refers to "Temps Atomique International," which is the same as UTC except that UTC occasionally has leap seconds added to it while TAI does not. TAI64N timestamps are just hexadecimal numbers that represent the number of nanoseconds since the TAI epoch. You can read more about TAI64N on Dan Bernstein's web site: `http://cr.yp.to/libtai/tai64.html`.

[7] You can read more about bundles in the Construct the s6-rc service database section.

[8] If you would like to write different types of log message to different locations, this can trivially be done by extending the s6-log command line, since s6-log can direct messages to multiple locations based on regular expressions. I don't bother with this because I always configure services not to use syslog at all when I can!

[9] This could probably be done with a little ingenuity, but it is a lot easier just to defer the removal of the scaffolding until the final CBL system is booted.

Chapter 14. When The Build Crashes Or Breaks

This happens.

Sometimes it's the result of updating a package version, or even updating a branch-update patch — new versions of GCC, for example, often add new compiler warnings that will crash any build that uses `-Werror` to turn warnings into errors. Sometimes it's because build processes are fragile and don't always run to completion on the first attempt. Sometimes it's because of changes to the CBL blueprints that introduce bugs into the process itself.

Regardless of the cause, when a command fails unexpectedly you'll see a bunch of `DISMAL FAILURE` messages followed by a `#` root shell prompt: when a litbuild-generated script fails, it `exec`s into a bash process. The very first thing you'll want to do is `source /scaffolding/etc/initscript`, which will set environment variables like `PATH` so you can actually do some poking around to see what happened.

After that… since there are so many different reasons a CBL build might not run to completion, it's difficult to come up with a recipe for what you should do when it crashes. What I do is look at the most recent log messages to see if there's any indication of what the problem was; if there's no obvious reason that the build would have failed, I just restart the step that failed and see if it gets further that time. If there are any error messages in the log files, I search for those error messages on the Internet and see if anything interesting turns up.

Cleaning up the build artifacts left over from the step where the build failed — basically the `/tmp/cblwork` directory — can be a good idea, unless it is *really* late in the process: once you've reached the Complete the CBL System section, you need to pay a lot more attention to the state of the system and possibly just finish the process manually. If you clean out `/tmp/cblwork`, you should probably also remove the package user — that's `cfggit reset --hard HEAD` to throw away configuration file changes including the creation of user database records for the package user, and then `rm -rf /usr/src/USERNAME` to get rid of the package user home directory entirely.

Once you find and correct an issue, you can resume the build more or less at the right spot by running `exec /scaffolding/target-side-build.sh`. (That's assuming that the very first part of the build succeeded — if the build process crashes while running the `startup-target-system.sh` script, that's really weird and definitely problematic.)

There's one more category of problem that needs to be treated differently. When running a build on a virtual machine, sometimes bugs in the QEMU program cause it to crash with a segmentation fault — it turns out that building a complete GNU/Linux system puts a lot of stress on QEMU, especially when it is emulating a machine with limited memory resources (say, less than 8 GB of RAM).

If QEMU crashes, you won't wind up at a rescue shell in the virtual machine, because the virtual machine itself won't be around any more! In that case, you should start by making sure the disk image file has not been corrupted by having QEMU crash; `qemu-img check` is the command you should use for this. Then you can inspect the state of the filesystem using `qemu-nbd`, or by attaching the volume to a different virtual machine, or by restarting the QEMU program but changing the

`-append` option so it specifies `init=/scaffolding/bin/bash` rather than `init=/scaffolding/sbin/init`.

Again, cleaning up the build artifacts from the package user home directory and `/tmp/cblwork` might be a good idea before restarting. The restart procedure in this case is just to re-run QEMU and let the init process run.

If you can't figure out what the problem is, you can ask for help! People in the free software community are often very nice and helpful. There are mailing lists for most projects, or IRC channels. Stack Overflow is a really good resource.

But, basically, when things go wrong you have to just use your intelligence guided by experience.

Appendix A: Trustworthy Host-System Programs

CBL should be easy to build on most GNU/Linux systems.

The blueprints that make up CBL are designed to be used with litbuild, which is a ruby program. You don't *have* to use litbuild — you can just extract the commands from the blueprints, insert parameter values, and run them yourself — but that's a huge hassle! A better approach, if you want to run every command manually, is to generate scripts with litbuild and then run (via a lot of typing or copy-paste) commands from those scripts.

A.1. Host system requirements

Since Ruby is needed to run the litbuild program, you need ruby. We recommend using the latest stable version of C Ruby, which is generally the version specified in the ruby blueprint. Litbuild probably works fine in JRuby or Rubinius or other implementations of the Ruby language, but it's developed in C Ruby.

You also need a working C++ compiler, able to compile GCC, of course — according to the GCC documentation, this can be any ISO C++98 compiler, but we haven't actually tried using anything other than GCC to start with. You need a complete C standard library and its header files, which will typically be glibc, the GNU C library. And you need other tools that are typical parts of development systems, mostly from the GNU project: bash, bc, binutils, file, gawk, grep, m4, make, ncurses, patch, perl, sed, tar, texinfo, and perhaps others. (If you run into problems when you have all of those installed, please let us know!)

You don't have to start with all of those provided by the host system, of course. If it's more convenient, you can build most of the above from the source code packages that are already part of the CBL build process. But you do need at least a working C++ toolchain and the `make` program to build those, and sometimes other things as well. It's *probably* easier to start by installing the stuff mentioned above using whatever facility is provided by the host operating system.

If you are building CBL with a QEMU emulated virtual machine as the target device, you will need to have the nbd (Network Block Device) kernel feature available, either built-in or as a module. When configuring your kernel, this is under Device Drivers, Block Devices, and is called "Network block device support." This will be needed to copy files into the QEMU disk image files that the virtual machine will use for its filesystems.

It's a very good idea to start a new CBL effort by making sure you're working with a recent version of the GNU toolchain — really, we're only talking about the binutils and GCC at this point, because the C library is whatever you've got on the system, and the kernel headers need to be consistent with the C library you're using, even if you've upgraded the kernel since then. Ideally, it would be good to start with modern versions of the other dependencies mentioned above, as well, but the toolchain is the most important thing: an issue with the host system's toolchain can cause all sorts of weird problems. There are also a few other programs that aren't *always* installed by default on GNU/Linux systems, or have very specific version requirements: in the CBL repository of upstream sources and patches, everything is compressed with `lzip` because it is a little more space-efficient

than anything else, so of course you need that to be installed. Also, the `file` package is fussy about cross-compilation: you need the same version of `file` that will be built for the target system to be installed and available on the host.

With all that in mind, an optional step before you start the CBL process *per se* is to build a set of host-system components that will be used to build the cross-toolchain, as well as the QEMU emulator that is used to verify the cross-toolchain and may also be used for the target system build.

Another advantage of building these tools is as a way of verifying that the host system is reliable enough to use as the starting point for the CBL process! Native toolchain builds are usually straightforward and problem-free, so if you can't build a native binutils and GCC to start with, that's a good indication that you should consider using a different host system for the CBL build.

CBL may be re-run any number of times using an existing set of trusted host tools. We set LITBUILDDBDIR to a host-tools-specific location while executing this build to avoid unnecessary rebuilds of these programs.

Environment variable: LITBUILDDBDIR

`/home/random/cbltools/litbuilddb`

You can install these tools anywhere, but you should make certain that location is earlier on your `PATH` than any of the normal system directories for all the subsequent steps.

Environment variable: PATH

`/home/random/cbltools/bin:$PATH`

Environment variable: LD_LIBRARY_PATH

`/home/random/cbltools/lib:/home/random/cbltools/lib64`

Environment variable: PKG_CONFIG_PATH

`/home/random/cbltools/lib/pkgconfig`

Environment variable: LC_ALL

`POSIX`

Environment variable: CFLAGS

(should not be set)

Environment variable: CXXFLAGS

(should not be set)

Environment variable: LDFLAGS

(should not be set)

It's important to build the lzip package first, because otherwise it might not be available to build the rest of these packages.

A.2. lzip (host-prerequisites phase)

For an overview of lzip, see lzip.

A lot of systems don't have lzip automatically installed, and sometimes they don't even have a version easily available for installation. So the first host-system tool we build as a prerequisite is lzip, so that all the subsequent sources can be decompressed during the build.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.3. ed (host-prerequisites phase)

For an overview of ed, see ed.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.4. bc (host-prerequisites phase)

For an overview of bc, see bc.

The linux kernel uses `bc` in its build process, so we need to make sure we have one handy.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

A.5. binutils (host-prerequisites phase)

For an overview of binutils, see binutils.

Build Directory	`../build-binutils`

This is a completely ordinary native build of the binutils. It's built without multilib support, because that eliminates a couple of problems that can come up later on.

GNU binutils, like many toolchain components, should be constructed from a directory outside the source directory.

Build Directory

`../build-binutils`

When configuring binutils, we always specify `--enable-64-bit-bfd`. This is needed to enable 64-bit support, which is important when the host or target systems have a 64-bit userspace. It's unimportant for entirely 32-bit builds (for example, an i686-to-mipsel CBL build), but doesn't cause any problems when it's used in one of those builds.

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/home/random/cbltools \
  --disable-nls --enable-shared --disable-multilib \
  --enable-64-bit-bfd
```

```
make configure-host
```

The binutils build is set up to treat all compiler warnings as errors, which is generally a good idea — it protects against some types of poor programming practice that might lead to bugs, by causing the build to abort entirely if the compiler complains about anything. However, the latest version of GCC adds new kinds of compiler warnings that can be triggered by minor existing issues in the binutils codebase, causing the build to fail either for native or cross binutils builds. We *could* patch the binutils source to clean up the compiler warnings, but another option is simply to tell the build process to treat those specific warnings simply as warnings rather than errors so the build can proceed.

If the compiler being used to build the binutils is not modern enough to support these error messages, the build will fail because of these references to them. If that happens, just remove these `sed` commands and try again.

Configuration commands:

```
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=stringop-truncation@' bfd/Makefile
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=stringop-truncation@' gas/Makefile
sed -i -e '/^WARN_CFLAGS/s@$@ -Wno-error=format-overflow@' binutils/Makefile
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

A.6. bzip2 (host-prerequisites phase)

For an overview of bzip2, see bzip2.

bzip2 is installed in the host-prerequisites because it is used by lrzip.

Configuration commands:

```
(none)
```

Compilation commands:

```
sed -i 's@^CFLAGS=@CFLAGS=-fPIC @' Makefile
```

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make PREFIX=/home/random/cbltools install
```

A.7. cmake (host-prerequisites phase)

For an overview of cmake, see cmake.

Configuration commands:

```
./bootstrap --prefix=/home/random/cbltools \
  --no-system-curl \
  --no-system-expat \
  --no-system-jsoncpp \
  --no-system-zlib \
  --no-system-bzip2 \
  --no-system-liblzma \
  --no-system-libarchive \
  --no-system-librhash \
  --no-system-libuv
```

Compilation commands:

```
make
```

Test commands:

```
make test || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

A.8. libressl (host-prerequisites phase)

For an overview of libressl, see libressl.

Configuration commands:

```
./configure --prefix=/home/random/cbltools --enable-nc
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.9. curl (host-prerequisites phase)

For an overview of curl, see curl.

Configuration commands:

```
./configure --prefix=/home/random/cbltools --enable-optimize \
  --with-ca-bundle=/etc/ssl/cert.pem --with-openssl
```

Compilation commands:

```
make
```

On my computer, test 1474 fails — one test out of 1330. I don't have enough energy or motivation to try to track down the one failure.

Test commands:

```
make -k check || echo "Exit code $? - continuing anyway."
```

Installation commands:

```
make install
```

A.10. dosfstools

Name	DOS Filesystem Tools
Version	4.2

Project URL	https://github.com/dosfstools/dosfstools
SCM URL	https://github.com/dosfstools/dosfstools
Download URL	https://github.com/dosfstools/dosfstools/releases

A.10.1. Overview

This package contains programs for creating and validating various types of FAT filesystems.

A.10.2. dosfstools (host-prerequisites phase)

This package uses the normal GNU build system.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

The test suite uses the tool `xxd`, which is part of the vim package. If vim is not installed, the tests will basically be skipped.

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.11. elfutils (host-prerequisites phase)

For an overview of elfutils, see elfutils.

With GCC 8, there is a compiler warning that gets treated as an error and causes the build to abort. As per usual, we can tell GCC not to treat that warning as an error after all.

This package also includes something called "debuginfod" — some kind of client/server system that presumably provides debugging information. It depends on some other package, I'm not entirely sure what, but we can simply disable debuginfod rather than tracking down and setting up whatever it requires.

Configuration commands:

```
CFLAGS="$CFLAGS -Wno-error=packed-not-aligned -I/home/random/cbltools/include" \
```

```
  ./configure --prefix=/home/random/cbltools \
  --disable-debuginfod --disable-libdebuginfod
```

Compilation commands:

```
make
```

Some of the tests do not reliably succeed.

Test commands:

```
make check || echo "exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

A.12. expat (host-prerequisites phase)

For an overview of expat, see expat.

We frequently configure expat with `--without-docbook`, which causes the package to be installed without a man page for the `xmlwf` program. That eliminates a dependency on docbook2X or xmlto.

Configuration commands:

```
./configure --prefix=/home/random/cbltools --without-docbook
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.13. file (host-prerequisites phase)

For an overview of file, see file.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.14. gmp (host-prerequisites phase)

For an overview of gmp, see gmp.

Build Directory	`../build-gmp-1`

This is a standard build of GMP with C++ support enabled.

Build Directory

`../build-gmp-1`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/home/random/cbltools --enable-cxx
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.15. mpfr (host-prerequisites phase)

For an overview of mpfr, see mpfr.

Build Directory	../build-mpfr-1

Build Directory

../build-mpfr-1

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/home/random/cbltools \
  --with-gmp=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.16. mpc (host-prerequisites phase)

For an overview of mpc, see mpc.

Build Directory	../build-mpc-1

Build Directory

../build-mpc-1

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/home/random/cbltools \
  --with-gmp=/home/random/cbltools --with-mpfr=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.17. isl (host-prerequisites phase)

For an overview of isl, see isl.

Build Directory	`../build-isl-1`

Build Directory

`../build-isl-1`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/home/random/cbltools \
  --with-gmp-prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.18. gcc (host-prerequisites phase)

For an overview of gcc, see gcc.

Build Directory	`../build-gcc`

This is a fairly ordinary native compiler build. As with binutils, this is built to be as simple as possible: no multilib support, targeting just the one type of binary format we want to use. This simplicity prevents a few potential problems from cropping up later on. (You can review the GCC installation instructions if you want more details about this.)

By default, GCC does a three-stage "bootstrap" build of the C++ compiler: it builds an initial compiler with whatever system C++ compiler is already present on the system (which might be GCC or might be some other C++ compiler like LLVM's `clang++`). Then that first compiler is used to compile a second copy of itself, and then *that* copy is used to build a third copy; if the second and third copies are not basically identical, the build machinery deduces that the resulting GCC cannot build itself consistently and aborts the build. Depending on how important you consider the compiler we're building at this point, you may decide that's not necessary. If you don't want to do a three-stage bootstrap, you can add the directive `--disable-bootstrap` to the configure command here.

Setting `--with-local-prefix` as well as `--prefix` prevents this compiler from searching the host system's `/usr/local` directory for header files. Making CBL be as independent of the host system as possible is one of the ways we reduce the fragility of the process.

Build Directory

`../build-gcc`

Configuration commands:

```
${LB_SOURCE_DIR}/configure --prefix=/home/random/cbltools \
  --with-local-prefix=/home/random/cbltools --disable-multilib \
  --disable-nls --enable-shared --enable-languages=c,c++ --enable-c99 \
  --enable-long-long --enable-threads=posix \
  --with-gmp=/home/random/cbltools --with-mpfr=/home/random/cbltools \
  --with-mpc=/home/random/cbltools --with-isl=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make -k check || echo "Exit code $?; continuing anyway."
```

Installation commands:

```
make install
```

A.19. libffi (host-prerequisites phase)

For an overview of libffi, see libffi.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.20. pcre (host-prerequisites phase)

For an overview of pcre, see pcre.

Configuration commands:

```
./configure --prefix=/home/random/cbltools --enable-jit --enable-pcre16 \
  --enable-pcre32 --enable-utf --enable-unicode-properties
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.21. pkgconf (host-prerequisites phase)

For an overview of pkgconf, see pkgconf.

