Linux by Intent

Version draft-r0.2

Published April 16th, 2026

Created by Someone Who Decided This Was Necessary (SFG545)

Managing Editor: still me, unfortunately

Table of Contents

4. **7. Chroot and Remaining System Utilities**
    - 7.1 Introduction
    - 7.2 Reset Target Tree Ownership to root
    - 7.3 Create virtual filesystem link targets
    - 7.4 Copy selected build variables and helper functions into target profile
    - 7.5 Enter chroot environment
    - 7.6 Create essential system files
    - 7.7 gettext-tiny 0.3.3
    - 7.8 byacc 20260126
    - 7.9 python 3.14.4
    - 7.10 ubase git snapshot
    - 7.11 Cleanup
5. **8. Compiling the Remaining utilities for the system**
    - 8.1 Introduction
    - 8.2 iana-etc 20260409
    - 8.3 musl libc final pass 1.2.6
    - 8.4 pigz stage 2 2.8
    - 8.5 xz stage 2 5.8.3
    - 8.6 zstd stage 2 1.5.7
    - 8.7 file stage 2 5.47
    - 8.8 bc 7.0.3
    - 8.9 pkgconf 2.5.1
    - 8.10 Shadow 4.19.4
    - 8.11 ncurses stage 2 6.6-20260418
    - 8.12 byacc stage 2 20260126
    - 8.13 bsdgrep stage 2 master snapshot
    - 8.14 LibreSSL 4.2.1
    - 8.14.1 ca-certificates 2026-03-19
    - 8.15 Flex 2.6.4
    - 8.16 SQLite 3.53.0
    - 8.17 python 3.14.4
    - 8.18 Python-Flit-Core 3.12.0
    - 8.19 bfs stage 2 4.1

Audience

Audience

This section explains who the book is for, which turns out to be readers who looked at Linux From Scratch and decided the main thing missing was a little more adversity.

Open standalone page

> **Short version:** this is for people who did not quite hate life enough to stop, yet still looked at LFS and thought it was being a bit too accommodating.

This book is for readers who already know what Linux From Scratch is, understand why someone would do it, and somehow concluded that the concept still had room to become more opinionated. If LFS struck you as educational but just a little too polite, then yes, you are very much in the target audience.

It assumes a reader who is comfortable around shells, source builds, compiler toolchains, partitioning, boot problems, and the general principle that when the machine becomes unreasonable, you are the nearest qualified adult. You do not need to be a compiler engineer, a kernel developer, or someone who reads ABI notes for recreation. But you do need to be willing to read errors carefully, look things up, and fix your own mess with something resembling composure.

This is not meant as a first introduction to Linux. If your current strategy for unfamiliar terms is to continue confidently until the system either boots or catches fire, the book may still teach you something, but it will do so in the tone of a disappointed physics experiment.

> If Linux From Scratch looked useful but insufficiently adversarial, this book may be exactly the questionable decision you were hoping to make.

The intended reader is patient, moderately stubborn, and interested in understanding why the system works rather than merely confirming that it currently does. The book is especially aimed at people who want an LLVM-centered build on purpose, who are willing to question inherited filesystem layout, and who prefer deliberate choices over historical leftovers with good branding.

In short, this is for readers who value intent, traceability, and the occasional regrettable quantity of manual labor. If that sounds appealing, welcome. If it sounds exhausting, that is also useful information, and likely the kindest diagnostic this project is going to provide.

1.1 Why

1.1. Why

This section explains the motivation for the project, which would have remained mercifully short if “because I felt like it” had been permitted to stand without supervision.

Open standalone page

> **Short version:** because I felt like it, which was somehow judged inadequate and made to attend documentation.

Every project prefers an origin story with posture. It wants to begin with a requirement, a need, a measurable deficiency, or at least a sentence that sounds expensive. This one began with something much less polished. I wanted to build it. That is the reason before it put on formal clothes and started pretending to have a strategic outlook.

That answer may sound unserious, but it is perfectly serviceable. Curiosity is still a motive. Mild irritation with opaque systems is still a motive. The desire to inspect the machinery instead of being asked to trust the appliance is also a motive, and arguably a better one than whatever a committee would have written after three meetings and a slide deck.

> There are easier ways to obtain a Linux system. Naturally, that is not the point. The point is to replace convenience with understanding, then act surprised when it takes longer.

Building the system by hand forces each component to stop hiding in the crowd. Libraries, toolchain pieces, configuration choices, and old assumptions all have to present themselves one by one. Some of them turn out to be essential. Some turn out to be inherited habits with excellent public relations.

There is also a quieter reason. A finished distribution tells you what someone else thought was reasonable. A hand-built system tells you what you are willing to understand, maintain, and debug once the pleasant idea of the project has been replaced by the project itself. That answer is usually more honest than the polished one.

So the full explanation remains what it was from the beginning. I felt like it. Then, in keeping with the spirit of unnecessary elaboration, that impulse was expanded into a section and given enough structure to pass for intent.

1.2 Scope

1.2. Scope

This section defines the boundary of the book so the project does not quietly grow from “build a usable system” into “reinvent every tool ever shipped because restraint is for other people.”

Open standalone page

> **In scope:** bootstrap with LLVM, Clang, libc++, and compiler-rt; install enough userland to get a working system; then build and install the Linux kernel.

The book covers a deliberately limited target. The goal is not to produce an encyclopedic distribution, a general-purpose desktop, or a monument to package count. The goal is to bootstrap a functioning system with an LLVM-centered toolchain, install the pieces required for that system to operate, and carry the process far enough that the machine can actually boot into something useful instead of merely being philosophically complete.

The bootstrap path will center on LLVM and its immediate ecosystem: LLVM itself, Clang as the compiler front end, libc++ as the C++ standard library, and compiler-rt for runtime support. That is the spine of the build. The intent is not to sprinkle these parts into a mostly conventional GNU toolchain and pretend something interesting happened. The intent is to make them foundational and follow the consequences wherever they lead.

> In other words, the system is being built around LLVM on purpose, not by accident, and certainly not because the easier route seemed insufficiently annoying.

Past the toolchain, the book will install enough software to make the system behave like a real machine rather than an elaborate proof of concept. That means the core userland, the libraries and utilities needed to compile, configure, inspect, and operate the system, and the basic layout required for boot, login, and routine maintenance. It does not mean every optional package that might someday become desirable after the first successful boot.

The endpoint is the Linux kernel. The kernel is the one explicit exception to the broader licensing preference behind the project, because refusing to use it would be an impressively committed way to stop building Linux while insisting that Linux was still the subject. So the kernel stays. It will be configured, built, and installed as part of the system, because eventually the book needs to produce more than a directory tree and a sense of moral accomplishment.

Everything outside that boundary is deferred unless it proves necessary for the system to function. The point is to reach a usable base system with a clear toolchain story, not to wander into endless expansion because there is always one more package that seems harmless right before it costs an afternoon.

1.3 Licenses and the FHS

1.3. Licenses and the FHS

This section explains the licensing policy behind the book and the filesystem conventions it will, and will not, follow.

Open standalone page

Let us say you release a browser. Good for you. Amazing. You publish the source tree with no license and no code headers, assume everyone will understand your intentions, and then act surprised when the situation becomes a mess. The first correction is important: without a license, other people do not receive permission to copy, modify, or redistribute your work. That does not protect you by magic; it mostly creates confusion, blocks legitimate reuse, and leaves everyone arguing about what is allowed.

A license is the part where you stop relying on vibes and state the legal terms plainly. It tells other people what they may do with your code, what obligations attach to redistribution, and what liability you are refusing to accept. If you want software to be usable by other people, licensing is not optional paperwork. It is the mechanism that makes distribution and reuse legally intelligible.