Some of the programs we build later on like to use pkg-config to find their dependencies, so we should provide one.

Since the build scripts produced by the autotools sometimes like to re-execute autotools based on the timestamp of different build-related files, it's a good idea to reset the timestamps after applying patches like the `run-autogen` patch. The `touch` command here resets the timestamp for every file in the distribution to be the same as the timestamp on the `README.md` file.

Configuration commands:

```
find . -exec touch -r README.md {} \;
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
ln -sf pkgconf /home/random/cbltools/bin/pkg-config
```

A.22. zlib (host-prerequisites phase)

For an overview of zlib, see zlib.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.23. python (host-prerequisites phase)

For an overview of python, see python.

Dependencies

expat, libffi, zlib.

Configuration commands:

```
CFLAGS='-I/home/random/cbltools/include' \
  LDFLAGS='-L/home/random/cbltools/lib -L/home/random/cbltools/lib64' \
  ./configure --prefix=/home/random/cbltools --with-system-expat \
  --with-system-ffi --enable-shared --enable-optimizations
```

Compilation commands:

```
make
```

The tests take a long time to run and are not reliable. This is an ephemeral build of python and only needed by QEMU, so the tests are overkill at this point.

Test commands:

```
(none)
```

Installation commands:

```
make install
```

This installation should be available using the program name `python` as well as `python3`. The same applies to some other programs.

Installation commands:

```
ln -sf python3 /home/random/cbltools/bin/python
ln -sf pip3 /home/random/cbltools/bin/pip
ln -sf idle3 /home/random/cbltools/bin/idle
```

A.24. setuptools (host-prerequisites phase)

For an overview of setuptools, see setuptools.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
/home/random/cbltools/bin/python3 setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
/home/random/cbltools/bin/python3 setup.py install
```

A.25. ninja (host-prerequisites phase)

For an overview of ninja, see ninja.

Configuration commands:

```
(none)
```

There is a bootstrap process that uses a python script to compile an initial `ninja` program just by compiling all non-test source code files and linking them together, and then uses that program to build the *real* `ninja` program.

Compilation commands:

```
./configure.py --bootstrap
```

Test commands:

```
./ninja ninja_test
./ninja_test
```

Ninja is installed just by copying the `ninja` program to a binary directory.

Installation commands:

```
cp ninja /home/random/cbltools/bin
```

A.26. meson (host-prerequisites phase)

For an overview of meson, see meson.

Meson can be installed using the standard setuptools process.

Configuration commands:

```
(none)
```

Compilation commands:

```
/home/random/cbltools/bin/python3 setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
/home/random/cbltools/bin/python3 setup.py install
```

A.27. flit (flit_core_prerequisites phase)

For an overview of flit, see flit.

Flit includes a separate `flit-core` module that can be used by itself, and in many cases, `flit-core` is good enough, and the full `flit` program is not needed. That is the case for Trustworthy Host-System Programs, so we will only build `flit_core` at this point.

Configuration commands:

```
(none)
```

Compilation commands:

```
pushd flit_core
/home/random/cbltools/bin/python3 -m flit_core.wheel
popd
```

Test commands:

```
(none)
```

Installation commands:

```
pushd flit_core
/home/random/cbltools/bin/python3 bootstrap_install.py dist/flit_core-*.whl
popd
```

A.28. packaging (host-prerequisites phase)

For an overview of packaging, see packaging.

Dependencies

flit (flit_core_prerequisites phase).

Configuration commands:

```
(none)
```

Compilation commands:

```
/home/random/cbltools/bin/python3 -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
/home/random/cbltools/bin/pip3 install dist/packaging*whl
```

A.29. glib (host-prerequisites phase)

For an overview of glib, see glib.

By default, glib wants to link something against libmount, which is provided by util-linux. I don't know what functionality within glib makes calls to libmount functions, but it's not anything we need for CBL, so we can simply disable it.

On some host systems, the glib build will not find the host-prerequisites libffi unless `CFLAGS` is set. Meson honors the `CFLAGS` environment variable, so we can use the same technique we use with autotools-based projects.

Configuration commands:

```
CFLAGS=-I/home/random/cbltools/include meson setup --prefix /home/random/cbltools \
  --libdir /home/random/cbltools/lib -Dlibmount=disabled _build
```

Compilation commands:

```
ninja -C _build
```

On some host systems, I find that some tests fail with errors. As with python, this build is probably not important enough to worry about.

Test commands:

```
(none)
```

Installation commands:

```
ninja -C _build install
```

A.30. libslirp (host-prerequisites phase)

For an overview of libslirp, see libslirp.

Configuration commands:

```
meson setup --prefix /home/random/cbltools --libdir /home/random/cbltools/lib _build
```

Compilation commands:

```
ninja -C _build
```

Test commands:

```
(none)
```

Installation commands:

```
ninja -C _build install
```

A.31. lzo (host-prerequisites phase)

For an overview of lzo, see lzo.

LZO is installed as one of the host tools because it's used by lrzip.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.32. lz4 (host-prerequisites phase)

For an overview of lz4, see lz4.

lz4 is used by lrzip, so we set it up as one of the host system prerequisites.

Configuration commands:

```
(none)
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make prefix=/home/random/cbltools install
```

A.33. lrzip (host-prerequisites phase)

For an overview of lrzip, see lrzip.

LRZip is not really necessary, but I use it to compress the QEMU disk images when I package up Little Blue Linux builds — see the `qemu-to-qemu-build` blueprint for more details.

Configuration commands:

```
./autogen.sh
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.34. ncurses (host-prerequisites phase)

For an overview of ncurses, see ncurses.

One package built in the host scaffolding section — specifically, texinfo — needs to compile an intermediate program, used during its own build process, with a native compiler. The build system only compiles that program if the `ncurses` library is available — even though, as far as I can tell, it doesn't actually use any functions provided by that library.

Rather than figuring out exactly how to modify the build system so that it doesn't require ncurses, we can ensure that ncurses is available by adding it to the set of host-system prerequisites.

The `--without-ada` option stops ncurses from adding support for the Ada compiler, since Ada may not be available in the host system and we don't need that support in any event. We also specify `--with-versioned-syms` so that the shared library is built with versioned symbols — some programs and libraries complain if these are not present.[1]

Configuration commands:

```
./configure --prefix=/home/random/cbltools --with-shared --without-debug \
  --without-ada --with-versioned-syms
```

Compilation commands:

```
make
```

Test commands:

```
(none)
```

Installation commands:

```
make install
```

A.35. pixman (host-prerequisites phase)

For an overview of pixman, see pixman.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

A.36. pth (host-prerequisites phase)

For an overview of pth, see pth.

Configuration commands:

```
./configure --prefix=/home/random/cbltools
```

Compilation commands:

```
make -j1
```

Test commands:

```
make -j1 test
```

Installation commands:

```
make -j1 install
```

A.37. qemu (host-prerequisites phase)

For an overview of qemu, see qemu.

Dependencies

pixman, zlib.

When configuring QEMU, the `--target-list` option controls which emulation targets are built, using a comma-separated list. For each machine architecture, the usermode program emulator is the "linux-user" target, and the full system emulator is the "softmmu" target. (I believe that "softmmu" is actually a QEMU feature that provides accelerated memory mapping and I/O in the virtual machine, or something like that.) If you want to save time, you can add a `--target-list` option with a comma-delimited list of the usermode and full system emulator targets for all the machine types

you plan to build CBL for. Otherwise, leave the configure command alone and all targets will be built.

The usermode emulator is always used in CBL to verify that the cross-toolchain works properly. The full-system emulator is used only when performing target-side builds (or, sometimes, host-side builds) in a virtual machine, but that's very often the case when following the CBL process.

Configuration commands:

```
./configure --prefix=/home/random/cbltools --cc=gcc --enable-slirp
```

Compilation commands:

```
make
```

One of the tests fails for me — `test-qga`. I don't know what that is, but I'm pretending that it does not matter enough to worry about. You should probably check the log file to see if you have any other issues.

Note that you really should not try to run the test suite in an emulated machine! It will take forever. So if for some reason you're building the host-prerequisites on an emulated VM, change the `test-commands` here to `(none)`.

Test commands:

```
make check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

A.38. yoyo (host-prerequisites phase)

For an overview of yoyo, see yoyo.

Configuration commands:

```
(none)
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
cp -a -v build/yoyo /home/random/cbltools/bin
```

[1] If you see messages like `libncursesw.so.6: no version information available (required by foo)`, that's what I'm talking about here. Usually, programs still work just fine, but I find the warning messages a little irritating.

Appendix B: Rebuild the Packages Whose Tests Have Not Been Run

This section is strictly optional but highly recommended!

The test suite for a few packages — the exact list changes over time, as issues are resolved and new ones emerge — is problematic during the target side of the CBL build. This is generally because the system is not complete enough to run the tests successfully or because the network is unavailable during the target-side build.

In some cases, this is not such a big deal — the test suite for curl, for example, doesn't matter too much to me because if curl doesn't work properly that will probably be pretty obvious. But others *are* important: among the problematic packages is glibc, and since the C library is one of the most foundational packages in a GNU/Linux system, and issues with the C library are likely to be extremely subtle and present themselves as obscure issues in *other* packages, it is a really good idea to run its test suite at some point. And as long as we're going to run its test suite, we might as well rebuild the other packages that also have problematic tests, and run *their* test suites as well.

This should only be done after the base CBL process is complete and the reuslting Little Blue Linux system is running and on the network.

You'll find that the directives for this phase of the packages are almost identical to those used to build the package in the final system; the only typical difference is in the `test-commands`. Arguably, since the same process is being used to build the package this time, with the same software construction tools, same source code, and so on, the result should be basically identical to the version already installed on the system, and there's no real need to install the newly-built version of the package over the currently-installed one.

I'm doing that anyway, though, because it's always possible that there *will* be differences — perhaps because the configuration routine will observe that there are optional dependencies available at this point that were not present in the previous build, or something of the sort. In some cases, as with python, this can actually provide a significant benefit — the version installed here has optimizations enabled, while the base system version does not.

B.1. curl (rebuild-untested-packages phase)

For an overview of curl, see curl.

Configuration commands:

```
./configure --prefix=/usr --enable-optimize \
  --with-ca-bundle=/etc/ssl/cert.pem --with-openssl
```

Compilation commands:

```
make
```

curl has a variety of test targets. I have trusted that they mean basically what they look like they mean, and opt to run the "non-flaky" tests. If you `grep : Makefile | grep test`, you'll see them all.

As is often the case, the test target frequently exits with a nonzero status; you'll need to review the test output and results manually to determine whether there are any real problems. At the conclusion of the run, you'll get a bunch of summary ouptut; as an example, I just got:

```
These test cases failed: 77 78 141 249 1044 1126 1127 1128 1239 1424
```

You can see what those tests are doing by looking at the test definition files under `tests/data`; for example, test number 141 is defined at `tests/data/test141`.

Test commands:

```
make -k test-nonflaky || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

B.2. gawk (rebuild-untested-packages phase)

For an overview of gawk, see gawk.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc
```

Compilation commands:

```
make
```

I get one test failure, for `mktime`. I'm not sure which of these is expected and which is what is actually observed, but the failure is because `1483228800` and `1483228827` are not the same. I suspect some timer is either higher resolution than expected, or not-as-high a resolution as expected.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

We don't have to re-install the documentation, it's still there from the previous build.

Installation commands:

```
make install
```

B.3. glibc (rebuild-untested-packages phase)

For an overview of glibc, see glibc.

Build Directory	../build-glibc-3

Build Directory

../build-glibc-3

This is one of the packages that runs install-info in a problematic way. That doesn't matter for the build we did at the beginning of the target system build, since glibc was the first package to write info files, but it matters when rebuilding it.

Pre-build (as root) commands:

```
chown glibc:install /usr/share/info/dir
```

Post-installation (as root) commands:

```
chown root:install /usr/share/info/dir
chmod 664 /usr/share/info/dir
```

If you've adjusted the configuration for the final system, you'll want to make the same adjustment here.

Configuration commands:

```
CFLAGS="-g -O2" ${LB_SOURCE_DIR}/configure --prefix=/usr \
  --disable-timezone-tools
```

Compilation commands:

```
make
```

The test results summary will show you how many tests failed, passed, and so on. Potential problems are tests that failed unexpecetdly (that's reported as FAIL), and tests that *passed* unexpectedly (that's XPASS). You should look at those. I frequently see failures in io/tst-open-tmpfile, resolv/tst-p_secstodate, and rt/tst-shm, and don't consider those to be real problems. (I also got six XPASS results just now, but I don't pay as much attention to those.)

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
pushd ${LB_SOURCE_DIR}
```

```
sed '/test-installation/s@$(PERL)@echo skipping@' -i Makefile
popd
make install
make localedata/install-locales
```

B.4. gnutls (rebuild-untested-packages phase)

For an overview of gnutls, see gnutls.

Configuration commands:

```
./configure --prefix=/usr --with-included-unistring \
  --without-p11-kit --enable-ssl3-support --disable-ssl2-support \
  --enable-openssl-compatibility
```

Compilation commands:

```
make
```

I get a few test failures — `crq_apis`, `dn2`, `gnutls_x509_crt_sign`, `crl_apis`, and `ocsp`. If you have similar results and have enough energy to investigate, you can find the test programs under `tests`. As an example of what I find when I investigate, the `dn2` test fails because it expects a distinguished name with an `acivated` field of `2008-11-13 00:00:00 UTC` but instead gets one with `2008-11-12 23:59:37 UTC`. Maybe that's because of leap seconds or something? I can imagine circumstances where I'd be concerned about that kind of thing, but I'm not in any of those circumstances here.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

B.5. python (rebuild-untested-packages phase)

For an overview of python, see python.