> **Practical rule:** no license means no permission. A real license creates permissions, limits, and obligations in writing.

Copyleft

Copyleft licenses are licenses that preserve downstream freedoms by requiring certain redistributed derivative works to remain under the same license, or under terms that preserve the same rights. The GNU General Public License is the standard example. The simplified version many people remember is directionally right but incomplete: copyleft does not mean every modification must be published immediately. The obligation is usually triggered when the modified work is conveyed or distributed to others, not when it is used privately.

The GPL is therefore not simply “a license that forces any derived software to be released.” More accurately, it requires that recipients of a distributed derivative work receive the corresponding source code and the same GPL rights. That distinction matters. Private internal modifications are one thing. Distribution to other parties is another.

This book is intentionally avoiding strong copyleft in the userland where practical. That is a project policy choice, not a claim that copyleft is invalid or unworkable. Copyleft is effective at keeping code and modifications available to recipients. The tradeoff is that it imposes reciprocal conditions the project does not want as its default licensing posture.

Permissive and Weak Copyleft Licenses

The preferred licenses for this book are permissive licenses and, in limited cases, weak copyleft licenses. Permissive licenses generally allow reuse in proprietary and open source projects alike, subject to modest conditions such as preserving notices, disclaimers, or attribution text.

- The BSD licenses are acceptable. In practice this usually means the 2-clause or 3-clause BSD variants, which are short, permissive, and easy to satisfy.
- The MIT license is acceptable. It is one of the simplest permissive licenses in common use and is broadly compatible with mixed-source environments.
- The Apache License 2.0 is acceptable. It is permissive, includes an express patent grant, and is often preferable when patent language matters.
- The Mozilla Public License 2.0 may be acceptable in some cases. It is not a permissive license; it is a weak copyleft license with file-level reciprocity. That means modified MPL-covered files generally remain under the MPL when distributed, while larger combined works may use other licenses for separate files.
- The PostgreSQL license and ISC license would also fit the overall policy, even if they are not the main examples listed here.
- The SQLite blessing is usable as a public-domain style dedication, but it is unusual. Where legal certainty matters across jurisdictions, an established permissive license is often simpler to evaluate.

> In short: BSD, MIT, and Apache are the default comfort zone. MPL is only acceptable when its file-level copyleft is understood and deliberately tolerated.

There is one explicit exception to the broader preference against copyleft software: the Linux kernel. The project is building a Linux system, so excluding the kernel on licensing grounds would reduce the

exercise to a very elaborate refusal to finish the sentence. The kernel remains in scope, and its GPL obligations are accepted as part of using it.

The FHS

The Filesystem Hierarchy Standard attempts to define conventional locations for binaries, libraries, configuration files, variable data, temporary files, and other system components. Its purpose is predictability. A system that follows the FHS closely is easier for administrators, packages, and third-party tooling to reason about because file placement is less surprising.

This book is not committed to following the FHS strictly. The reason is straightforward: the project does not want to. That answer can be made more formal by saying the filesystem layout should serve the build and maintenance model of this system rather than satisfy a standard whose priorities are not automatically ours. The short version is still the accurate one.

That does not mean the layout will be random. It means the book reserves the right to choose directory structure deliberately, even when that structure diverges from FHS conventions. Compatibility concerns will still matter, especially where software expects particular paths, but the standard itself is not being treated as binding law.

Readers should therefore expect occasional departures from conventional directory placement when those departures make the system easier to understand, maintain, or bootstrap. Standards are useful tools. They are not obligations unless a project decides they are.

2.1 Host Resources and Readiness

2.1. Host Resources and Readiness

Before the build begins, the host system must provide the required development tools and should be verified. This section combines the host resource requirements with the readiness checks used to confirm that the machine can actually begin the build.

Open standalone page

> **Required host tools:** a POSIX-compatible shell, a working C and C++ compiler, `git`, `gzip`, `awk`, a yacc-compatible parser generator, CMake, Meson, `rustc`, `cargo`, and `rsync`.

The build described in this book assumes that the host environment is already capable of building nontrivial software. The host system is not expected to match the final target system, but it must be sufficiently complete to configure source trees, compile code, and run the build systems used by the packages in the early chapters.

A usable `sh` implementation is required for basic scripting and package build logic. Many configure scripts, bootstrap steps, and helper tools assume the presence of a POSIX shell and will fail

immediately or behave incorrectly when the shell is absent or nonconforming. The exact implementation is less important than correctness.

The host must also provide a working C and C++ compiler. This requirement is fundamental. The compiler does not need to match the toolchain ultimately produced by the book, but it must be reliable enough to build the initial stages of the LLVM-based stack. A broken or partially functional compiler will produce misleading failures later in the process, where diagnosis is significantly harder.

CMake is required because several components in the LLVM ecosystem use it as their primary build system. Without CMake, the bootstrap path stops early. The version available on the host should be recent enough to support the LLVM release being built; using an arbitrarily old version is an efficient way to introduce avoidable configuration errors.

Meson is also required. A growing number of modern packages use it for configuration and build orchestration, and it is part of the expected toolset for a contemporary source-based environment. Its absence is not theoretical; it will block real packages needed to assemble a usable system.

The host should also provide Rust's core build tools: `rustc` and `cargo`. They are increasingly required by modern developer tooling and by packages that include Rust components even when the larger project is not primarily written in Rust. If either is missing, the build eventually stops in a much less entertaining place.

`git` and `gzip` are required by the source staging helper when a manifest entry points at a pinned upstream git commit instead of a release tarball. `awk` and a yacc-compatible parser generator are required by small userland packages that generate build-time source files.

`rsync` is also required. Section 5.1 (Linux API headers) uses it to copy header trees predictably while preserving metadata.

`clang`, and `ninja`, are not strictly required, but they are common tools that can be used to speed up the build process. If they are available, the build scripts will use them automatically. If they are missing, the build will fall back to more basic tools like `gcc` and `make`, which may be slower but still functional.

Common Host Package Sets

On most distributions, the fastest way to assemble a usable host is to start from the distribution's standard development meta-package or package group, then verify the exact tools required by this book. These package sets are starting points, not guarantees.

- Arch Linux and Arch-based systems: `base-devel`.
- Debian and Ubuntu: `build-essential`.
- Fedora, RHEL, Rocky, and AlmaLinux families: the `Development Tools` group.
- Alpine Linux: `build-base`.
- Void Linux: `base-devel`.

These package sets usually provide a compiler, linker, libc development files, `make`, `awk`, and part of the classic build stack. They do not always include everything needed here. In particular, `cmake` and `meson` are often missing from older or more conservative development groups, yacc-compatible parser generators may be packaged as `byacc` or as a `yacc` command provided by another package, and the Rust toolchain is frequently packaged separately.

For that reason, the host should still be checked explicitly even after the distribution meta-package is installed. If the package set is incomplete, add the missing tools individually and rerun the readiness checks.

> **Provided scripts:** one implementation each for `sh`, `zsh`, and `fish`. All three check shell availability, required tool presence, C and C++ compilation, and basic linking.

These scripts are intended to be run on the host system before following the rest of the book. They perform four categories of checks.

- They verify that `sh` is present and can execute a trivial command.
- They locate a working C compiler and C++ compiler.
- They compile and run small C and C++ test programs.
- They compile object files separately and link them into an executable to confirm that linking works.
- They check for the required build tools: `git`, `gzip`, `awk`, a yacc-compatible parser generator, `cmake`, `meson`, `rustc`, `cargo`, and `rsync`.

The scripts are available at the following paths:

- scripts/check-ready.sh
- scripts/check-ready.zsh
- scripts/check-ready.fish

Each script prints a pass or fail result for every check and exits with a nonzero status if any requirement is missing or any compile or link test fails. The checks are intentionally small. The goal is not to exhaustively validate the host system; the goal is to detect obvious blockers before the real build starts.

> If one of these scripts fails, fix the host first. Continuing with a broken shell, incomplete toolchain, or missing build system only turns a simple prerequisite problem into a harder diagnostic problem later.

The `sh` script is the baseline implementation and should work on any reasonably POSIX-conforming system. The `zsh` and `fish` variants exist for users who prefer to run the readiness check from their normal interactive environment without translating syntax by hand.

Other utilities will be introduced as needed in later chapters, but the list above establishes the baseline. If the host cannot provide a shell, a functioning compiler, and the build systems named here, it is not

ready to begin bootstrapping.

2.2 Package Management

2.2. Package Management

bruh we are NOT using a package manager. too easy.

Open standalone page

no

3.1 Partition and Directory Layout

3.1. Partition and Directory Layout

Before filesystems are created or packages are installed, the target disk layout and the corresponding mount directories should be planned explicitly.

Open standalone page

> **Minimum layout:** a root filesystem and a boot path that the reader understands. A new dedicated EFI System Partition is optional; reusing an existing ESP is perfectly acceptable.

The book does not require a single fixed partition scheme. Readers may use a minimal layout or split the system across multiple filesystems, provided the final boot path exists and the mount layout remains coherent. What matters is that the choices are made deliberately before the build begins, because the directory structure created later will depend on them.

> **TODO:** this book still needs a full partitioning walkthrough of its own. For now, readers should use one of these references for the actual disk setup steps before continuing:
>
> - Gentoo Handbook: Preparing the disks
> - ArchWiki: Partitioning

Required Partitions

The only partition that is always mandatory is the root filesystem. It contains the target system itself and remains required regardless of how many other filesystems are introduced later.

On UEFI systems, the finished installation also needs access to an EFI System Partition so the bootloader and related files have somewhere to live. That does not mean the reader must create a fresh ESP specifically for this book. Reusing an existing ESP is fine if its size, mount policy, and ownership expectations are already understood.

Optional Layout Choices

The simplest supported arrangement is a root filesystem plus whatever boot partition arrangement the machine already uses. For many systems that will mean a root filesystem and an existing ESP. For others it may mean a new ESP created just for this installation. The book can work with either choice.

Readers who prefer a split layout may also create additional filesystems, such as a separate `/home`, `/var`, or another dedicated mount point. The book will not require those extra partitions, but it will not prevent them either. If a separate filesystem is created, the corresponding mount directory must also exist in the target layout.

> A split layout should exist because it serves a maintenance or operational purpose, not because adding partitions feels like progress.

Top-Level Directory Scheme

This book is not planning to mirror the FHS directory names one-for-one. The target layout should use a clearer, deliberately named top-level structure instead of inheriting every traditional path just because history left it lying around.

The intended core directory scheme is under `/system`:

- `/system/configuration` for system configuration data.
- `/system/binaries` for general user-facing executable programs.
- `/system/systembinaries` for executables needed for boot, recovery, and system administration.
- `/system/libraries` for shared and static libraries needed by the installed system.
- `/system/headers` for development headers.
- `/system/share` for architecture-independent package data such as terminfo records.
- `/system/documentation` for packaged manuals, reference material, and other installed documentation.
- `/system/tools` for the helper toolchain used during the build, which may be removed or repurposed after the final system takes over.
- `/system/variable` - an optional directory for variable data, logs, and other runtime files that do not fit into the above categories.
- Traditional FHS paths are compatibility links into `/system`, for example: `/etc -> /system/configuration`, `/bin -> /system/binaries`, `/sbin -> /system/systembinaries`, `/lib -> /system/libraries`, `/var -> /system/variable`, `/home -> /system/users`, and `/root -> /system/charlie`. Common `/usr/*` paths should

likewise link into `/system/*`, and `/system/lib` should link to `/system/libraries` for packages that derive old-style paths from their configured prefix.

- `/devices, /processes, /system, and others are virtual filesystems that are mounted at runtime and do not need to be created or populated during the build, but will be link targets for /dev, etc.`
- `/system/charlie` - freebsd has a joke about root's full name, 'charlie &', here, `/root` should be a link to it
- `/system/users` - a directory for user home directories, `/home` should be a link to it

Other directories may still exist where they make operational sense, especially for boot and mount structure. In practice, that means paths such as `/boot`, `/efi`, and any other explicitly chosen mount points may still appear alongside the custom layout. The point is not to ban every conventional name on sight. The point is to make the primary system layout intentional.

Filesystem Tooling

The filesystem tools required on the host depend on which target filesystems the reader chooses. The book does not force a single root filesystem, so the matching userspace tools must be present for the selected format.

- EFI System Partition, if one is being created or reformatted: `dosfstools`.
- ext2, ext3, or ext4: `e2fsprogs`.
- XFS: `xfsprogs`.
- Btrfs: `btrfs-progs`.
- F2FS: `f2fs-tools`.
- Swap space, if used: the host must provide the tools that create and manage swap, commonly from `util-linux`.

The practical rule is simple: if the target disk layout includes a filesystem, the host must have the userspace tools needed to create it. That should be confirmed before any disk changes are made.

Directories

Once the partition plan is chosen, the mount directory layout should mirror it. At minimum, the target root mount directory is required. If the installation uses an ESP, the chosen mount point for that partition must also exist, whether it is `/boot`, `/efi`, or another deliberate location. The custom base directories described above should exist under `/system` in the target tree: `/system/configuration`, `/system/binaries`, `/system/systembinaries`, `/system/libraries`, `/system/headers`, `/system/share`, and `/system/documentation`. Any optional partition or dedicated mount point should then map onto one of those paths, or onto another path the reader has chosen on purpose.

Later sections will describe how the target tree is mounted and populated. For now, the important point is that the directory layout follows the partition layout, not the other way around.

3.2. Build Variables and Environment Files

The build environment should be explicit, shell-specific, and temporary. The supplied env files keep the book's variables out of the user's normal login setup and clear common host build overrides before work begins.

Open standalone page

> **Policy:** put the book's variables in a dedicated build env file, not in `~/.profile`, `~/.zshrc`, or `config.fish`. When the build shell ends, the book-specific environment should end with it.

A hand-built system is easier to reason about when the environment is narrow and intentional. Exporting build variables in a normal interactive shell and forgetting about them is a reliable way to create later confusion, because the next unrelated package build or shell session inherits choices that belonged only to this project.

For that reason, this book uses shell-specific env files. They set the variables the build needs, clear the host overrides most likely to leak into toolchain work, and leave room for local customization without forcing everyone into the same directory names or mirror choices.

Provided Files

The repository includes one base env file for each supported shell and one matching example override file for local customization:

- scripts/build-env.sh
- scripts/build-env.zsh
- scripts/build-env.fish
- scripts/build-env.local.sh.example
- scripts/build-env.local.zsh.example
- scripts/build-env.local.fish.example

The base files establish a predictable baseline. The example override files exist for values that are site-specific or personal, such as a local source mirror, a different root directory, or custom compiler flags that should be kept separate from the defaults.

Variables Used by the Book

The environment files use `LBI_` and `LWI_` prefixes so the variables are obviously tied to *Linux by Intent* and less likely to collide with unrelated tooling.

- `LBI_ROOT`: top-level working directory for the build.