Since we're running the tests this time, we can also enable optimizations. The configuration is otherwise just like the previous one.

Configuration commands:

```
./configure --prefix=/usr --with-system-expat --enable-shared \
  --with-lto --with-ensurepip=install --with-openssl=/usr \
  --enable-optimizations
```

Compilation commands:

```
make
```

The tests that are run during compilation, in order to produce an optimized build, is a small subset of the full test suite, so we should still do that.

I get eight test failures, and don't have enough energy to look into them all — from the names of the tests, I suspect that they mostly have to do with timezones or not being able to reach the Internet.

On some architectures — this happens sometimes on AArch64, but I haven't seen it on x86_64 — some tests hang, apparently, forever. I've seen this with `test_multiprocess_fork` and `test_concurrent_futures`. If you're keeping an eye on the build and can take manual action to resolve it, you can kill the hanging tests after you're tired of waiting for them to terminate on their own, with e.g. `pkill -f test_multiprocess_fork`. Since I typically run the entire Rebuild the Packages Whose Tests Have Not Been Run blueprint unattended and don't want it to hang forever, I kill the test run after two hours using the `timeout` program.[1]

Test commands:

```
timeout --kill-after=5m 2h make -k test || \
  echo "Exit code $?: continuing anyway"
```

All of the extra stuff we did to set up Python after the basic installation is still fine and does not have to be redone.

Installation commands:

```
make install
```

B.6. ruby (rebuild-untested-packages phase)

For an overview of ruby, see ruby.

Configuration commands:

```
./configure --prefix=/usr --sysconfdir=/etc --enable-shared \
  --enable-debug-env
```

Compilation commands:

```
make
```

I have a handful of errors and failures, mostly related to timestamps being slightly off and TLS certificate management — since I'm using LibreSSL instead of OpenSSL, that's probably the cause for at least the certificate-related issues.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

B.7. texinfo (rebuild-untested-packages phase)

For an overview of texinfo, see texinfo.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make -k check
```

Installation commands:

```
make install
```

B.8. util-linux (rebuild-untested-packages phase)

For an overview of util-linux, see util-linux.

Configuration commands:

```
./configure --enable-write
```

Compilation commands:

```
make
```

I have one failing test but that does not seem problematic enough to fuss about.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

[1] The command line arguments here mean: "Run `make -k test`. If that's still running after two hours, send a `TERM` signal to the process. If it's still running five minutes after that, send a `KILL` signal to the process."

Appendix C: Creating a LB Linux AWS EC2 Machine Image

C.1. What's This All About?

Amazon Web Services, or AWS, is a computing services platform available to anyone who wishes to use it. It's one of the platforms people mean when they talk about a "public cloud" or more generally "the cloud." You can sign up for an AWS account and then use it to launch virtual computers, store files with a variety of different options, set up managed services like databases and caches — I'm not going to try to itemize all the kinds of services that are available in AWS, because they're adding new services and options on a weekly basis.

The Elastic Compute Cloud, or "EC2," service is one of the most foundational AWS offerings. It provides virtual computers and storage devices; with it, you can (among other things) launch a new virtual machine called an "instance," based on a template called an "Amazon Machine Image" or "AMI."

I like to be able to launch instances of the basic Little Blue Linux system easily within EC2. This section is about how to do that: it describes how to set up a new LB Linux AMI, starting from a freshly-built system that reached `COMPLETE SUCCESS` in the CBL process.

There are lots of different kinds of AMIs with different virtualization technology (HVM, Paravirtual), different kinds of storage (EBS-backed, Instance Store-backed), and probably other variations — and an equally-diverse way of creating them. This section would be a rather large book all on its own if I wanted to be exhaustive about all the possibilities! If you really want to know all about the subject, you can read the documentation at `https://doc.aws.amazon.com/AWSEC2/latest/UserGuide`. All I'm going to do here is describe one way to create an EBS-backed HVM AMI — which is the type of instance you will almost always want; while other types can still work, they appear to be treated as legacy options. New instance types are all designed for EBS HVM instances.

NOTE If you search on the Internet for instructions on how to build new AMIs from the ground up, you may still find a lot of ancient information about how to build AMIs with instance storage or with Amazon kernel images. As of 2020, I am pretty sure the instructions here describe by far the best approach.

Unpacking those specific terms: "EBS-backed" means that, on instances created from the AMI, the storage device seen by the CBL kernel as its root filesystem volume will actually be network-attached volumes provisioned in the "Elastic Block Store" service. These volumes can be detached from one instance and attached to another, if desired, and can persist long after any given instance is destroyed. "HVM" means that images will be "Hardware-Assisted Virtual Machines" rather than paravirtualized guests — they use the virtualization features available on modern CPUs to avoid a substantial performance penalty for running as virtual machines.

Again, I'm not trying to make all these terms crystal clear! If you want to know more, read the EC2 documentation.

The process I'm going to follow in setting up the CBL AMI has three main stages:

1. First, I'm going to install all the additional software that I like to have on EC2 instances, since there's no point in doing that compilation and installation work on an actual EC2 instance. Always remember, with AWS, that you pay for everything you provision. If you build software on your own computer and then upload it to AWS, you're not paying them for the CPU cycles and memory that compilation takes!
2. Second, I'm going to get a basic LB Linux machine set up as an EC2 instance that boots and is accessible via SSH.
3. Finally, once that's working, I'm going to finish configuring it as a general-purpose AWS AMI that can be used by anyone who wants to create instances from it.

CAUTION	Although you can absolutely use litbuild to generate a document for this entire section, you should *not* run the script produced by litbuild! Install Additional Software For EC2 is designed to run on a basic LB Linux system before creating the initial AMI. The Optimizing and Improving the Base CBL AMI scripts should be run on an EC2 instance produced from the initial AMI.

C.2. Install Additional Software For EC2

There are a bunch of packages that I don't use enough to justify including them as part of the base CBL process but are useful or necessary on AWS EC2 instances. Let's set those up here.

EC2 instances that run on the Nitro hypervisor are expected to shut down when an ACPI power-button event occurs. This requires a userspace program to observe and respond to the event; typically this is `acpid`.

C.2.1. acpid

Name	ACPI event daemon
Version	2.0.34
Project URL	http://sourceforge.net/projects/acpid2/
SCM URL	git://git.code.sf.net/p/acpid2/code
Download URL	https://sourceforge.net/projects/acpid2/files/latest/download

Modern computers often support a standard (but really quite complicated) system control mechanism called the Advanced Configuration and Power Interface, generally referred to as ACPI. You can find the latest ACPI specification at `http://www.acpi.info/spec.htm` if you'd like to read all about it.

ACPI allows an operating system to discover hardware components available on a system and often control aspects of those components. ACPI also allows the operating system to respond to various sorts of events — for example, when the screen of a laptop computer is closed or opened, this can generate events that are visible to the operating system, which might then respond to those events by telling the computer (also via ACPI) to switch to a low-power-consumption "suspended" state.

This package provides a long-running daemon process, `acpid`, that lets you control the system's response to ACPI events using configuration files in userspace, conventionally located in the `/etc/acpi` directory.

This package is built as usual for the GNU build system, so mostly what we need to do in this section is configure `acpid`; the only thing we're going to do here is set the system up so that the system shuts down cleanly when the power button is pressed.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

Test commands:

```
make check
```

Installation commands:

```
make install
```

Any time an ACPI event occurs, `acpid` looks for any rule that matches that event in `/etc/acpi/events` and runs the program specified by that rule. Rule files can be named almost anything, and the only requirement is that they specify an `event` regular expression and an `action` to perform when that event is detected. In the `action` specification, you can specify `%e` to pass the full text of the event to the script; in this case, since we just want to shut down the computer, that's unnecessary.

It's actually unnecessary even to run a script as the action; we could run `/sbin/poweroff` directly. It's conventional to have acpid run scripts, though, in case it's necessary to have additional logic that determines the correct thing to do; I'm retaining that pattern because it may become helpful at some point in the future.

Installation commands:

```
mkdir -p /etc/acpi/events
echo 'event=button/power' > /etc/acpi/events/powerbutton
echo 'action=/etc/acpi/power.sh' >> /etc/acpi/events/powerbutton
echo '#!/bin/bash' > /etc/acpi/power.sh
echo '/sbin/poweroff' >> /etc/acpi/power.sh
chmod 775 /etc/acpi/power.sh
```

Configuration Files

- `/etc/acpi`
- `/etc/s6-rc/source/acpid-svc`

- `/etc/s6-rc/source/acpid-log`

Service Pipeline: `acpid` *(in bundle ``)*

Service 1: Longrun `acpid-svc`	
Dependencies	• `mount-sys`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`fdmove -c 2 1`
	`/usr/sbin/acpid --netlink --foreground --logevents`

Service 2: Longrun `acpid-log`	
notification-fd	`3`
Dependencies	• `remount-root-rw`

Run script	`#!/bin/execline -P`
	`emptyenv`
	`s6-log -d3 T s10000000 n10 /var/log/acpid`

Post-installation (as `root`*) commands:*

```
mkdir -p /var/log/acpid
chown root:root /var/log/acpid
chmod 0755 /var/log/acpid
```

It's recommended for EC2 instances to use NTP to configure their system clock.

C.2.2. chrony

Name	Chrony
Version	4.4
Project URL	https://chrony.tuxfamily.org/
SCM URL	*(unknown)*
Download URL	https://chrony.tuxfamily.org/download.html

The Network Time Protocol, NTP, is a protocol defined in the Internet RFC 5905, which is used to synchronize the clocks on computers. That sounds really simple but is actually trickier than you would think, and it's also quite important — if different computers that are part of a single distributed system have different notions of what time it is, any time-series data collected within that system will be inconsistent and misleading.

Chrony is a program that, like the `ntpd` program provided by the `ntp` package, implements NTP. It's designed to work well in a wider range of conditions than `ntpd` does — for example, computers that are only intermittently connected to the Internet, or that are often suspended or turned off, can

maintain a correct system clock more easily with chrony than with ntpd.

It's also simpler than `ntpd`, in that it doesn't implement some modes that aren't often used, like multicast client and server.

There are a lot of libraries that are not necessary to build chrony, but provide additional capabilities when they are available. These are all part of the base LB Linux system already:

- If libcap is available, chronyd will support dropping root privileges after being started.
- If nettle is available, chronyd will support better cryptographic hash functions than md5.
- The readline library can be used to provide line editing support in the chronyc program.

Configuration commands:

```
./configure --prefix=/usr
```

Compilation commands:

```
make
```

As is often the case, the tests don't all pass for me. In my case I get a failure in `util`, on "line 82," whatever that means.

Test commands:

```
make -k check || echo "Exit code $?, continuing anyway"
```

Installation commands:

```
make install
```

The chrony package includes two programs: a long-running process `chronyd` that is intended to run as a daemon, and a command-line program `chronyc` that can be used to monitor and manipulate it. Here, we're going to set up `chronyd` as a long-running process managed by s6.

The chronyd program is configured using the `chrony.conf` file.

There are a considerable number of configuration settings you *can* tune, but as usual for CBL we're going to go simple. First, we need to tell `chronyd` where to find upstream servers. There's a pool of publicly- accessible NTP servers that can be found simply by using DNS: when you query DNS for `pool.ntp.org`, you get a set of four IP addresses that change every hour. Chronyd can be told to use all four of these addresses with the `pool` directive. Adding `iburst` to that directive causes the first few requests sent to the servers to be only a few seconds apart, which allows chronyd to start updating the system clock more rapidly than a longer interval does.

Installation commands:

```
echo 'pool pool.ntp.org iburst' > /etc/chrony.conf
```

If the clock is off by a lot, adjusting it by slowing down or speeding up the system clock would take a really long time to get the clock adjusted to the correct time. `makestep` can be used to adjust the clock more rapidly in that case.

Installation commands:

```
echo 'makestep 0.1 3'
```

Chronyd can store a measurement history of all the time sources it uses; the `dumpdir` directive tells it where to put that history.

Installation commands:

```
mkdir -p /var/db/chronyd/dumpdir
echo 'dumpdir /var/db/chronyd/dumpdir' >> /etc/chrony.conf
```

The real-time clocks on most computers tend to be inaccurate and gradually drift away from the correct time. Chronyd measures and corrects for this drift, and can write the rate of that drift to a file so it doesn't need to re-learn the drift rate for the local system after restarting ntpd.

Installation commands:

```
echo 'driftfile /var/db/chronyd/drift' >> /etc/chrony.conf
```

Chronyd can also save data related to the accuracy of the real-time clock.

Installation commands:

```
echo 'rtcfile /var/db/chronyd/rtc' >> /etc/chrony.conf
```

If the real-time clock is set to UTC rather than local time, a directive can be specified so that chronyd is aware of that.

Installation commands:

```
echo 'rtconutc' >> /etc/chrony.conf
```

Chronyd is able to switch to a non-root user after startup, when linked against libcap.

Installation commands:

```
echo 'user chronyd' >> /etc/chrony.conf
```

Pre-build (as root) commands:

```
grep -q ^chronyd: /etc/group || groupadd -r chronyd
grep -q ^chronyd: /etc/passwd || useradd -r -g chronyd \
  -c 'chronyd daemon user' -d /var/empty -s /bin/false chronyd
```

Post-installation (as root) commands:

```
chown -R chronyd:chronyd /var/db/chronyd
```

If you're using chrony on a computer with no real-time clock hardware, and the system clock reverts to the epoch (January 1, 1970) at every boot, you can add an additional directive to the chronyd-svc run script: put `foreground { /usr/sbin/chronyd -q }` right before `/usr/sbin/chronyd -d`. This will run chronyd in a mode that immediately sets the system clock from some viable time source, and then terminates. Then the daemon can be started as normal and will keep the clock in synch.

Service Pipeline: chronyd (in bundle network-services)

Service 1: Longrun chronyd-svc	
Dependencies	• network-available • loopback-interface
Run script	#!/bin/execline -P emptyenv fdmove 2 1 /usr/sbin/chronyd -d
Service 2: Longrun chronyd-log	
notification-fd	3
Dependencies	• remount-root-rw
Run script	#!/bin/execline -P emptyenv s6-setuidgid chronyd s6-log -d3 T s10000000 n10 /var/log/chronyd

Post-installation (as root) commands:

```
mkdir /var/log/chronyd
chown chronyd:chronyd /var/log/chronyd
```

Configuration Files

- /etc/chrony.conf
- /etc/s6-rc/source/chronyd-svc
- /etc/s6-rc/source/chronyd-log

AWS provides a command-line tool (invoked simply as `aws`), written in Python, that can be used to invoke all of the APIs that control AWS resources. It's often very useful to have that tool available — although it has a lot of dependencies that need to be pulled in as well.

C.2.3. pyyaml

Name	PyYAML
Version	6.0.1
Project URL	https://pyyaml.org/wiki/PyYAML
SCM URL	https://github.com/yaml/pyyaml/
Download URL	https://github.com/yaml/pyyaml/tags

PyYAML is a Python library for parsing and emitting YAML.

As is typical for Python programs, there aren't any configuration or build stages.

There are some tests, which is a refreshing change of pace! But some of them fail for me, which is less so.

Configuration commands:

```
(none)
```

Compilation commands:

```
(none)
```

Test commands:

```
python setup.py test || echo "Exit code $?: continuing anyway"
```

This library can optionally use a native `libyaml` library — and, ideally, we would set it up that way, since `libyaml` is part of the basic LB Linux system. However, the location of a header file has changed in Python 3.11, and this causes compilation of the native extension to fail. For now we will just disable it.

Installation commands:

```
python setup.py --without-libyaml install
```

C.2.4. typing_extensions

Name	Typing extensions for python
Version	4.8.0
Project URL	https://pypi.org/project/typing_extensions/

SCM URL	https://github.com/python/typing_extensions
Download URL	https://pypi.org/project/typing_extensions/#files
Dependencies	flit (flit_core phase)

This is a library that enables the use of new type system features on older Python versions, and enables experimentation with new type system PEPs before they are accepted and added to the typing module.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/*whl
```

C.2.5. python-wheel

Name	wheel
Version	0.42.0
Project URL	https://pypi.org/project/wheel/
SCM URL	https://github.com/pypa/wheel
Download URL	https://pypi.org/project/wheel/#files
Dependencies	flit (flit_core phase)

The wheel module is the reference implementation for the Python "wheel" binary packaging standard, and provides an extension to setuptools for building wheel files.