- `LBI_SOURCES`: where source archives and unpacked trees are kept.
- `LBI_SYSROOT`: target system root while the build is being assembled.
- `LBI_TOOLS`: helper toolchain prefix used before the final system takes over.
- `LBI_BOOT`: chosen boot mount point inside the target layout.
- `LBI_ESP_MOUNT`: where an ESP is mounted when one is used.
- `LBI_ARCH`: target architecture component used when composing the default target triple.
- `LBI_TARGET`: target triple for cross and staged builds.
- `LWI_MAKE_JOBS`: preferred parallel job count for tools that require a numeric value.
- `LWI_MAKE_FLAGS`: preferred make-style parallel flags used by `make`, `bmake`, `ninja`, and `cmake --build` examples.
- `LWI_CFLAGS`, `LWI_CXXFLAGS`, and `LBI_CUSTOM_LDFLAGS`: optional local tuning hooks that stay outside the default build policy.

Stopping Host Variable Leakage

The supplied env files explicitly clear common build-time variables before defining the book's defaults. In practice, that means host settings such as `CC`, `CXX`, `CPPFLAGS`, `CFLAGS`, `CXXFLAGS`, `LDFLAGS`, `LD_LIBRARY_PATH`, `PKG_CONFIG_PATH`, `PKG_CONFIG_LIBDIR`, and `DESTDIR` do not quietly bleed into the bootstrap unless the reader chooses to set them again on purpose.

> The goal is not to pretend the host environment does not exist. The goal is to stop accidental host preferences from impersonating book policy.

Representative Defaults

```
unset CC CXX CPPFLAGS CFLAGS CXXFLAGS LDFLAGS
unset LD_LIBRARY_PATH PKG_CONFIG_PATH PKG_CONFIG_LIBDIR DESTDIR

export LBI_ROOT="${LBI_ROOT:-$HOME/linux-by-intent}"
export LBI_SOURCES="${LBI_SOURCES:-$LBI_ROOT/sources}"
export LBI_TOOLS="${LBI_TOOLS:-$LBI_ROOT/tools}"
export LBI_BOOT="${LBI_BOOT:-/boot}"
export LBI_ESP_MOUNT="${LBI_ESP_MOUNT:-$LBI_SYSROOT/boot/efi}"
export LBI_ARCH="${LBI_ARCH:-x86_64}"
export LBI_TARGET="${LBI_TARGET:-$LBI_ARCH-lbi-linux-musl}"
export LWI_MAKE_JOBS="${LWI_MAKE_JOBS:-$(getconf _NPROCESSORS_ONLN 2>/dev/null |
export LWI_MAKE_FLAGS="${LWI_MAKE_FLAGS:--j$LWI_MAKE_JOBS}"
```

The fish variant expresses the same policy with fish-native syntax, and the zsh variant follows the same defaults while preserving normal zsh behavior. The values themselves may be changed, but they

should be changed in one of the provided env files or in a local override file, not spread across login startup files and half-remembered shell history.

Custom Variables

Local customization belongs in the matching example override file. That keeps personal choices visible and contained instead of silently becoming the documented default for everyone else.

```
export LWI_CFLAGS="-O2 -pipe"
export LWI_CXXFLAGS="$LWI_CFLAGS"
export LBI_CUSTOM_LDFLAGS=""
export LBI_BOOTLOADER_ID="LinuxByIntent"
```

Readers who already have their own variable naming scheme may keep using it if they want, but the book will speak in terms of the `LBI_` variables so that examples, notes, and scripts stay internally consistent.

Reusable Build-System Functions

The base env files also define helper functions for the three most common build systems used in this project:

- `lbi_configure` for Autotools-style `./configure` projects.
- `lbi_meson` for Meson projects.
- `lbi_cmake` for CMake projects.

Each helper applies the custom directory layout defaults from section 3.1 under the `/system` prefix (`/system/binaries`, `/system/systembinaries`, `/system/libraries`, `/system/headers`, `/system/configuration`, `/system/variable`, and documentation paths under `/system/documentation`) so package installs follow the book's intended structure.

When installing from the host build environment, use `DESTDIR="$LBI_ROOT"` with packages configured by these helpers. The helpers already place the runtime prefix under `/system`.

All three helpers accept extra flags. Extra arguments are forwarded directly to the underlying tool, so you can add project-specific options without rewriting the common layout flags each time.

For `lbi_meson` and `lbi_cmake`, the first positional argument may be used as the build directory. If omitted, they default to `build`.

```
# Autotools
lbi_configure --build="$LBI_TARGET" --disable-nls

# Meson (explicit build directory)
```

```
lbi_meson build --buildtype=release -Db_lto=true

# CMake (default build directory, extra generator/flags)
lbi_cmake -G Ninja -DCMAKE_BUILD_TYPE=Release
```

Makeflags and Parallel Jobs

Set `LWI_MAKE_FLAGS` for build commands throughout the book, and keep `LWI_MAKE_JOBS` for tools that require a numeric job value.

Typical setup:

```
export LWI_MAKE_JOBS="$(getconf _NPROCESSORS_ONLN 2>/dev/null || printf '1')"
export LWI_MAKE_FLAGS="-j$LWI_MAKE_JOBS"
```

Choosing `LBI_CUSTOM_*` Values

If you want to tune builds, do it deliberately and keep notes about why. The safest path is to start simple, build once, and only then add aggressive options.

In practice, most readers are choosing between a few repeatable profiles rather than inventing flags from scratch:

Profile	`LWI_CFLAGS`	`LWI_CXXFLAGS`	`LBI_CUSTOM_LDFLAGS`	Best fit	Main caution
Recommended starting point	`-O2 -pipe`	`$LWI_CFLAGS`	*(empty)*	First full build, stable troubleshooting, and easy comparison with upstream defaults.	Usually not the absolute fastest result.
Portable x86-64	`-O2 -pipe -mtune=generic`	`$LWI_CFLAGS`	*(empty)*	The image may move between different x86-64 systems.	Gives up some CPU-specific optimization.
Native CPU tuning	`-O2 -pipe -march=native`	`$LWI_CFLAGS`	*(empty)*	The built system will stay on one known machine or CPU family.	Less portable; failures can be harder to compare across hosts.

Profile	`LWI_CFLAGS`	`LWI_CXXFLAGS`	`LBI_CUSTOM_LDFLAGS`	Best fit	Main caution
Debug-heavy build	`-O0 -g3`	`$LWI_CFLAGS`	*(empty)*	You are chasing a toolchain, linker, or package-level failure.	Produces much slower and larger binaries.
Clang ThinLTO	`-O2 -pipe -flto=thin`	`$LWI_CFLAGS`	`-fuse-ld=lld`	You are building with Clang and deliberately optimizing for runtime performance.	Longer builds, higher toolchain sensitivity, and a hard dependency on `lld`.
Size-aware linking	`-O2 -pipe`	`$LWI_CFLAGS`	`-Wl,--as-needed -Wl,-O1`	You want modest link-time cleanup without changing the basic compile profile.	Can expose fragile upstream link ordering.

Practical rules:

- Start with the recommended row unless you can clearly explain why another profile helps.
- Keep `LWI_CXXFLAGS` aligned with `LWI_CFLAGS` unless you are isolating a C++-specific failure.
- If a package breaks under LTO, native tuning, or extra linker flags, retry with plain `-O2 -pipe` before assuming the package itself is broken.
- For Clang-based ThinLTO builds, keep `-flto=thin` in both compile flag variables and pair it with `-fuse-ld=lld` in `LBI_CUSTOM_LDFLAGS`.

3.3 Creating the Directory Layout on the Target Partition

3.3. Creating the Directory Layout on the Target Partition

After the target partition exists, mount it and create the base directory tree deliberately so the rest of the build lands in the paths you intended.

Open standalone page

> **Scope:** this section assumes the reader already created the partition and filesystem, and now needs to lay out directories on that mounted target.

Section 3.1 defined the layout policy, and 3.2 defined variables such as `LBI_ROOT`, `LBI_BOOT`, and `LBI_ESP_MOUNT`. This section applies that policy to the real partition the reader already prepared.

In this stage, the primary layout is created under `/system`, while traditional FHS paths at `/` are exposed as symlinks that point into `/system/*`.

All command examples in this section should be run as the `root` user.

Confirm the Target Mount

Before creating directories, ensure the target partition is mounted where the build expects it. The examples below assume that mount point is `LBI_ROOT`.

```
mkdir -p "$LBI_ROOT"
mount /dev/ROOT_PARTITION "$LBI_ROOT"
```

If the root partition is not mounted first, directory creation commands will run against the host filesystem instead of the new system tree.

Create the Base Layout

Create the custom directory layout under its own `/system` tree on the mounted partition:

```
mkdir -p "$LBI_ROOT/system"

for d in configuration binaries systembinaries libraries headers share documenta
do
  mkdir -p "$LBI_ROOT/system/$d"
done
```

If this installation uses variable runtime storage in its own top-level path, create that now as well:

```
mkdir -p "$LBI_ROOT/system/variable"
```

Create Boot and ESP Paths

Create the boot mount path you chose, then create an ESP mount path only if your installation uses one.

```
mkdir -p "$LBI_ROOT$LBI_BOOT"
mkdir -p "$LBI_ESP_MOUNT"
```

If the reader is reusing an existing ESP that is mounted elsewhere during setup, adjust `LBI_ESP_MOUNT` first and then create only the path that matches that decision.

Add Compatibility Symlinks

The intentional layout remains primary under `/system`, but standard FHS paths should exist as compatibility links:

```
ln -sf system/configuration "$LBI_ROOT/etc"
ln -sf system/binaries "$LBI_ROOT/bin"
ln -sf system/systembinaries "$LBI_ROOT/sbin"
ln -sf system/libraries "$LBI_ROOT/lib"
ln -sf system/users "$LBI_ROOT/home"
ln -sf system/charlie "$LBI_ROOT/root"
ln -sf system/variable "$LBI_ROOT/var"

mkdir -p "$LBI_ROOT/usr"
ln -sf ../system/binaries "$LBI_ROOT/usr/bin"
ln -sf ../system/systembinaries "$LBI_ROOT/usr/sbin"
ln -sf ../system/libraries "$LBI_ROOT/usr/lib"
ln -sf ../system/headers "$LBI_ROOT/usr/include"
ln -sf ../system/share "$LBI_ROOT/usr/share"
ln -sfn libraries "$LBI_ROOT/system/lib"
```

This keeps core FHS paths and common `/usr/*` compatibility paths available from the start. Additional compatibility links can be added later as needed, but creating this core set now reduces surprises during early toolchain stages.

The `/system/lib` link exists for software that looks for the traditional library directory under a prefix. It points at `/system/libraries`, so do not create data directories through that path.

Add Mount Directories for Optional Partitions

If the reader created separate partitions such as `/users`, `/variable`, or another dedicated mount path, create those mount directories at their real locations under `/system` before any additional `mount` calls.

```
mkdir -p "$LBI_ROOT/system/users"
mkdir -p "$LBI_ROOT/system/variable"
```

The rule is straightforward: every partition that will be mounted inside the target tree must have a deliberate directory waiting for it.

Validate Before Continuing

A quick tree listing should show the intended top-level structure on the mounted partition, not on the host root:

```
ls -la "$LBI_ROOT"
```

Once this structure is in place, the rest of the build can assume a stable target layout and avoid accidental path drift later.

Next Steps

It is now recommended to give your user account permission to write into the target tree so the build can run without `sudo` after this point. The exact command depends on your user and group names, but it will look something like this:

```
chown -R $USER:$(id -gn) "$LBI_ROOT"
```

> **Warning:** commands in this section run as `root`. A typo at this privilege level can damage your host, destroy data, or make the system go kaboom. If you skip the ownership handoff and continue the build as `root`, that same risk persists for every later build and install command. Double-check device names, mount points, and `LBI_ROOT` before pressing Enter.

4.1 Sources

4.1. Sources

Source archives should be staged in a dedicated directory using a repeatable POSIX shell fetch workflow.

Open standalone page

> **Goal:** place source archives in `LBI_SOURCES` (or another deliberate directory) before package builds begin.

This section introduces a POSIX `sh` helper that uses `curl` for source archives and `git archive` for pinned source-control snapshots. It is intended for repeatable source staging, not for package management.

All command examples in this section are intended for a normal user shell, not a `root` shell.

Provided Files

The source staging files are distributed with this website and can be downloaded here:

- Download `fetch-sources.sh`
- Download `sources.manifest`
- Download `sources.b2sums`
- Download `linux-by-intent-patches.zip`

The helper accepts URLs directly, or a manifest file containing one source per line. The patch zip contains the book's local patch files; extract it into `LBI_SOURCES` before building packages that list a distributed patch in their input assumptions.

Direct URL Usage

After downloading the script into your current working directory, run it with one or more source URLs:

```
sh ./fetch-sources.sh "https://mirror.example.org/sources/pkg-a-1.0.tar.xz" "htt
```

When a source needs an explicit local filename, place the output name immediately after that URL:

```
sh ./fetch-sources.sh \
    "git+https://git.example.org/pkg#0123456789abcdef" \
    "pkg-0123456789ab.tar.gz"
```

By default, downloads are written to `LBI_SOURCES` when that variable is set; otherwise they are written to `./sources`.

Manifest Usage

A manifest is a plain text file with one entry per line:

```
# URL [OUTPUT_NAME]
https://mirror.example.org/sources/pkg-a-1.0.tar.xz
https://mirror.example.org/sources/pkg-b-2.3.tar.gz pkg-b.tar.gz
git+https://git.example.org/pkg#0123456789abcdef pkg-0123456789ab.tar.gz
```

The provided starter manifest currently includes:

- `musl-1.2.6.tar.gz`

- `llvm-project-22.1.3.src.tar.xz`
- `mimalloc-v3.3.0.tar.gz` (downloaded from the upstream `v3.3.0.tar.gz` tag URL but saved with this explicit output name)
- `linux-7.0.tar.xz`
- `om4-6.7.tar.gz`
- `ncurses-6.6-20260418.tgz`
- `libedit-20251016-3.1.tar.gz`
- `bheaded-0.0.1-mk2.tar.gz`
- `sbase-c1341583c963.tar.gz` (generated from the pinned upstream git commit in `sources.manifest`)
- `BSD-Diffutils-main.zip`
- `file-5.47.tar.gz`
- `zstd-1.5.7.tar.gz`
- `bsdgrep-master.zip`
- `zlib-ng-2.3.3.tar.gz`
- `byacc-20260126.tgz`
- `bfs-4.1.tar.gz`
- `bmake-20260406.tar.gz`
- `awk-20251225.tar.gz`
- `samurai-1.3.tar.gz`
- `make-4.4.1.tar.gz`
- `bsdpatch-v0.99.1.tar.gz`
- `mandoc-1.14.6.tar.gz`
- `bsdsed-v0.99.2.tar.gz`
- `libarchive-3.8.7.tar.xz`
- `xz-5.8.3.tar.xz`
- `gettext-tiny-0.3.3.tar.xz`
- `ubase-e8249b49ca3e.tar.gz` (generated from the pinned upstream git commit in `sources.manifest`)
- `Python-3.14.4.tar.xz`
- `pigz-2.8.tar.gz`
- `dash-0.5.13.3.tar.gz`
- `oksh-7.8.tar.gz`
- `bc-7.0.3.tar.xz`
- `pkgconf-2.5.1.tar.xz`
- `iana-etc-20260409.tar.gz`