In Little Blue Linux, this package is referred to as `python-wheel` because otherwise the package user name conflicts with the standard operating system group `wheel`. That does unfortunately mean that the release file must be repackaged so that it expands into a `python-wheel-$VERSION` directory.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/wheel*whl
```

C.2.6. setuptools_scm

Name	setuptools_scm
Version	8.0.4
Project URL	https://pypi.org/project/setuptools_scm/
SCM URL	https://github.com/pypa/setuptools_scm/
Download URL	https://pypi.org/project/setuptools_scm/#files
Dependencies	packaging, typing_extensions, python-wheel

setuptools_scm lets other python projects manage versions using SCM tags. The release tar file may be named as though the project name contains a hyphen, rather than an underscore — that is, `setuptools-scm-8.0.4.tar` rather than `setuptools_scm-8.0.4.tar` — and, if that's the case, it's important to repackage it using the litbuild naming convention.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation --wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/setuptools*whl
```

C.2.7. six

Name	Six
Version	1.16.0
Project URL	https://pypi.org/project/six/
SCM URL	https://github.com/benjaminp/six
Download URL	https://pypi.org/project/six/#files

Six is a compatibility library; it provides functions to make it easier to write python programs that work in both python 2 and 3.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

C.2.8. python-dateutil

Name	dateutil
Version	2.8.2
Project URL	https://dateutil.readthedocs.io/en/stable/
SCM URL	https://github.com/dateutil/dateutil/
Download URL	https://pypi.org/project/python-dateutil/#files
Dependencies	setuptools_scm, six

dateutil extends the Python datetime module with some (presumably convenient) additional methods.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

C.2.9. jmespath

Name	JMESpath
Version	1.0.1
Project URL	http://jmespath.org/
SCM URL	https://github.com/jmespath/jmespath.py
Download URL	https://github.com/jmespath/jmespath.py/releases

JMESpath is a library that lets you declaratively extract elements from a JSON document.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

C.2.10. botocore

Name	botocore
Version	1.31.77
Project URL	https://github.com/boto/botocore
SCM URL	https://github.com/boto/botocore
Download URL	https://github.com/boto/botocore/releases
Dependencies	python-dateutil, docutils, jmespath, urllib3

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

As mentioned in the python section, python packages often try to overwrite files that were already installed by other packages, so we need to work around that to avoid package-user permission violations.

Pre-build (as `root`) commands:

```
for prog in jp rst2html5 rst2latex rst2html rstpep2html rst2odt \
  rst2s5 rst2html4 rst2pseudoxml rst2odt_prepstyles rst2xml \
  rst2xetex rst2man; \
  do \
  if [ -f /usr/bin/${prog}.py ]; \
  then \
  mv /usr/bin/${prog}.py /usr/bin/${prog}.py.MOVED; \
  fi \
  done
```

Post-installation (as `root`) commands:

```
for prog in jp rst2html5 rst2latex rst2html rstpep2html rst2odt \
  rst2s5 rst2html4 rst2pseudoxml rst2odt_prepstyles rst2xml \
  rst2xetex rst2man; \
  do \
  if [ -f /usr/bin/${prog}.py.MOVED ]; \
  then \
  mv /usr/bin/${prog}.py.MOVED /usr/bin/${prog}.py; \
  fi \
  done
```

C.2.11. s3transfer

Name	s3transfer
Version	0.7.0
Project URL	https://github.com/boto/s3transfer
SCM URL	https://github.com/boto/s3transfer
Download URL	https://github.com/boto/s3transfer/releases
Dependencies	botocore

s3transfer is an Amazon S3 Transfer Manager written in Python.

As is typical for Python programs, there isn't any configuration to do.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

s3transfer has some automated tests, but they require additional packages that I don't feel like pulling in right now.

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
rm -f /usr/bin/jp.py
```

As mentioned in the python section, python packages often try to overwrite files that were already installed by other packages.

Pre-build (as `root`) commands:

```
for prog in jp rst2html5 rst2latex rst2html rstpep2html rst2odt \
  rst2s5 rst2html4 rst2pseudoxml rst2odt_prepstyles rst2xml \
  rst2xetex rst2man; \
  do \
  if [ -f /usr/bin/${prog}.py ]; \
  then \
  mv /usr/bin/${prog}.py /usr/bin/${prog}.py.MOVED; \
  fi \
  done
```

Post-installation (as `root`) commands:

```
for prog in jp rst2html5 rst2latex rst2html rstpep2html rst2odt \
  rst2s5 rst2html4 rst2pseudoxml rst2odt_prepstyles rst2xml \
  rst2xetex rst2man; \
  do \
  if [ -f /usr/bin/${prog}.py.MOVED ]; \
  then \
  mv /usr/bin/${prog}.py.MOVED /usr/bin/${prog}.py; \
  fi \
  done
```

C.2.12. pyasn1

Name	pyasn1
Version	0.4.8
Project URL	http://snmplabs.com/pyasn1/
SCM URL	https://github.com/etingof/pyasn1
Download URL	https://github.com/etingof/pyasn1/releases

pyasn1 is a pure-python implementation of the ASN.1 types and some standard codecs, like DER.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

C.2.13. poetry-core

Name	Poetry PEP-517 build backend
Version	1.8.1
Project URL	https://github.com/python-poetry/poetry-core
SCM URL	https://github.com/python-poetry/poetry-core
Download URL	https://pypi.org/project/poetry-core/#files

Poetry is a build system for python that appears to focus on dependency management and installation. I think it's kind of like bundler, in the ruby ecosystem.

This package, poetry-core, is the PEP-517 build backend for poetry, which can be used independently of the poetry package itself. We don't actually need the full poetry system, but some python packages require this build backend, so we need to make it available.

For some reason, even though the project is called `poetry-core`, the release tar file is distributed as `poetry_core` with an underscore. Remember to repackage it so that litbuild will be able to find it!

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/poetry*whl
```

C.2.14. python-rsa

Name	Python-RSA
Version	4.9
Project URL	https://stuvel.eu/rsa
SCM URL	https://github.com/sybrenstuvel/python-rsa/
Download URL	https://github.com/sybrenstuvel/python-rsa/releases
Dependencies	pyasn1, poetry-core

Python-RSA is a pure-python RSA implementation. The distribution tarfile is sometimes packaged as `python-rsa-version-X.Y`; when this is the case, it will need to be repackaged to conform to the litbuild convention.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/rsa*whl
```

C.2.15. pathspec

Name	Python pathspec library
Version	0.11.2
Project URL	https://github.com/cpburnz/python-pathspec
SCM URL	https://github.com/cpburnz/python-pathspec
Download URL	https://pypi.org/project/pathspec/#files

Pathspec is a library for pattern matching of file paths.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

C.2.16. pluggy

Name	Python plugin and hook calling library
Version	1.3.0
Project URL	https://github.com/pytest-dev/pluggy
SCM URL	https://github.com/pytest-dev/pluggy
Download URL	https://pypi.org/project/pluggy/#files

Pluggy is a library that provides hook-calling mechanisms. It is used by several low-level python modules like `pytest`, `tox`, and `dvipi`.

Configuration commands:

```
(none)
```

Compilation commands:

```
python setup.py build
```

Test commands:

```
(none)
```

Installation commands:

```
python setup.py install
```

C.2.17. editables

Name	Python editable wheel library
Version	0.5
Project URL	https://github.com/pfmoore/editables
SCM URL	https://github.com/pfmoore/editables
Download URL	https://pypi.org/project/editables/#files
Dependencies	flit (flit_core phase)

Editables is a library that facilitates building editable packages. You can use `editables` to create wheel files that install into a package structure that can be edited in place without being rebuilt or reinstalled.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m flit_core.wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/editables*whl
```

C.2.18. calver

Name	calendar version extension
Version	2022.6.26
Project URL	https://github.com/di/calver
SCM URL	https://github.com/di/calver
Download URL	https://pypi.org/project/calver/#files

Some projects are released with version numbers that are just a representation of the calendar date of the release. This is a python package that makes it easy to use calendar version (or "CalVer") version numbers for other python packages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation --wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/calver*whl
```

C.2.19. trove-classifiers

Name	python classifiers
Version	2023.11.29
Project URL	https://github.com/pypa/trove-classifiers
SCM URL	https://github.com/pypa/trove-classifiers
Download URL	https://pypi.org/project/trove-classifiers/#files
Dependencies	calver

Python packages can be classified into categories defined in PEP 301[1]. This project is the canonical source for these classifiers.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation --wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/trove_classifiers*whl
```

C.2.20. hatchling

Name	Hatchling build backend
Version	1.18.0
Project URL	https://pypi.org/project/hatchling/
SCM URL	https://github.com/pypa/hatch
Download URL	https://pypi.org/project/hatchling/#files
Dependencies	pathspec, pluggy, editables, trove-classifiers

Hatch is a build system for python. One of the components that can be built and installed separately is its build module, which is called "hatchling'; that's this.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/hatch*whl
```

C.2.21. colorama

Name	colorama
Version	0.4.6
Project URL	https://github.com/tartley/colorama
SCM URL	*(unknown)*
Download URL	https://github.com/tartley/colorama/releases
Dependencies	hatchling

This is a library for rendering terminal text in multiple colors.

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/color*whl
```

C.2.22. awscli

Name	Amazon Web Services Command Line Interface
Version	1.29.77
Project URL	https://aws.amazon.com/cli/
SCM URL	https://github.com/aws/aws-cli
Download URL	https://pypi.org/project/awscli/#files
Patches	• `awscli-1.29.77-permit-later-dependencies-1.patch`
Dependencies	pyyaml, s3transfer, botocore, python-rsa, colorama

The AWS CLI is a Python program that can be used as a Linux command-line program to manipulate Amazon Web Services resources. It's basically a wrapper for the AWS HTTPS API.

This package declares a number of other python packages it depends on, and is unnecessarily restrictive about what versions it can use for some of them — the practice used by the AWS CLI developers is apparently to specify the *earliest* acceptable version of these dependencies that provides the features that are actually needed, and also specify that nothing *later* than the version they happen to use is acceptable. This is a pretty conservative approach, since it assumes that all future versions of these dependencies will introduce incompatibilities that make it unusable. So far, I've never run into any issues by using the latest stable version of the dependency packages instead.

So *my* practice is to override the practice used by the project team and apply a patch so that the installation process will accept up-to-date versions.

Patch:

- awscli-1.29.77-permit-later-dependencies-1.patch

As is typical for Python programs, there aren't any configuration or test stages.

Configuration commands:

```
(none)
```

Compilation commands:

```
python -m build --no-isolation --wheel
```

Test commands:

```
(none)
```

Installation commands:

```
pip install dist/awscli*whl
```

EC2 instances need to use an early userspace to find their root filesystem, for reasons that we'll describe more fully later on. The early userspace file is presumed to be a CPIO archive, rather than a TAR archive, so we need the `cpio` program in order to set it up.

C.2.23. cpio

Name	GNU cpio
Version	2.14
Project URL	https://www.gnu.org/software/cpio/
SCM URL	*(unknown)*
Download URL	https://ftp.gnu.org/gnu/cpio/

Cpio is an archiving program like `tar`: it creates a single archive file from a bunch of other files and/or directories, or extracts files and directories from archive files. The name is derived from "copy in and out," which is basically what it does: it copies things in and out of archive files.

The two main modes of operation are "copy-out," which takes file and directory paths from its standard input stream and produces an archive byte stream on its standard output; and "copy-in," which takes an archive byte stream on standard input and creates files and directories as specified in that byte stream.

Cpio includes an `rmt` program, just like the one that `tar` provides, so we should tell it not to build or install its own version.

Configuration commands:

```
./configure --prefix=/usr --with-rmt=/usr/libexec/rmt
```

Compilation commands:

```
make
```

Some of the automated tests sometimes fail for me, and I'm not interested enough to investigate why.

Test commands:

```
make -k check || echo "Exit code $?: continuing anyway"
```

Installation commands:

```
make install
```

C.2.24. Kernel Configuration and Build

The kernel configuration we will use for AWS instances should support all the hardware available on those instances, so we'll build a new kernel for these machines.

C.2.25. linux (aws-ami-software phase)

For an overview of linux, see linux.

This configures and builds a Linux kernel intended for use with an AWS (Amazon Web Services) Machine Image. The process of arriving at the configuration here will be discussed a little later on, when we're getting this new version of Little Blue Linux running in AWS; for now, we'll just use the configuration we've patched in to the kernel sources.

Configuration commands:

```
make mrproper
make awsami_defconfig
```

Compilation commands:

```
make all
make Image.gz
```

Test commands:

```
(none)
```

Installation of the kernel and modules is done as per usual.

Installation commands:

```
make modules_install
export KERNELRELEASE=$(ls -1t /lib/modules | head -n 1)
export KERNELPATH=$(find . -name Image.gz -a -type f)
cp -v $KERNELPATH /boot/kernel-$KERNELRELEASE
cp -v .config /boot/config-$KERNELRELEASE
lzip -9 /boot/config-$KERNELRELEASE
cp -v System.map /boot/System.map-$KERNELRELEASE
depmod -e -F /boot/System.map-$KERNELRELEASE -a $KERNELRELEASE
```

C.3. Creating a Bootable Instance

This part is just narrative — a process you'll follow to produce a new AWS instance running the new version of Little Blue Linux. This may involve creating a new AMI, or it may not; there's more than one way to do things. But regardless, it's just a description of the process and instructions that you'll follow to get to a state where you have an AWS instance that can be optimized for use as an EC2 machine image.

The next step after this one — when we optimize this instance so that you can create an AMI from it that works the way AWS recommends — is much more amenable to automation.