- `libressl-4.2.1.tar.gz`
- `cacert-2026-03-19.pem`
- `curl-8.19.0.tar.xz`
- `cmake-4.3.2.tar.gz`
- `flex-2.6.4.tar.gz`
- `sqlite-autoconf-3530000.tar.gz`
- `flit_core-3.12.0.tar.gz`
- `rustc-1.95.0-src.tar.xz`
- `coreutils-0.8.0.tar.gz`
- `red-v1.0.2.tar.gz`
- `dinit-0.21.0.tar.xz`
- `limine-5be26a73d7b7.tar.gz` (generated from the pinned upstream git commit in `sources.manifest`)

Then fetch everything listed in that file:

```
sh ./fetch-sources.sh --manifest ./sources.manifest
```

Checksum File

The provided `sources.b2sums` file records BLAKE2b checksums for the files named in `sources.manifest`, including explicit output names such as `mimalloc-v3.3.0.tar.gz` and the generated git snapshot archives. It also includes the checksum for `linux-by-intent-patches.zip`.

Keeping the manifest and checksum file in sync makes source staging auditable and repeatable as this chapter grows.

Script Behavior

The helper uses `curl --fail --location --retry 3 --continue-at -` so downloads follow redirects, retry on transient failures, and resume partial files when possible.

Entries beginning with `git+` must include a pinned ref after `#`. The helper clones that repository, checks out the ref, and writes a deterministic gzip archive using `git archive` and `gzip -n`.

Existing non-empty files are skipped by default. Use `--force` to re-download existing archives.

Once source archives are staged, later build steps can consume them from `LBI_SOURCES` without mixing fetch policy into each package section.

4.2. Final Checks

Before building packages, verify that the environment is loaded, the target partition is mounted where expected, and the directory and compatibility-link layout actually matches policy.

Open standalone page

> **Gate before package builds:** do not proceed until variables, mount state, directories, and links are all confirmed. A five-minute check here is cheaper than debugging path drift and/or breaking your system in chapter 5.

Section 3.2 defined the build variables, section 3.3 defined the target directory and link layout, and section 4.1 staged source archives. This section is the last consistency check before package build steps begin.

What Should Be True Right Now

At this point, all of the following should already be true:

- The build shell has sourced the appropriate environment file (`sh`, `zsh`, or `fish` variant).
- The target root partition is mounted at `LBI_ROOT`.
- The intended top-level directories under the mounted target tree exist.
- Compatibility links (`/bin`, `/sbin`, `/lib`, `/home`, `/root`, and `/usr/*` links) were created according to section 3.3.

If any item in that list is uncertain, treat it as a blocker and resolve it before continuing.

Variable Verification Checklist

Confirm that the build shell currently exposes the expected values for at least these variables:

- `LBI_ROOT`
- `LBI_SOURCES`
- `LBI_TOOLS`
- `LBI_BOOT`
- `LBI_ESP_MOUNT`
- `LBI_ARCH`
- `LBI_TARGET`

Also confirm that optional tuning variables are either deliberately set or intentionally empty (`LWI_MAKE_FLAGS`, `LWI_CFLAGS`, `LWI_CXXFLAGS`, `LBI_CUSTOM_LDFLAGS`).

The practical rule is simple: if a variable value surprises you now, it will surprise you more during a failed configure or install phase.

Mount Verification Checklist

Confirm that the filesystem mounted at `LBI_ROOT` is the target partition and not the host root. If this is wrong, any “successful” directory or install action is writing into the wrong tree.

Also confirm that any chosen boot/ESP mount paths are present and match your intended boot layout policy from chapter 3.

Directory and Link Verification Checklist

Confirm that the primary directories from section 3.1/3.3 exist inside the mounted target tree, including at least:

- `/system/configuration`
- `/system/binaries`
- `/system/systembinaries`
- `/system/libraries`
- `/system/headers`
- `/system/share`
- `/system/documentation`
- `/system/tools`

Then confirm that compatibility links exist and resolve to the intended targets, including:

- `/bin -> /system/binaries`
- `/sbin -> /system/systembinaries`
- `/lib -> /system/libraries`
- `/home -> /system/users`
- `/root -> /system/charlie`
- `/usr/bin -> ../system/binaries`
- `/usr/sbin -> ../system/systembinaries`
- `/usr/lib -> ../system/libraries`
- `/usr/include -> ../system/headers`
- `/usr/share -> ../system/share`
- `/system/lib -> libraries`

If these links are missing or point somewhere unexpected, correct them before package installation begins.

Readiness Outcome

When variables, mount state, directories, and compatibility links are all verified, chapter 5 can proceed on a stable base.

If one of these checks fails, pause and fix the layout first. Continuing anyway only converts a clear setup issue into a noisier and less enjoyable build failure later.

5.1 Linux API Headers 7.0

5.1. Linux API Headers 7.0

Linux API headers are the exported kernel-to-userspace interface definitions (constants, structs, flags, and syscall-related declarations) that userland software compiles against. We install them early in the target tree so libc, toolchains, and other userspace packages including clang/llvm build against a consistent ABI contract.

Open standalone page

> **Input assumption:** `linux-7.0.tar.xz` is already present from the chapter 4 manifest fetch step.

Earlier, `linux-7.0` was downloaded from `sources.manifest`. This section installs the kernel's exported Linux API headers into the target tree so subsequent userspace builds have the interface definitions they require.

Extract and Enter the Kernel Source Tree

```
cd "$LBI_SOURCES"
tar -xf linux-7.0.tar.xz
cd linux-7.0
```

Install Headers

Run the headers install step with the target install path under `LBI_ROOT/system/headers` so later userspace packages can build against the installed Linux API headers.:

```
make INSTALL_HDR_PATH=/tmp/linux/usr headers_install
mv -v /tmp/linux/usr/include/* "$LBI_ROOT"/system/headers/
```

This places the exported Linux headers under the target tree so later userspace packages can build against them.

5.2 musl libc headers 1.2.6

5.2. musl libc headers 1.2.6

musl libc headers provide the target C library interface that LLVM compiler-rt expects during configuration and compilation, so they must be installed before compiler-rt is built.

Open standalone page

> **Input assumption:** `musl-1.2.6.tar.gz` is already present from the chapter 4 manifest fetch step.

musl is a lightweight C standard library implementation for Linux systems. we need it to provide libc header files required for this book's LLVM `compiler-rt` build.

Extract and Enter the musl Source Tree

```
cd "$LBI_SOURCES"
tar -xf musl-1.2.6.tar.gz
cd musl-1.2.6
```

Configure and Install Headers

Run configure once so musl can prepare its generated header set, then install headers into the target tree:

```
lbi_configure
make install-headers DESTDIR="$LBI_ROOT"
```

This installs musl headers to the target include path (`$LBI_ROOT/usr/include`). In the custom layout used by this book, that path can be linked into the primary `/system` header tree, so later toolchain and runtime components see a consistent libc interface.

5.3 llvm/clang pass 1 22.1.3

5.3. llvm/clang pass 1 22.1.3

The first llvm/clang pass establishes an initial toolchain in the tools prefix so later stages, including compiler-rt, can build against a controlled compiler and linker set.

Open standalone page

> **Performance note:** This stage will take a while. It is **NOT recommended** to run this on a

device with fewer than 8 CPU cores.

> **Input assumption:** `llvm-project-22.1.3.src.tar.xz` is already present from the chapter 4 manifest fetch step.

llvm/clang is a modular compiler and linker toolchain project. we need it to produce the pass 1 compiler components (`clang`, `lld`) used by later runtime work, including `compiler-rt`.