CPU Optimizations

To get optimal performance on AMD64 systems, it's a good idea to specify a `-march` GCC compilation flag like `-march=native`. That ensures that the programs built there are optimized to run on the system where the compilation is done, but also means that they may not run properly on other computers with different CPU types: for example, if a program is compiled with the expectation that the CPU has support for the `RDSEED` instruction set (which I think was added in the Broadwell microarchitecture), and you try to run it on a CPU that lacks that instruction, it's not surprising that problems might occur.

The virtual CPU presented to EC2 instances varies by instance family. For example, `t2` instances have Haswell CPUs, while `t3` instances are Skylake or Cascade Lake. As a result, using the compilation flag `-march=native` may not be safe depending on what kind of computer the Little Blue Linux system was built on! You may wish to change the `TARGET_SYSTEM_CFLAGS` parameter to specify `-mtune=native` or `-mtune=intel`... or possibly something else! Look at the `x86 Options` section in the gcc info documentation for a complete discussion of these flags.[2]

Another option you have is to build separate AMIs for different instance families. This might be a good idea if you plan to run some of your workload on AMD instances (like `c5a`) and others on Intel instances (like `c5`). If you do that, you can also get rid of some of the later complexity, like setting up an initramfs image. On the other hand, it's a hassle to recompile everything multiple times to target different types of instance, and managing half-a-dozen AMIs is more work than managing one.

C.3.1. Analyzing simulated hardware

The Linux kernel was actually built along with the other software that will be part of the AMI — but, as promised, I want to talk about how I arrived at the kernel configuration used in it.

Some time ago, you couldn't use your own Linux kernel on EC2 instances; you had to use a kernel provided by Amazon. This changed when HVM instances became available: these treat their root volume like a normal storage device, look for a master boot record and boot loader on it, and generally behave like a real PC-style computer. So, just like with a real computer, we either need to use an early userspace (initramfs or initrd), or ensure that the kernel directly supports the simulated storage device hardware that will be perceived by the virtual machine, as well as the kind of partition table we use for the root device and the root filesystem type.

Aside from that, it's obviously a good idea to have a kernel that supports *all* the hardware devices that will be presented to the virtual machine — otherwise, we won't be using all the capabilities that EC2 is providing. I'm going to walk through that process here, even though you don't need to do any hardware analysis yourself, both because you *will* need to create an instance as I'm doing here — it will become the initial bootable LB Linux instance — and because you might wish to do a similar analysis yourself for some reason.

For the setup of this initial AMI, I'm going to skip the early userspace entirely and just have the kernel mount the root filesystem directly. But to have a full-featured, general-purpose GNU/Linux AMI, you really need to use an early userspace, for reasons that I'll get to later; a later section will walk through setting that up, as we optimize the AMI and make it suitable for use by others and publication in the AWS EC2 Marketplace.

It would be ideal to start with a completely bare-bones kernel — what you get with `make allnoconfig` — and then turn on just the things that are necessary for EC2. That turns out to be very difficult, though, so instead I'm going to start with the kernel configuration from an existing AMI. Then, because I would like to avoid needing to load any modules in an early userspace, I'm going to find all the device drivers that are loaded as modules on the running AMI and build those directly into the kernel. Once I have a working AMI, I'll start turning off things that aren't necessry to get the kernel to boot, and eventually arrive at a minimal kernel configuration that way.

The AMI I'm using for the initial configuration template is the latest Amazon Linux AMI. Amazon Linux is Amazon's own distribution, derived from Redhat Linux. It's not perfect for our purposes, but it's good enough and it's very easy to find: when you click "Launch Instance" in the EC2 console, the top entry in the Quick Start section is the latest Amazon Linux AMI. I've published some Little Blue Linux AMIs in some AWS regions, so you might be able to use one of those instead, if you prefer.

I'm actually going to do the same hardware analysis twice — first, using a `t2.micro` instance, and second after changing the type of the instance to `t3.micro`. The reason I'm doing that is that `t3` instances are a more modern generation of virtual machine — they run on a newer kind of hardware that Amazon calls the "nitro" architecture — and support different simulated hardware. The two big differences are that instead of providing the instance with a Xen front-end network adapter, it provides an Amazon Elastic Network Adapter; and rather than presenting storage as Xen front-end block devices, it presents them as NVMe devices.

NOTE: Creating a `micro` t2 instance in a new AWS account can be free during the first year; Amazon makes a modest set of services available without charge in a "free usage" tier, presumably hoping that you'll get hooked and start using more and more services. This is not the case with `t3` instances — you always have to pay for those. A `t3.micro` instance costs about ten cents USD every hour, so it probably won't break the bank, but it's good to be aware!

I normally use scripts that use the AWS command-line (a python program; in LB Linux, this is the awscli package) or the Terraform program from Hashicorp to manipulate AWS resources, because that makes it easier to automate things and set them up consistently. For a lot of the stuff I'm doing in this section, though, I'm just going to use the web console because I don't expect to be doing it often enough that it makes sense to script it all. Eventually that will probably change — I like the idea of having a continuous delivery pipeline that runs the entire CBL process and the Creating a LB Linux AWS EC2 Machine Image appendix, and automatically publishes new versions of Little Blue Linux as changes are made to these blueprints.

Launching an instance from the console is pretty easy, but these instructions may not work exactly as written! Every so often, AWS changes the console UI enough that these instructions become invalid — if it doesn't work exactly this way, don't panic; you might need to pay more attention and read some documentation, but the process should still be pretty straightforward.

1. From the EC2 console page, select the `Instances` tab and click the big orange "Launch instances" button. This leads to a "Launch an instance" page where you'll fill out all the details you need.
2. Specify a name for the new instance. This will become the initial, pre-optimization LBL instance, so I call it `lbl-bootable-YYYYMMDD`; you can follow that convention or name it whatever else you please.
3. In the "Application and OS Images" section, select the "Amazon Linux" button or put "little blue" in the search box and click the "Community AMIs" tab, which should let you choose one of the Little Blue Linux AMIs that I've already published. (Currently, I publish these only in the Ohio (us-east-2) region, so you'll have to select that region to find them.)
4. In the "Instance type" section, pick the instance type `t2.micro` or `t3.micro`.
5. The SSH key-pair that will be used to access the new instance must be generated by Amazon; after you generate a key pair, you can download the private key, and Amazon will hold on to the public key and make it available to the instance so that it can configure itself to permit SSH access to some account — on Amazon Linux, and on published Little Blue Linux AMIs, this is the `ec2-user` account rather than root, and this is a convention that Amazon urges for other distributions. The private key that will be used to access this new instance is configured in the "Key pair (login)" section here; if you don't have a key pair yet, you can create one by clicking the "Create new key pair" link.
6. In the "Network settings" section, note that "Auto assign public IP" is enabled — that's important, since you want to be able to access the instance from outside of AWS.
7. A "Security Group" is basically a set of firewall rules imposed by Amazon on your instance. You won't see these as network filter rules on the instance, but Amazon will block and permit network traffic as specified by the rules. The "Firewall (security groups)" subsection in "Network settings" lets you specify the initial security group for the new instance; the default selection will create a new security group that allows SSH access from anywhere on the Internet. I suggest restricting SSH connections to your computer's public IP address. You can do

that by just selecting "My IP" in the dropdown.

8. In the "Configure storage" section, you can accept the default size for the root volume. I suggest using the `gp3` storage type; the storage used for `gp3` is an inexpensive and high-performance SSD. You can save a few cents every month by using spinning rust for the root disk; you can do that by changing the volume type to "Magnetic."
9. Also in the storage section, click "Add new volume" and add a second volume — this will become the root filesystem volume for the initial LB Linux instance we create. If you don't want to worry about the details here, just make it a 6-gigabyte `gp3` volume and make sure it is not encrypted. If you want to be more intentional, review the "About the LB Linux Storage Volume" setion below, and choose details appropriately!
10. You don't need to worry about the "Advanced details" section at all, although of course you are free to explore that section as much as you wish.
11. When you're happy with all the details for your new instance, click the orange `Launch instance` button.

That's it! Within a minute or two, your instance will be running. From the `Instances` tab on the console, you can select your new instance and see its public IP address and DNS name.

To analyze the virtual hardware available on the new instance, I:

1. run `dmesg` to obtain the sequence of messages logged by the kernel during boot;
2. copy the kernel configuration from the `/boot` directory to my local machine; and
3. run `lsmod` to show the kernel modules that have been loaded.

If you want to analyze hardware for more than one instance type, you can reuse this instance rather than creating new ones — just shut it down, change its instance type, and start it back up. Then you can re-run `dmesg` and `lsmod` to see what's changed with the new instance type.

A ProTip: Every time you boot an instance, it will wind up with a *different* public IP address. If you find it irritating to keep confirming that you trust the host key when you ssh to the new IP address, you can put a block in `$HOME/.ssh/config` with `StrictHostKeyChecking no` and `UserKnownHostsFile /dev/null` for the hostname pattern `*.compute.amazonaws.com` — that's the domain that Amazon uses for these instances. Alternatively, you can provision an "Elastic IP address" (or "EIP") and associate it with your instances; EIPs are persistent, but cost a little money: as I write this, an EIP costs half a cent USD per hour, which adds up to a little over $3.50 per month. There's also a charge of ten cents USD every time you reassign an EIP after the first hundred assignments in a month.

Once I have the `dmesg` with boot messages, initial kernel configuration, and `lsmod` output, I can do the actual kernel configuration.

As mentioned a moment ago, my preference is to avoid any need to load modules in an early userspace by compiling everything necessary to boot directly into the kernel. Actually, it's more than that: my goal is always to have the typical running state of a GNU/Linux system involve no modules loaded *at all.* I don't object to having some drivers and kernel features built as modules when I might sometimes want them but *usually* won't, but in the most typical case I like to avoid the added overhead imposed by modules. (On most systems, that overhead is negligible, but on a small system like a GnuBee Personal Cloud or Raspberry Pi, it can be significant.)

I'd also prefer for the kernel not to have anything extraneous compiled in to it — that saves RAM, which can be quite valuable on small instance types. So what I'd *like* to do is start with everything turned off, turn on all the things I'll *always* want, and then enable additional things I'll *sometimes* want as modules.

Unfortunately, that's a lot of work: the number of configuration settings you need to tweak to get the kernel to boot at all is quite large. So my fallback plan is to start with an existing AMI's kernel configuration, but with all the modules that wind up loaded into a running system compiled directly into the kernel instead. Once I have that working, I'll iteratively turn off things that I suspect aren't strictly necessary, and try the resulting kernel until I get to one that doesn't boot.

Note, I'm actually doing all this configuration work manually using the interactive kernel configuration tools, and saving the result as a new default configuration! That new `defconfig` file is added to the source tree as a patch. So this section is actually just a discussion of how the configuration was derived. The actual configuration and build was done earlier, along with all the other software built for the AWS AMI.

Instead of starting with the default configuration, or the original CBL configuration, I start by copying the existing Amazon Linux AMI's configuration to `.config` and running `make olddefconfig` so that the default values are used for any new configuration items. From that starting point, I use `make nconfig` and then, for each module I want to build into the kernel, find the configuration setting that enables it and change that setting to compile the feature directly into the kernel rather than building a module for it. I'll walk through `ata_piix` as an example:

1. `git grep ' ata.piix.o$'` shows that `drivers/ata/Makefile` has the line `obj-$(CONFIG_ATA_PIIX) += ata_piix.o`. (I replaced the underscore with a dot in my `grep` command, because sometimes the `.o` file has a hyphen instead of an underscore, and `.` will match either.)
2. In the nconfig session, I hit `F8` to search for a symbol and enter `ATA_PIIX`. It tells me that the configuration setting I want is "Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support" and that it is under "Device Drivers," "Serial ATA and Parallel ATA drivers (libata)," "ATA SFF Support (for legacy IDE and PATA)," "ATA BMDMA support." Descending to that location in the tree, I take everything in the hierarchy that is configured to be built as a module (`M`) and change it to be built in to the kernel (`Y`, so it shows as an asterisk). In this case, that means I change the ATA_PIIX setting and also the ATA setting, which produces the `libata` module.

The same process applies to all the other modules as well. It's a hassle, but not difficult. While going through the modules, I found a case where that process doesn't work quite right: the key `CONFIG_DM_MIRROR` actually causes all three modules `dm_log`, `dm_mirror`, and `dm_region_hash` to be built. I had to take the `$` off my `grep` regular expression to find those.

The device drivers that I am sure are needed on AWS EC2 instances are:

1. `XEN_BLKDEV_FRONTEND`, which allows the kernel to recognize the `/dev/xvd*` devices that are presented to older instance types for EBS volumes;
2. `BLK_DEV_NVME`, which is similar but works with the `/dev/nvme*` devices that appear on nitro instances like those in the c5, m5, and t3 families (and later generations);
3. `ENA_ETHERNET`, which is the network driver for many modern instance families and provides up to 20 Gbps throughput;

4. `IXGBEVF`, which is the network driver for some older instance families and provides up to 10 Gbps throughput; and
5. `XEN_NETDEV_FRONTEND`, which is the network driver for even older instance types and provides lower throughput.

The partition table type and root filesystem for the CBL volume also have to be supported! These are typically the PC BIOS partition table and ext4 filesystem.

Later in this process, I'm going to set up an initramfs for this kernel; when I do that, I could technically build some of those necessary things as modules and load those modules from the early userspace. I'm not going to do that, though, because I want to restrict the early userspace to just a couple of things: expand the root filesystem partition to cover the entire volume if it doesn't already, run `fsck`, and mount the real root filesystem.

That last part — mounting the real root filesystem — is the essential reason we need an initramfs. If you specify the root filesystem on the kernel command line, you have to use a device name that the kernel will recognize directly, and depending on what EC2 instance type is being used, that device might be `/dev/xvda1` or `/dev/nvme0n1p1` or something else entirely.

Once I got the kernel to boot into a functioning userspace on an EC2 instance, I minimized it by disabling everything built as a module and then went through and manually re-added all the stuff I care about.

C.3.2. About the LB Linux Storage Volume

You need an EBS volume that will contain the root filesystem for the new LBL system. This is the purpose of the additional volume you attached to the instance you've just created — you don't *have* to do it at the time you create the instance, but it's most convenient to do it that way. The default suggestion earlier, a six-gigabyte unencrypted `gp3` volume, works fine for this, but let's break that down a little.

I typically encrypt all of the EBS volumes I create, since it's as easy as selecting an option when creating it, but I do not encrypt this particular volume! It will eventually become the root filesystem volume in an LB Linux AMI, and I intend to make that AMI public so it needs to be unencrypted. The root filesystem on instances created from the AMI can be encrypted, even if the AMI itself is based on an unencrypted volume.

There is nothing magical about the six-gigabyte size I use — you can create a larger (or smaller) volume if you like! But you pay by the GB-month, and any instance you create from this AMI must be given a root volume of at least this size. I opt for six gigabytes because the basic LB Linux system fits within that space.

If you launch an instance with a larger root volume than the volume the AMI was created with, you'll have a bit of a hassle actually using the extra space: first you need to expand the partition table so it covers the entire volume, then expand the root filesystem partition likewise — which will *probably* require a reboot so the kernel will recognize the extra space — and then expand the filesystem so it includes that space. This is one of the reasons that an early userspace is worthwhile for EC2 AMIs: from the early userspace environment, you can look to see if there is unused space on the storage device, modify the partition table to use it, expand filesystems, and whatever else

needs to be done. That's something we'll set up in the Optimizing and Improving the Base CBL AMI section.