Extract Sources and Create a Build Directory

```
cd "$LBI_SOURCES"
tar -xf llvm-project-22.1.3.src.tar.xz
cd llvm-project-22.1.3.src
mkdir -p build-llvm
cd build-llvm
```

Configure llvm/clang Pass 1

```
cmake -G Ninja "../llvm" \
    -DCMAKE_C_COMPILER=/usr/bin/clang \
    -DCMAKE_CXX_COMPILER=/usr/bin/clang++ \
    -DCMAKE_C_COMPILER_LAUNCHER=ccache \
    -DCMAKE_CXX_COMPILER_LAUNCHER=ccache \
    -DCMAKE_INSTALL_PREFIX=$LBI_ROOT/system/tools \
    -DLLVM_ENABLE_PROJECTS="lld;clang" \
    -DLLVM_ENABLE_RUNTIMES="compiler-rt" \
    -DCOMPILER_RT_BUILD_BUILTINS=ON \
    -DCOMPILER_RT_BUILD_SANITIZERS=OFF \
    -DCOMPILER_RT_BUILD_XRAY=OFF \
    -DCOMPILER_RT_BUILD_LIBFUZZER=OFF \
    -DCOMPILER_RT_BUILD_PROFILE=OFF \
    -DLLVM_DEFAULT_TARGET_TRIPLE="$LBI_TARGET" \
    -DLLVM_TARGETS_TO_BUILD="X86" \
    -DCMAKE_BUILD_TYPE=Release \
    -DCLANG_DEFAULT_CXX_STDLIB=libc++ \
    -DCLANG_DEFAULT_LINKER=lld \
    -DCLANG_DEFAULT_RTLIB=compiler-rt \
    -DDEFAULT_SYSROOT=$LBI_ROOT
```

Build llvm/clang Pass 1

```
ninja $LWI_MAKE_FLAGS
```

Install llvm/clang Pass 1 to the temp tools directory

```
ninja install
```

Command Explanations

- `-G Ninja`: Uses the Ninja generator for fast incremental builds.
- `-DCMAKE_C*_COMPILER=/usr/bin/clang*`: Forces Clang as the C and C++ compilers for pass 1. Remove these if you don't have clang installed, but clang is faster for the 4000+ source files in this pass.
- `-DCMAKE_C_COMPILER_LAUNCHER=ccache`: Enables `ccache` for C compilations to speed rebuilds.
- `-DCMAKE_CXX_COMPILER_LAUNCHER=ccache`: Enables `ccache` for C++ compilations to speed rebuilds.
- `-DCMAKE_INSTALL_PREFIX=$LBI_ROOT/system/tools`: Installs pass 1 outputs into the tools prefix used by later stages.
- `-DLLVM_ENABLE_PROJECTS="lld;clang"`: Builds only the `clang` and `lld` projects from the LLVM monorepo.
- `-DLLVM_ENABLE_RUNTIMES="compiler-rt"`: Enables `compiler-rt` in the runtimes build.
- `-DCOMPILER_RT_BUILD_BUILTINS=ON`: Builds compiler-rt builtins.
- `-DCOMPILER_RT_BUILD_SANITIZERS=OFF`: Skips sanitizer runtime libraries in this pass.
- `-DCOMPILER_RT_BUILD_XRAY=OFF`: Skips XRay runtime components in this pass.
- `-DCOMPILER_RT_BUILD_LIBFUZZER=OFF`: Skips libFuzzer runtime components in this pass.
- `-DCOMPILER_RT_BUILD_PROFILE=OFF`: Skips profile runtime components in this pass.
- `-DLLVM_DEFAULT_TARGET_TRIPLE="$LBI_TARGET"`: Sets the default target triple for generated code.
- `-DLLVM_TARGETS_TO_BUILD="X86"`: Restricts LLVM backend generation to the selected architecture.
- `-DCMAKE_BUILD_TYPE=Release`: Uses release optimizations for a stable bootstrap toolchain.
- `-DCLANG_DEFAULT_CXX_STDLIB=libc++`: Makes Clang default to `libc++` as its C++ standard library.
- `-DCLANG_DEFAULT_LINKER=lld`: Makes Clang default to `lld` as its linker.
- `-DCLANG_DEFAULT_RTLIB=compiler-rt`: Makes Clang default to `compiler-rt` runtime libraries.
- `-DDEFAULT_SYSROOT=$LBI_ROOT`: Sets the default sysroot for clang to the root prefix, so it can find headers and libraries from pass 1 and later stages, so it doesn't use host system headers and

libraries by mistake.

This pass 1 configuration keeps `compiler-rt` limited to builtins-only output.

`ccache` is recommended, but not required. If `ccache` is not installed on your host, remove the two `CMAKE_*_COMPILER_LAUNCHER` lines and re-run configure.

For accepted `LLVM_TARGETS_TO_BUILD` values, choose the target that matches your architecture and verify against upstream documentation:

- https://llvm.org/docs/CMake.html

Create Target-Prefixed llvm Tool Symlinks

Some builds expect cross-style tool names like `$LBI_TARGET-ld`. Create symlinks for the installed LLVM tools so those names resolve in `$LBI_ROOT/system/tools/bin`.

```
cd "$LBI_ROOT/system/tools/bin"

ln -sf clang "$LBI_TARGET-clang"
ln -sf clang++ "$LBI_TARGET-clang++"
ln -sf clang "$LBI_TARGET-cc"
ln -sf clang++ "$LBI_TARGET-c++"

ln -sf llvm-ar "$LBI_TARGET-ar"
ln -sf llvm-ranlib "$LBI_TARGET-ranlib"
ln -sf llvm-as "$LBI_TARGET-as"
ln -sf llvm-nm "$LBI_TARGET-nm"
ln -sf llvm-objcopy "$LBI_TARGET-objcopy"
ln -sf llvm-objdump "$LBI_TARGET-objdump"
ln -sf llvm-readelf "$LBI_TARGET-readelf"
ln -sf llvm-strip "$LBI_TARGET-strip"

ln -sf ld.lld "$LBI_TARGET-ld"
```

5.4 musl libc pass 2 1.2.6

5.4. musl libc pass 2 1.2.6

musl pass 2 builds and installs the full target libc with mimalloc integration, using the controlled toolchain from the previous pass.

Open standalone page

> **Input assumption:** `musl-1.2.6.tar.gz` and `mimalloc-v3.3.0.tar.gz` are already present in `$LBI_SOURCES`, and the three musl-related patches distributed with the website (`musl-1.2.6-mimalloc.patch`, `mimalloc-3.3.0-for-musl.patch`, and `musl-1.2.6-runtime-lib-from-compiler.patch`) have been placed in `$LBI_SOURCES`.

musl is a lightweight C standard library implementation for Linux systems. we need it to provide the full target libc in pass 2, with the allocator integration used by this book.

mimalloc is a general-purpose memory allocator library. we need it to provide the upstream allocator sources embedded into musl for this pass.