C.3.3. Optional: Remove Software-Development Files

The base LB Linux system is a self-hosting development server, complete with all the programs and libraries needed to rebuild every package it contains. This is not necessarily what you want for servers, particularly servers that are running Internet-accessible services.

You can save a lot of space — and potentially improve the security profile of your EC2 virtual machines — by removing files that are only used for compiling other programs. (If you do this, you can then add additional packages to servers running in EC2 by building them on a normal LB Linux system and using the `create_binary_tarfile` script to package up the resulting artifacts, then extract that tarfile on the EC2-hosted server.)

The blueprint `remove-development-files` will archive and remove those files for you. After executing that process, you can remove the directory `/root/dev-archive` and be left with a system that can't be used for software construction (but is useful for *running* programs that are built elsewhere).

Of course, you don't have to do that *now* — you can set up a system with whatever server software it needs, and *then* remove development tools from it. The only advantage to purging development files before setting up the AMI is that you then have less data to transfer into AWS.

C.3.4. Packing up Little Blue Linux

Now we need to create a tarfile containing the entire LB Linux system.

NOTE

This is all similar to the process for remastering a disk image. That shouldn't be surprising, since we're basically doing the same thing! You can read more about this in the `qemu-to-qemu-build` blueprint, which also describes how to use multiple disk image files so that the filesystem being packed up is not in use.

However you have this set up, just mount the LB Linux root filesystem at some convenient mountpoint — here, I'm using `/mnt` — then go to the directory above the mount point and pack everything up into a tar archive; I use the commands:

```
tar --create --numeric-owner --file=/lbl.tar --acl --xattrs mnt
lrzip -L9 -D /lbl.tar
```

I use the lrzip program to compress the tar file, because it usually compresses large files better than other programs and runs multiple threads so it's still reasonably fast. It doesn't work well in a pipeline, though, so it's a little less convenient than other compression programs. You can use `lz4` or `lzip` if you prefer (or `gzip` or `bzip2` — the GNU/Linux world has many options for compression programs!) Obviously, if you change the compression program you use, you also have to adjust the later steps to extract the archive.

(The `--xattrs` option given to `tar` above preserves extended file attributes, and `--acl` preserves access control lists. The normal CBL process doesn't use access control lists, but the package users

build process sets an extended attribute with the package name for all installed files and directories so that there's a way to determine what package owns files and directories even if they have been chown'ed and chgrp'ed away from the owning package user. It's also a good idea to preserve all file metadata, just in general. The `tar` command above is the one I always use for full system backups. ACLs in Linux are implemented with extended attributes, but it appears those attributes are not preserved or restored by tar unless the `--acl` argument is used as well? That's a little bit confusing.)

C.3.5. ...to the Cloud!

The same way that you can use pretty much any kind of GNU/Linux system as a host to start a CBL build, you can use pretty much any kind of GNU/Linux system as a temporary staging server to bootstrap an AMI. Here in these instructions, I'm going to use the same Amazon Linux (or LB Linux, or whatever) instance I created earlier, in the "Analyzing simulated hardware" section. If you haven't created an instance, you'll need to go back to that section and create one now.

The same partition table/boot loader discussion from [setup-bootloader-grub] applies here. There's no reason to make the CBL system more complicated than that, so I just create a new PC BIOS partition table with one partition of type "Linux" and create a filesystem on it: `mkfs.ext4 -L lblroot -O ext_attr /dev/nvme1n1p1`. You can also add additional options (described in `man 5 ext4`) if you like. Assuming that grub still has a bug preventing it from accessing filesystems using the `metadata_csum_seed` option, you should make sure to disable that option with `tune2fs -O ^metadata_csum_seed /dev/nvme1n1p1`.

C.3.6. Unpack CBL on the new volume

Mount the lblroot filesystem at `/mnt`.

As of early 2023, Amazon Linux doesn't have `lzip` installed by default, or even available in the default `yum` repositories; it also doesn't have GCC or other development tools installed. If you're using an Amazon Linux instance for this bootstrap, the easiest thing to do is probably to build lzip or lz4 or whatever on your own computer and then `rsync` or `scp` it to the EC2 instance. Of course, if you're using a LB Linux instance, you already have lzip available there.

Copy the `lbl.tar.lrz` file to the instance — again, `rsync` or `scp` can be used for this. Then you just need to unpack it:

```
lrzip -d lbl.tar.lrz
tar --extract --preserve-permissions --numeric-owner \
  --file=lbl.tar --acl --xattrs
```

Once that completes, we need to install the boot loader. To do this, we'll first bind-mount the special filesystems `/dev`, `/sys`, and `/proc` so they're visible at the LB Linux mount points as well as the current locations, then chroot into the LB Linux userspace:

```
mount --bind /dev /mnt/dev
mount --bind /proc /mnt/proc
mount --bind /sys /mnt/sys
```

```
chroot /mnt /bin/bash
```

And re-install grub. If you still have the original LB Linux kernel files in `/boot` and `/lib/modules`, you should probably delete them before running `grub-mkconfig` here — that kernel shouldn't be used, so it is just clutter at this point.

```
grub-install /dev/nvme1n1
grub-mkconfig > /boot/grub/grub.cfg
```

Take a look at the generated grub configuration; when testing, I like to make the timeout a bit longer so I can see the grub boot screen by taking a screenshot (see Diagnosing problems below). Once things are working, I set the timeout to 1 so instances boot as quickly as possible.

On Amazon Linux AMIs, the kernel command line has `console=tty0` and `console=ttyS0,115200n8` parameters, so I add those. I'm pretty sure that both are needed, because `tty0` is *probably* used by the screenshot facility, and `ttyS0` is *definitely* used to populate the system log.

At boot time, the device won't be on `/dev/nvme1n1p1` any longer, so you need to change it. You have two options for this: if you're planning to boot the initial EC2 instance, pre-optimization, as a `t2` instance, you should change it to `/dev/xvda1`, because `t2` instances are given EBS volumes as Xen virtual disks; if you're going to use a `t3` instance (or any other Nitro instance type), you should change it to `/dev/nvme0n1p1` because the volume will still show up as NVMe storage.

To make the final AMI flexible enough to work with any instance type, we'll need to use an early userspace. The linux kernel is only smart enough to interpret its command line `root` directive as a direct device name (like `/dev/nvme1n1p1`); since different instance types present devices with different names, that is not good enough. An early userspace (initramfs or initrd) provides a way around this, because the `init` script run from the early userspace can do anything it needs to do to find and mount the real root filesystem. More on this later, in Optimizing and Improving the Base CBL AMI.

You could also just set up different AMIs for different instance families. That's not necessarily a bad idea, but is definitely more work to prepare. We're not going to do that here.

Once you're done adjusting the grub configuration, remember to add it to the configuration file repository:

```
cfggit add /boot/grub/grub.cfg
cfggit as-default -m "Update grub configuration for aws-ami-bootable"
```

Don't exit the `chroot` shell yet! There's one more tweak we need to do before we boot the new LB Linux instance.

C.3.7. Set root password

We need to be able to log in to the instance after it is launched, so let's set a root password and update `sshd` so that root is allowed to log in. This is strictly temporary: before we're done setting up

the the machine, we'll disable password authentication again and set up SSH public key authentication properly.

NOTE	This step is not *strictly* necessary, because you can now connect to the EC2 serial console even when `sshd` is not running — or even when the network is not accessible! But that capability is relatively new, and I originally wrote this blueprint when there was no way to interact with an instance other than getting to it over the network, and that is still how I typically execute this process.

```
echo 'root:lbluser' | chpasswd
echo 'PermitRootLogin yes' >> /etc/ssh/sshd_config
cfggit stageall
cfggit as-default -m "Set a root password temporarily"
```

Now you can exit the `chroot` shell.

C.3.8. Prepare To Boot The Initial LB Linux

There are two ways to proceed, depending on how you set up the instance you've been working on. Go to the `Instances` page in the AWS console, select that instance so you see its details in the lower pane, and click on the `Storage` tab in those details. If the `Root device details` section shows the root device name as `/dev/sda1`, you can simply swap the volumes around to make the new LBL volume the root filesystem device for the existing instance. If the root device name is shown as `/dev/xvda1` — which will probably be the case if you started with an Amazon Linux AMI — then you have to create a new, intermediate, AMI from a snapshot of the LBL volume.

To be clear — you *can* always take the approach of creating an intermediate AMI! But that is more time-consuming, and creates a number of additional resources that will eventually need to be cleaned up. So my preference and recommendation, when it's possible, is just to swap the volumes around. I'll describe that process first, then the process of creating an intermediate AMI.

C.3.8.1. Swapping Volumes

This is pretty simple! We will simply detach both volumes from the instance we've been using, then re-attach the LBL volume as the primary (and only) storage device.

So shut down the instance you've been using — you can either run `poweroff` as root, or use the `shutdown` action from the AWS console — and wait for its status to show as `Stopped`. Then, in the console:

1. Select the instance so that its details are shown in the lower pane.
2. Select the `Storage` tab.
3. Open the links for both of the currently-attached storage volumes (I open these in new tabs, but you can do whatever you like.)
4. For each of those volumes, make sure it is selected so that its details are shown in the lower pane, then select the `Detach volume` action.

5. For the original root filesystem volume, you can leave it around for now or select the `Delete volume` action — we will no longer need that volume at all. The only reason to keep it around is in case there are issues booting the instance with the LBL volume; you can switch the volumes back to the original configuration to make changes. If you don't destroy the volume now, make sure you do so later! Otherwise AWS will keep charging you a monthly fee for it.
6. For the new LBL root filesystem volume, also select the `Attach volume` action to reattach it to the instance, and override the `Device name` field to give it the value `/dev/sda1`.

Now return to the instance details and refresh the page; you should find that the `Storage` tab shows the LBL volume as the only attached volume. That's all there is to it — you can now use the `Start` action to boot the instance into the LB Linux userspace.

Pretty simple, right? That's why it's my preferred approach. Setting up an intermediate AMI requires additional steps, as we'll see now.

C.3.8.2. Creating a new AMI with LBL as its root device

If you're taking this approach, you can leave the LBL volume attached to the instance you've been using. You can even leave the instance running, if you like, as long as you make sure there are no pending writes to the LBL volume. What we're going to do is take a snapshot of the LBL volume, then register a new AMI based on that snapshot; then we'll be able to launch a new instance from that AMI.

The first thing to do is to make sure there are no pending write operations to the root filesystem volume — if there are any, they won't be captured as part of the snapshot, which can obviously be problematic! To do this, you can simply remount the `/mnt` filesystem read-only with `mount -o remount,ro /mnt`, and then `sync` to make sure that all buffered changes have been written to the persistent store. If you don't trust that approach, you can unmount the `dev sys` and `proc` bind-mounts from earlier, and then unmount `/mnt` altogether, but, honestly, remounting the filesystem read-only and then running `sync` is just as good.

Then go back to the `Volumes` tab in the console and make a snapshot of the LBL volume. This may take a little while to complete — sometimes the new snapshot is ready in a few seconds, sometimes it takes a few minutes. Once its status is `available`, you can select it in the `Snapshots` tab and use the `Actions` menu to create a new image from it. Give it a suitable name (e.g. "LB Linux Bootable YYYY-MM-DD"). Make sure the virtualization type "Hardware-assisted virtualization," which corresponds to the HVM instance type, is selected; and also ensure the Kernel ID is "Use default." Then click `Create`. It may take a few minutes to create the AMI, as well, but often I find they are available almost instantly.

Once the AMI is shown as `Available`, launch a new instance using that AMI. It can otherwise be similar to the instance you launched earlier — although it will not need an additional storage volume.

Regardless of which approach you've taken, you should now have a working LB Linux instance. This is the moment of truth: if all is well, you'll be able to `ssh` into the instance and proceed with the optimization of the AMI. If not… it's probably a good time to turn your attention to the "Diagnosing problems" section below.

At this point, there is no `ec2-user` account on the instance, so you should ssh directly to the `root` user, using the `lbluser` password we set earlier.

NOTE	If it's not immediately possible to `ssh` to the new LB Linux instance, don't panic! Instances created from this initial (pre-optimization) AMI can take several minutes to boot to a working userspace, because the `/dev/random` device is not always initialized as early as it needs to be. We'll take care of that shortly.

C.3.9. Diagnosing problems

If an instance doesn't boot properly, there are a few things you can do to try to figure out what is going on. You can't simply attach a VNC or remote desktop viewer to an instance and control it that way — AWS does not support that kind of remote administration. But sometime in 2021 or 2022, they added the ability to connect to EC2 instances using a serial console, if the instance has a getty process running on `ttyS0`, and that is by far the most convenient approach to dealing with networking or boot problems. If you are viewing instance details in the AWS EC2 web interface, just click the `Connect` button and one of the tabs available there will be `EC2 serial console`. You'll need to enable serial console access, but that's easy to do. I believe it's also an account-wide setting, so you only need to do it once.

If the serial console does not work for you, or you want other options for figuring out what's going on with your instance, keep reading!

In the `Instances` console, you can use the `Actions` menu, `Instance Settings` sub-menu, to see the console output (via `Get System Log`) and to see a screenshot of what is currently shown on the machine console (via `Get Instance Screenshot`).

There is an important caveat to keep in mind, though: although the instance screenshot is always pretty much up-to-date, the system log can lag *substantially*! When diagnosing boot problems on an EC2 instance, I have sometimes seen the system log reported as being completely empty for twenty to thirty minutes after the instance was actually booted.

You can add additional parameters to the kernel command line by editing the grub configuration file. Useful options include:

- boot_delay — set this to an integer between 1 and 10,000, and Linux will delay that many milliseconds after each printk message.
- loglevel — set this to an integer up to 7 (maximum verbosity) to increase the verbosity of kernel logging. I don't know what the default log level is! Probably 5 or 6.
- panic — set this to a number higher than 0, and the kernel will wait that many seconds after a kernel panic before it reboots. Set this to 0, and the kernel won't reboot at all after a panic.
- pause_on_oops — if the instance screenshot shows a bunch of stuff after a problem happens, you can set this to a number of seconds it should pause after the first "oops" so you can see what's going on there.

All of those are potentially most useful with the screenshot facility mentioned above.

You can detach the root volume from an instance that is failing to boot, attach it to a different

instance, and inspect log files to see if anything is written anywhere. That's not likely to be the case if the kernel isn't even starting, obviously. You can also re-attach the volume as the root device on the old instance again, as long as you specify that it should be device `/dev/sda1` — that's the primary mechanism for rescuing an instance that won't boot, in fact: move its root volume to a different instance, monkey with it there, then re-attach it as `/dev/sda1` on the original instance.

Once you have things working, remember to clean up old EBS volumes, snapshots, and AMIs left over from the work you've done — they don't cost much, but clutter is a problem all by itself.

C.4. Optimizing and Improving the Base CBL AMI

Once you have a new LBL instance in a state where it can boot successfully, it's time to optimize it so that it works the way that AWS recommends.

CAUTION	Make sure you set the `LOGIN` parameter correctly before generating scripts from this blueprint! If you have changed `LOGIN` away from the default of `lbl`, the script for Authentication Fixes will crash.

All of these actions need to be taken as root on an instance created from the initial AMI.

A lot of this section implements the guidelines that Amazon publishes for shared or public AMIs, which as of January 2022 are available at:

1. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/building-shared-amis.html
2. https://aws.amazon.com/articles/how-to-share-and-use-public-amis-in-a-secure-manner/ and
3. https://aws.amazon.com/articles/public-ami-publishing-hardening-and-clean-up-requirements/

C.4.1. Miscellaneous Minor Tweaks

C.4.1.1. Flatten the configuration repository

The configuration file repository contains all of the commits produced through the entire CBL build process. Let's flatten that into a single commit; that way, if any of the past configuration work has authorship information or other metadata that you don't want to leak into a public AMI, it will be sanitized away.

We can also do some basic git housekeeping at this point.

Commands:

```
cfggit reset --soft empty
cfggit as-default -m "Add all standard CBL configuration files"
cfggit reflog expire --expire=0 --all
cfggit gc
cfggit prune
cfggit pack-refs
cfggit repack -A -d -f --depth=250 --window=250
```

C.4.1.2. Wait for instance metadata to become available

EC2 provides a set of standard metadata to all EC2 instances. This can be fetched via HTTP from a magical IP address: `169.254.169.254`.

This — along with all other IP addresses in the `169.254.0.0/16` network — is a "link-local" address, as defined in RFC 3927.[3] Link-local addresses are only valid for communication within the network segment or broadcast domain to which the server is connected; routers never forward packets with link-local addresses. AWS frequently uses link-local addresses as a mechanism for permitting instances to communicate directly with the hypervisor where they are running: `169.254.169.254` is used for instance metadata queries, `169.254.169.253` is a DNS resolver, `169.254.169.123` is an NTP server… there are probably others, as well. These addresses are very handy because they are always accessible from EC2 instances, even when those instances are not allowed to communicate with public Internet addresses.

There are two versions of the Instance Metadata Service (IMDS). The preferred version as of January 2022 is the IMDSv2, which has security benefits compared to the original IMDSv1 but requires a little bit of additional complexity: any request for instance metadata must include a "session token," which is generated by a different HTTP request. I like to use the recommended or preferred approach when possible, so the startup scripts I'm using here use this approach.

As with other services that need to use the instance metadata, the service here needs to wait until the network is actually up.

Commands:

```
echo 'network-available' >> /etc/s6-rc/source/set-hostname/dependencies
```

C.4.1.3. Set the hostname based on instance metadata

The `set-hostname` service set up for the basic CBL process always sets the hostname to the same value. New instances launched from an AMI should use an appropriate EC2 hostname instead, though.

The standard EC2 metadata provides at least a hostname that resolves to the instance's private IP address; for instances that have a public IP address as well, there's also a public-hostname metadata entry that we'll prefer to the private hostname when it's available.

When using IMDSv2, session tokens can be valid for up to six hours. This script is pretty fast, though, so here I'm only requesting a token valid for a minute.

File /tmp/set-hostname-up:

```
foreground { s6-echo "Obtaining host name for instance..." }
define imdsurl "http://169.254.169.254/latest/meta-data/"
backtick -E token {
  curl -s -f --request PUT "http://169.254.169.254/latest/api/token"
    -H "X-aws-ec2-metadata-token-ttl-seconds: 60"
}
define tokenhdr "X-aws-ec2-metadata-token: ${token}"
```

```
backtick -E awshostname {
  ifthenelse
    { pipeline {
        curl -H ${tokenhdr} -s -f ${imdsurl}
      }
      grep -q public-hostname
    }
    { curl -H ${tokenhdr} -s -f ${imdsurl}/public-hostname }
    { curl -H ${tokenhdr} -s -f ${imdsurl}/hostname }
}
foreground { s6-hostname ${awshostname} }
s6-echo "Found and set host name '${awshostname}'"
```

Commands:

```
mv /tmp/set-hostname-up /etc/s6-rc/source/set-hostname/up
```

C.4.1.4. Complete text of files

/tmp/set-hostname-up

```
foreground { s6-echo "Obtaining host name for instance..." }
define imdsurl "http://169.254.169.254/latest/meta-data/"
backtick -E token {
  curl -s -f --request PUT "http://169.254.169.254/latest/api/token"
    -H "X-aws-ec2-metadata-token-ttl-seconds: 60"
}
define tokenhdr "X-aws-ec2-metadata-token: ${token}"
backtick -E awshostname {
  ifthenelse
    { pipeline {
        curl -H ${tokenhdr} -s -f ${imdsurl}
      }
      grep -q public-hostname
    }
    { curl -H ${tokenhdr} -s -f ${imdsurl}/public-hostname }
    { curl -H ${tokenhdr} -s -f ${imdsurl}/hostname }
}
foreground { s6-hostname ${awshostname} }
s6-echo "Found and set host name '${awshostname}'"
```

C.4.2. Authentication Fixes

C.4.2.1. Password and user modifications

Amazon suggests removing passwords for system users on shared and public AMIs, and locking the root account so it can't be logged into directly. There's really not much point in doing this — if you can `ssh` to a non-root user like the "ec2-user" account Amazon recommends, and then freely run

any command under `sudo` without a password, it's no different from a security perspective than being able to `ssh` directly to root. So this is a kind of security theater: superficially, it *looks* more secure, so maybe people are more comfortable because of it.

Regardless of how pointless it is, though, if you want to publish an AMI in the AWS Marketplace — as I wish to do, to make it easy for people to spin up CBL instances to use or simply play with — Amazon checks to make sure that the root account is locked, so we'll go ahead and set things up that way.

We'll also remove the lbl user account and create the "ec2-user" account.

Commands:

```
passwd -d root
userdel -r lbl
useradd -c 'EC2 Default User' -m ec2-user
passwd -d ec2-user
echo 'ec2-user ALL=(ALL) NOPASSWD:ALL' > /etc/sudoers.d/ec2-user
chmod 440 /etc/sudoers.d/ec2-user
cfggit add /etc/sudoers.d
cfggit add /home/ec2-user
```

C.4.2.2. Disable password authentication

The default configuration for OpenSSH permits password-based logins. I don't think passwords are ever secure enough for Internet-connected devices, so I like to disable them entirely and force cryptographic key-based authentication. That's especially critical on systems where users don't necessarily have passwords! Here I'm also disabling `sshd` DNS checks, as recommended for shared AMIs.

Commands:

```
sed -i -r -e '/^\#?PasswordAuthentication/s@.*@PasswordAuthentication no@' \
  /etc/ssh/sshd_config
sed -i -r -e '/^\#?PermitRootLogin/s@.*@PermitRootLogin prohibit-password@' \
  /etc/ssh/sshd_config
sed -i -r -e '/^\#?UseDNS/s@.*@UseDNS no@' /etc/ssh/sshd_config
cfggit add /etc/ssh/sshd_config
cfggit stageall
cfggit as-default -m "Authentication fixes for EC2"
```

C.4.2.3. Set up SSH for root account

We need to permit users to log in to new instances using their own key pairs — the same way we were able to log into the Amazon Linux instance using a SSH key.

This service will set up SSH the same way that it's set up for Amazon Linux: you'll be able to use the SSH key to log in to either root or ec2-user, but if you `ssh` to root it will just tell you to use the ec2-user account instead — this is the "security theater" policy mentioned earlier.

The public key that we need to write into the `authorized_keys` file can be obtained as part of the instance metadata made available to all EC2 instances — as we have already used in Miscellaneous Minor Tweaks. As usual for these services, we need the network to be available before it makes sense to try to fetch it.

My execline-fu is not strong, so this script may not be very good!

Service Directory: Oneshot `allow-pubkey-login` *(in bundle* `rl-default`*)*

Dependencies	• `network-available` • `mount-disks`

Up script	`ifelse`
	`  { test -f /root/.ssh/authorized_keys }`
	`  { exit 0 }`
	`redirfd -w 1 /tmp/fetch-pubkey.out`
	`redirfd -w 2 /tmp/fetch-pubkey.err`
	`foreground { mkdir -m 700 -p /root/.ssh }`
	`define`
	`  keyurl`
	`  "http://169.254.169.254/latest/meta-data/public-keys/0/openssh-key"`
	`cd /tmp`
	`foreground { echo "Fetching SSH public key..." }`
	`backtick -E token {`
	`  curl -s -f --request PUT "http://169.254.169.254/latest/api/token"`
	`  -H "X-aws-ec2-metadata-token-ttl-seconds: 60"`
	`}`
	`define tokenhdr "X-aws-ec2-metadata-token: ${token}"`
	`if { curl -f -H ${tokenhdr} -o TMP.pubkey ${keyurl} }`
	`if { grep -q ^ssh TMP.pubkey }`
	`if { chown root:root TMP.pubkey }`
	`if { chmod 600 TMP.pubkey }`
	`if { cp TMP.pubkey /home/ec2-user/.ssh/authorized_keys }`
	`if { chown ec2-user:ec2-user /home/ec2-user/.ssh/authorized_keys }`
	`if { mv TMP.pubkey /root/.ssh/authorized_keys }`
	`sed -i -e "s@^@command=\"echo 'Please login as ec2-user';echo;sleep 10\" @"`
	`  -e "s@^@no-port-forwarding,no-agent-forwarding,no-X11-forwarding,@"`
	`  /root/.ssh/authorized_keys`

C.4.3. Adjust s6-rc init Service Definitions

C.4.3.1. Disable getty Services

As discussed earlier, there's no interactive console capability for EC2 instances. That means there's no point in having any getty processes providing login prompts on the virtual consoles.

Commands:

```
rm -rf /etc/s6-rc/source/getty-tty?
rm -f /etc/s6-rc/source/rl-default/contents.d/getty-tty?
cfggit add /etc/s6-rc/source
```

```
cfggit as-default -m "Service directory changes for EC2"
```

C.4.3.2. Adjust the Basic s6-linux-init Framework

The basic init framework produced by `s6-linux-init-maker` can be adjusted for the AWS environment similarly: there's no point in running an early getty on the virtual terminal `tty1`. But since the EC2 serial console can provide access to an otherwise-inaccessible instance on `ttyS0`, we will move the early getty to run on that device instead.

Commands:

```
pushd /etc/s6-linux-init/current/run-image/service
sed -i -e 's@tty1@ttyS0@' s6-linux-init-early-getty/run
cfggit stageall
cfggit as-default -m "Move early getty to serial device"
popd
```

C.4.4. Configure Chrony NTP Service for EC2

CBL includes blueprints for two packages that can provide support for the Network Time Protocol (NTP): chrony and ntp. I generally prefer chrony. The configuration provided by those blueprints use a pool of NTP servers available on the Internet; for EC2 instances, it's not necessary to use external NTP servers because Amazon provides the Amazon Time Sync Service to all EC2 instances on the link-local address 169.254.169.123. All we need to do here is adjust the configuration file to use it instead of the normal pool servers.

Commands:

```
sed -i -e '/^pool/s@.*@server 169.254.169.123 prefer iburst@' \
  /etc/chrony.conf
cfggit add /etc/chrony.conf
cfggit as-default -m 'reconfigure chrony for EC2'
```

C.4.5. Obtain and run instance "user data"

When you launch an instance in Amazon EC2, you have the option of providing up to 16 kilobytes of arbitrary data that will be made available to the instance for whatever custom processing you need to perform. This is found in the instance metadata as "user data"; you can read about it in the EC2 documentation online.

One (commonly-used) way you can use this capability is to treat the user data as a script that should be downloaded and executed as root the first time an instance is booted. That's how we're going to set up our AMI.

NOTE	You don't *have* to use user data this way! The user data can hold up to 16 kilobytes of whatever you want, and you can use it for whatever you want. Another common way of using user data is with the cloud-init package from Canonical.

Breaking down this oneshot script a bit:

1. Since the user data script should (by convention) only be executed the first time an instance is booted, we're using a flag file `/root/.firstboot` to indicate that this is the initial boot. If the flag file is missing, the service terminates without doing anything else.
2. As with other instance metadata, the link-local address can be used to obtain the user data.
3. In case the user data script emits output that could be useful or interesting, capture its standard output and error in `/tmp/userdata.out` and `/tmp/userdata.err` respectively.
4. If the user data script terminates successfully, we tidy up by removing it. If it terminates with an error code (any nonzero value), we keep it around — again, in case it's useful to inspect it.
5. We remove the flag file regardless.

As with other services that use instance metadata, we need to ensure this service isn't started until the network is up.

Service Directory: Oneshot `run-userdata`

Dependencies	• `network-available`

Up script	`ifelse -n`
	`  { test -f /root/.firstboot }`
	`  { exit 0 }`
	`foreground { echo 'Fetching the user data script...' }`
	`define`
	`  dataurl`
	`  "http://169.254.169.254/latest/user-data"`
	`backtick -E token {`
	`  curl -s -f --request PUT "http://169.254.169.254/latest/api/token"`
	`  -H "X-aws-ec2-metadata-token-ttl-seconds: 60"`
	`}`
	`define tokenhdr "X-aws-ec2-metadata-token: ${token}"`
	`ifthenelse`
	`  { curl -f -H ${tokenhdr} -o /tmp/userdata ${dataurl} }`
	`  {`
	`  foreground { echo "User data found, running it..." }`
	`  redirfd -w 1 /tmp/userdata.out`
	`  redirfd -w 2 /tmp/userdata.err`
	`  if { chmod u+x /tmp/userdata }`
	`  if { /tmp/userdata }`
	`  rm -f /tmp/userdata`
	`  }`
	`  { echo "No user data found" }`
	`rm -f /root/.firstboot`

C.4.6. Rebuild the s6-rc service database

We're done messing with the service database now, so this is a good time to rebuild it.

Commands:

```
pushd /etc/s6-rc
cv=$(ls -1d compiled-v* | sed 's@compiled-v@@' | sort -n | tail -n 1)
nv=$((cv + 1))
s6-rc-compile -v2 /etc/s6-rc/compiled-v${nv} /etc/s6-rc/source
s6-rc-update -v2 /etc/s6-rc/compiled-v${nv}
s6-ln -sfn compiled-v${nv} compiled
unset cv nv
popd
```

C.4.7. Set up early-userspace for AMI

As mentioned earlier, we need an early userspace if we want to have a fully-baked AMI that is usable with all instance types. I'm not going to go into exhaustive detail about how this works; it's documented in the Linux source tree at `Documentation/early-userspace`, and also at `Documentation/filesystems/ramfs-rootfs-initramfs.txt`. Rob Landley, who has collected and contributed a great deal of Linux kernel documentation, wrote a series of articles about how it works and how to use it; you can probably still find those at `https://landley.net/writing/`.

The basic idea, though, is that you can provide an archive of files (in a compressed cpio-format archive) to the Linux kernel, either by attaching it directly to the kernel image file or by having the boot loader load both the kernel file and the cpio archive before it transfers control to the kernel.

If you do this, Linux won't simply mount the root filesystem and execute the `init` program found there; instead, it will expand the cpio archive into a ram-backed filesystem, use that filesystem as the root filesystem, and run the program `/init` found *there*. This program can do whatever it needs to do to set up the real root filesystem, switch to it, and then exec into the real `init` program on it. This feature is called `initramfs`, since it is based on an initial `ramfs` filesystem.

In binary distributions like Redhat, Debian, and their derivatives, this is frequently used with highly modularized kernels: when the initramfs `init` process runs, it figures out what modules need to be loaded to support the hardware it finds, loads those, and then mounts and switches to the real root filesystem. When following the CBL process, since we've compiled everything we need to boot instances directly into the kernel, we don't have to worry about loading modules; all we need to do is locate and mount the real root filesystem, which might be on `/dev/xvda1` or might be on `/dev/nvme0n1p1` depending on which instance type is being used.

The initramfs needs to have everything necessary to mount the root filesystem and transfer control to the real init. To keep it as small as possible, the kernel developers provide a very small C library, klibc, intended for static linking of programs needed in the initramfs; that way, you don't need to include the entire glibc shared library and its dynamic loader. Since I want to do as little work as possible to set up the early userspace, I'm just going to go ahead and include the glibc loader and shared library in the initramfs — that's about 20 megabytes all by itself; if you don't want to include that much unnecessary binary heft for aesthetic reasons, you might want to do the extra work of using klibc, or an alternative C library like musl.

Early Userspace Options

There are actually two different ways of establishing an early userspace. The `initramfs` feature is the most modern approach, and I find it to be the most flexible and simple one, but an older mechanism called `initrd` — which uses an external file that contains a RAM disk block device image, optionally compressed — is still supported and works. It's documented in the Linux source at `Documentation/admin-guide/initrd.rst`, if you're interested in learning more about that alternative.

Another interesting thing about the `initrd` feature is that it's still used as the mechanism for loading an external initramfs image that's not built into the kernel image itself: if an initrd file is provided during the boot process, linux will check to see if it's a compressed cpio archive and will use it as an initramfs source rather than a RAM disk block device if it is.

Recompiling the entire kernel (so the initramfs archive can be attached directly to it) is a bit of a hassle, so here we're going to set up a separate initramfs file and tell GRUB to use it.

As per usual, I want to keep everything as minimalistic and simple as possible, so everything we put into the initramfs will be there because we need to use it in the early userspace.

The easiest way to figure out what we need in early-userspace is to write the program that will be run as `/init`. I'm more comfortable with bash than execline, so I'll make it a bash script.

File /tmp/init:

```
#!/bin/bash
```

Even though the kernel configuration is set to mount `devtmpfs` automatically at boot time, that doesn't appear to happen when using the early userspace. So we have to do that ourselves. We can also mount the `/sys` and `/proc` filesystems — I don't know whether anything running in the `init` script will expect either of those to be mounted, but if I assume something does I won't have to experiment to find out for sure, and it costs essentially nothing to mount them an extra time here.

File /tmp/init (continued):

```
mount -t devtmpfs none /dev
mount -t sysfs none /sys
mount -t proc none /proc
```

We need to find what device the root filesystem is on, obviously. The `findfs` program is useful for that purpose.

File /tmp/init (continued):

```
rootvol=$(findfs LABEL=lblroot)
```

We could just mount the root volume and switch to it at this point, but as long as we're going

through the hassle of setting up an initramfs, we might as well do some other useful stuff. The only thing that comes to mind at the moment is that we should resize the root filesystem if the underlying volume is larger than the instance was when it was shut down (or larger than the AMI's default volume size). Since EBS volumes can be trivially increased in size, making additional space immediately available on the root filesystem is a handy thing to do here, and it's easiest to do during the early userspace before that filesystem is mounted.

(If we wanted to be even more tricky, we could set up the root volume to have a small (a hundred megabytes or so) `/boot` partition as the first partition, and allocate the rest of the space on that volume to an LVM volume group; then we could have the root filesystem be an LVM logical volume, which could be managed more flexibly than a straight partition table. But that's not *vital*, so I'm skipping it.)

Modern versions of the `sfdisk` program included in the `util-linux` package have the capability to expand partitions in place. That's exactly what we want. Of course we need to find the node that represents the full storage device, rather than the partition that includes the root filesystem; that will either be `/dev/xvda` or `/dev/nvme0n1`, depending on instance type. The `lsblk` program has an field that references the "internal parent kernel device name," which is exactly what we want.

File /tmp/init (continued):

```
rootdisk="/dev/$(lsblk -no PKNAME $rootvol)"
echo ', +' | sfdisk -N 1 $rootdisk
```

Now all we need to do is resize the filesystem to take advantage of all available space. If you're using a different filesystem type, like xfs, you'll have to adjust this part of the `init` script, obviously.

File /tmp/init (continued):

```
e2fsck -f -y $rootvol
resize2fs $rootvol
```

That's all we need to do before mounting the real root filesystem, so we can go ahead and do that. Then we can use `switch_root` to make it the root filesystem. Since our init framework expects to have to mount `/proc` and `/sys` itself, we un-mount those first.

Since we want the real s6 supervisor process to wind up running as PID 1, we need to `exec` into the `switch_root` program — it will likewise `exec` into the `init` script we tell it to run, which will eventually `exec` into the s6 supervisor.

File /tmp/init (continued):

```
mount -o ro $rootvol /mnt
umount /sys
umount /proc
exec switch_root /mnt /sbin/init
```

That's it for our initramfs `/init` script! We'll assemble the initramfs in a temporary directory.

Commands:

```
tmpdir=$(mktemp -d)
pushd $tmpdir
mv /tmp/init .
chmod 775 init
```

Looking over the `init` script, we are expecting several directories to exist. Let's create them, along with some other standard directories and the standard `lib` symlinks we use as a multilib kludge.

Commands:

```
mkdir bin dev etc lib mnt proc sys
ln -s lib lib64
ln -s lib lib32
ln -s lib libx32
```

And of course we'll need the programs that are run by the `init` script.

Commands:

```
cp /bin/{bash,lsblk,mount,umount} bin
cp /sbin/{e2fsck,findfs,resize2fs,sfdisk,switch_root} bin
```

Now, this is the trickiest part of building an `initramfs`: we need to find all the shared libraries that are needed by any of those programs and copy *them* into the `lib` directory here.

Commands:

```
for program in bin/*; \
  do \
  ldd $program | egrep -o '/.* ' | while read library; \
  do \
  cp $library lib; \
  done; \
  done
```

Those libraries might also depend on shared libraries; if any of *those* aren't already in the `lib` directory, we need to copy them in as well, and that should be done recursively (since those new dependencies might have additional dependencies of their own, and so on). However! When I was writing this blueprint, I checked and found that these libraries don't introduce any new dependencies of their own, so we don't need to introduce that complexity here.

We need one more thing: `resize2fs` always checks `/etc/mtab` to see if the filesystem being resized is currently mounted. Since there is no `/etc/mtab`, that check fails! In the full CBL system, we set `/etc/mtab` to be a symbolic link to `/proc/self/mounts`, and there's no reason not to do the same thing here.

Commands:

```
ln -s /proc/self/mounts etc/mtab
```

That should be everything we need in our initramfs. Let's pack it up!

Commands:

```
find . | cpio -o -H newc | bzip2 -9 > /boot/initramfs_data.cpio.bz2
popd
rm -rf $tmpdir
```

To tell GRUB to load this initramfs file and provide it to the kernel at boot time, we have to modify the grub configuration file.

Commands:

```
sed -i -e '/linux.*kernel.*awscbl/a initrd /boot/initramfs_data.cpio.bz2' \
  /boot/grub/grub.cfg
cfggit stageall
cfggit as-default -m "Update grub configuration with initramfs"
```

If the early userspace doesn't boot successfully, the most effective thing I've found to do is copy the initramfs file to a real computer and try booting that. It will hopefully have the same issue booting that the EC2 instance does; you can add `rdinit=/bin/bash` to the Linux command line in the grub configuration and boot into an interactive shell rather than running the init script, just as you can use `init=/bin/bash` to bypass the normal init system and just run a rescue shell.

C.4.7.1. Complete text of files

/tmp/init

```
#!/bin/bash
mount -t devtmpfs none /dev
mount -t sysfs none /sys
mount -t proc none /proc
rootvol=$(findfs LABEL=lblroot)
rootdisk="/dev/$(lsblk -no PKNAME $rootvol)"
echo ', +' | sfdisk -N 1 $rootdisk
e2fsck -f -y $rootvol
resize2fs $rootvol
mount -o ro $rootvol /mnt
umount /sys
umount /proc
exec switch_root /mnt /sbin/init
```

C.4.8. Clean up the system for AMI preparation

We need to do this after everything else is done.

Dependencies

Rebuild the s6-rc service database, Set up early-userspace for AMI.

It's nice to start on a freshly-initialized EC2 instance with no lingering log files or history files from previous instances or boots. So let's arrange for that.

Commands:

```
find /var/log -type f -exec truncate -s 0 {} \;
rm -f /root/.*history
rm -f /root/.lesshst
rm -f /root/.viminfo
rm -f ~ec2-user/.*history
rm -f ~ec2-user/.lesshst
rm -f ~ec2-user/.viminfo
```

It's also nice to have each new server set up its own SSH host keys — otherwise, all the instances created from this AMI will claim to be the same server. The s6-rc service defined for OpenSSH will regenerate these keys at boot time. We should also clean up any authorized_keys or other SSH-related files as part of this sanitization process.

Commands:

```
rm -f /etc/ssh/ssh_host*key*
rm -rf /root/.ssh/*
rm -rf ~ec2-user/.ssh/*
```

Since we want to fetch and run the user data script, if there is one, at the first boot of a new AMI, we should create the flag file indicating that this *is* the first boot.

Commands:

```
touch /root/.firstboot
```

Also get rid of any temporary files. The redirects here are useful if the log files for scripts produced by litbuild are being written to a location under `/var/tmp` — we don't want this script to crash just because the log location has suddenly vanished!

We're not bothering to do anything with `/tmp` because CBL uses a tmpfs for that directory. If that's not the way you've got your system set up, add `/tmp` to these commands!

Commands:

```
rm -rf /var/tmp
mkdir /var/tmp > /dev/null 2>&1
```

```
chmod 1777 /var/tmp > /dev/null 2>&1
```

You should repeat this sequence of cleanup steps as part of the preparation process any time you're going to create a new AMI from an instance, immediately before you shut it down to create the new AMI from it!

C.4.9. Creating A New, Optimized AMI

It's straightforward to create a new AMI directly from the instance you've just optimized, since the root filesystem volume is already registered with this instance. Just select the `Create Image` action from the `Image and templates` sub-menu and fill out the UI form in some reasonable way.

As of May 2020, it appears that the resulting AMI will be configured to work with Nitro instance families by default, so you shouldn't need to do anything else. However, if you find that you can't launch some instance types, like `t3`, `m5`, `c5`, or `r5`, then you'll need to follow the rest of the instructions in this section.

Obviously, if you *can* use this AMI with any instance family, you can ignore the rest of this section!

The issue is with advanced networking adapters. The oldest instance types currently supported by AWS present Xen virtual network adapters to the instance, so they need to use Xen Network Front-end driver. However, more modern instance families are given an enhanced networking adapter rather than a Xen adapter — for a complete list of which types provide which adapters, look at the AWS documentation; for a partial list, you can refer to the [linux::aws-ami-bootable] section.

AWS keeps track of which AMIs support these network adapters, and won't allow you to launch instances from an AMI that does not support the corresponding type of interface. If that happens, you'll need to create one of the instance types you *can* use, like one of the `t2` family, and then make a couple of API calls to tell AWS that an instance supports these adapters:

- for the Intel 82599 VF interface supported by many instance types, run `aws ec2 modify-instance-attribute --instance-id YOUR-INSTANCE-ID --sriov-net-support simple`
- for the Elastic Network Adapter supported by the nitro hypervisor, run `aws ec2 modify-instance-attribute --instance-id YOUR-INSTANCE-ID --ena-support`

After you've done this, you can build a new AMI from this instance: this is, again, a straightforward process since you can just use the "Create Image" menu option in the web UI to do so.

It's strange that AWS doesn't provide a way to set the attributes on an AMI directly, rather than on instances launched from the initial AMI. There *is* a `modify-image-attribute` command available, but it only allows a couple of attributes to be changed.

(Of course, that might all have changed by the time you are reading this! AWS enhances and extends their service offerings constantly.)

C.5. Future Changes

As you create, customize, and use instances based on this AMI, you may from time to time want to create new AMIs from the current state of one or more of them. If you've followed along the

narrative so far, you should already know what options you have for doing this, but here's a recap.

If you intend for the new AMI to be public, remember to remove non-root users aside from "ec2-user" and run Clean up the system for AMI preparation right before you shut an instance down to create a new image from it! This — especially the final-cleanup part — is really always a good idea.

Aside from that, there are two basic approaches you can take: you can create a new AMI from the instance you've customized; or you can copy everything from the root filesystem of the instance onto a new EBS volume, and then turn that volume into a new AMI.

The first of these involves far fewer hoops, so that's usually what I opt for. The downside of that approach is that it's difficult to reduce the size of the root volume and there might theoretically be remnants of deleted data on unused sectors of the volume or in the filesystem journal or something. So if you want to start completely fresh, unpacking a tarfile onto a brand new volume might be justified.

The steps for the first approach are:

1. optionally, overwrite all currently-unused space with zeros, by running a command like `dd if=/dev/zero of=/zerofile bs=1M` and then removing the `/zerofile` after the storage volume fills up and the `dd` command terminates;
2. run the commands from Clean up the system for AMI preparation;
3. shut down the instance; and then
4. create a new AMI from that instance

Like I said, very few hoops to jump through! If you want to start with an entirely new volume, you should:

1. shut down the instance and detach its root filesystem volume;
2. attach that volume to a second (running) instance in the same subnet;
3. create a brand new EBS volume and attach it to the instance as well;
4. use a tar pipeline or `cp -a` or something to copy everything from the old root volume to the new root volume;
5. install a boot loader on the new root volume by bind-mounting `dev`, chrooting into it, and running `grub-install`;
6. shut down *that* instance;
7. detach the new root volume from the temporary instance;
8. attach the new root volume as `/dev/sda1` on the original instance; and finally
9. create a new AMI from that instance

That's not that bad, but you can probably see why I tend to use the former approach when possible.

[1] https://peps.python.org/pep-0301/

[2] If you're planning to use t3 instances, which is what I usually do for low-cost server instances, the best CPU setting is `skylake-avx512`. There is also a `cascadelake` setting, which enables the `AVX512VNNI` instruction set, but I am pretty sure that's only supported on *some* t3 instances, and I don't know of any way to control whether you get one of those.

[3] Outside of AWS, the typical situation where link-local addresses are used is for autoconfiguration of network interfaces when no external means of configuring IP addresses are available. We don't need to use them for this on EC2 instances, since EC2 always configures network interfaces via DHCP.