Extract musl and Apply the First Patch

```
cd "$LBI_SOURCES"
tar -xf musl-1.2.6.tar.gz
cd musl-1.2.6
patch -Np1 -i ../musl-1.2.6-mimalloc.patch
```

Add mimalloc Upstream Sources (`src` and `include`)

```
mkdir -p src/malloc/mimalloc/upstream
tar -xf ../mimalloc-v3.3.0.tar.gz \
    --strip-components=1 \
    -C src/malloc/mimalloc/upstream \
    mimalloc-3.3.0/src mimalloc-3.3.0/include

# The second patch updates this file in-place.
cp -v src/malloc/mimalloc/upstream/src/static.c src/malloc/mimalloc/static.c
```

Apply the Second Patch

```
patch -Np1 -i ../mimalloc-3.3.0-for-musl.patch
```

Apply the final patch for compiler-rt builtins support

```
patch -Np1 -i ../musl-1.2.6-runtime-lib-from-compiler.patch
```

Configure, Build, and Install musl Pass 2

```
lbi_configure --target="$LBI_TARGET" --with-malloc=mimalloc
make $LWI_MAKE_FLAGS
make install DESTDIR="$LBI_ROOT"

# musl can install this loader link as an absolute path.
# Rewrite it to a relative link so it resolves correctly from the mounted target
ln -snf ./libc.so \
    "$LBI_ROOT/system/libraries/ld-musl-${LBI_ARCH}.so.1"

ls -lh "$LBI_ROOT/usr/lib/ld-musl-${LBI_ARCH}.so.1"
```

5.5. LLVM runtimes (libunwind, libcxxabi, libcxx) 22.1.3

This pass builds and installs the LLVM C++ runtime stack against musl, so later C++ builds have unwinding, ABI support, and the C++ standard library available in the target tree.

Open standalone page

> **Input assumption:** `llvm-project-22.1.3.src.tar.xz` is already present from the chapter 4 manifest fetch step, and section 5.4 completed successfully so musl is available in the target tree.

libunwind is a platform-independent stack unwinding library. we need it to provide the unwinder used by C++ exception handling in this book.

libcxxabi is the LLVM C++ ABI support library. we need it to provide C++ ABI and exception runtime support required by libcxx.

libcxx is the LLVM C++ standard library implementation. we need it to provide the C++ standard library used by Clang in later stages of this book.

Extract Sources and Create a Build Directory

```
cd "$LBI_SOURCES"
tar -xf llvm-project-22.1.3.src.tar.xz
cd llvm-project-22.1.3.src/runtimes
```

Configure LLVM Runtimes for musl

```
lbi_cmake build-runtimes \
    -G Ninja \
    -DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_CXX_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang++" \
    -DCMAKE_SYSROOT="$LBI_ROOT" \
    -DCMAKE_FIND_ROOT_PATH="$LBI_ROOT;$LBI_ROOT/system" \
    -DCMAKE_FIND_ROOT_PATH_MODE_PROGRAM=NEVER \
    -DCMAKE_FIND_ROOT_PATH_MODE_LIBRARY=ONLY \
    -DCMAKE_FIND_ROOT_PATH_MODE_INCLUDE=ONLY \
    -DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=ONLY \
    -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY \
    -DCMAKE_C_FLAGS="--target=$LBI_TARGET" \
    -DCMAKE_CXX_FLAGS="--target=$LBI_TARGET" \
    -DLLVM_ENABLE_RUNTIMES="libunwind;libcxxabi;libcxx" \
    -DLIBUNWIND_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBCXXABI_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBCXX_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLLVM_ENABLE_ZLIB=OFF \
    -DLLVM_ENABLE_ZSTD=OFF \
    -DLLVM_ENABLE_LIBXML2=OFF \
    -DLIBCXX_HAS_MUSL_LIBC=ON \
    -DLIBCXX_HAS_ATOMIC_LIB=OFF \
    -DLIBCXXABI_HAS_CXA_THREAD_ATEXIT_IMPL=OFF \
    -DLIBCXXABI_USE_LLVM_UNWINDER=ON \
    -DLIBCXX_USE_COMPILER_RT=ON \
    -DLIBCXXABI_USE_COMPILER_RT=ON \
    -DLIBUNWIND_USE_COMPILER_RT=ON \
    -DCMAKE_BUILD_TYPE=Release
```

Build LLVM Runtimes

```
ninja -C build-runtimes $LWI_MAKE_FLAGS
```

Install LLVM Runtimes into the Target Tree

```
DESTDIR="$LBI_ROOT" ninja -C build-runtimes install
```

Command Explanations

- `-G Ninja`: Uses the Ninja generator for fast incremental builds.

- `lbi_cmake build-runtimes ...`: Uses the book's CMake helper so install layout follows the environment policy (`/binaries`, `/systembinaries`, `/libraries`, `/headers`, `/configuration`, `/variable`, and documentation paths) instead of defaulting to `/usr`.
- `-DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the pass 1 target-prefixed Clang C compiler.
- `-DCMAKE_CXX_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang++"`: Uses the pass 1 target-prefixed Clang C++ compiler.
- `-DCMAKE_SYSROOT="$LBI_ROOT"`: Directs configure and compile checks at the target sysroot.
- `-DCMAKE_FIND_ROOT_PATH="$LBI_ROOT;$LBI_ROOT/system"`: Restricts CMake's search roots to the target tree.
- `-DCMAKE_FIND_ROOT_PATH_MODE_PROGRAM=NEVER`: Lets build-time tools (like `cmake`, `ninja`, and scripting runtimes) come from the host.
- `-DCMAKE_FIND_ROOT_PATH_MODE_LIBRARY=ONLY`: Prevents linking target libraries from host paths such as `/usr/lib`.
- `-DCMAKE_FIND_ROOT_PATH_MODE_INCLUDE=ONLY`: Prevents header discovery from host include paths.
- `-DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=ONLY`: Forces package discovery into target roots instead of host package locations.
- `-DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY`: Prevents CMake's compiler checks from requiring an executable link against `libc++` before `libc++` has been built.
- `-DCMAKE_C_FLAGS="--target=$LBI_TARGET"`: Ensures C compilation targets the selected triple.
- `-DCMAKE_CXX_FLAGS="--target=$LBI_TARGET"`: Ensures C++ compilation targets the selected triple.
- `-DLLVM_ENABLE_RUNTIMES="libunwind;libcxxabi;libcxx"`: Builds exactly the unwinder, C++ ABI runtime, and C++ standard library runtimes.
- `-DLIBUNWIND_INSTALL_LIBRARY_DIR=/system/libraries`, `-DLIBCXXABI_INSTALL_LIBRARY_DIR=/system/libraries`, `-DLIBCXX_INSTALL_LIBRARY_DIR=/system/libraries`: Forces each runtime project to install its libraries into `/system/libraries`, because these projects can ignore generic `CMAKE_INSTALL_LIBDIR` defaults.
- `-DLLVM_ENABLE_ZLIB=OFF`, `-DLLVM_ENABLE_ZSTD=OFF`, `-DLLVM_ENABLE_LIBXML2=OFF`: Disables optional compression/XML dependencies in this pass so host `find_package` hits do not leak into the runtime bootstrap.
- `-DLIBCXX_HAS_MUSL_LIBC=ON`: Enables libcxx musl-specific behavior for the target libc environment.
- `-DLIBCXX_HAS_ATOMIC_LIB=OFF`: Prevents libcxx from linking `-latomic` as an external library in this pass.
- `-DLIBCXXABI_HAS_CXA_THREAD_ATEXIT_IMPL=OFF`: Prevents libcxxabi from requiring glibc's `__cxa_thread_atexit_impl` symbol, which is not provided by musl.

- `-DLIBCXXABI_USE_LLVM_UNWINDER=ON`: Configures libcxxabi to use libunwind from this runtime set.
- `-DLIBCXX_USE_COMPILER_RT=ON`: Uses compiler-rt runtime libraries when building libcxx.
- `-DLIBCXXABI_USE_COMPILER_RT=ON`: Uses compiler-rt runtime libraries when building libcxxabi.
- `-DLIBUNWIND_USE_COMPILER_RT=ON`: Uses compiler-rt runtime libraries when building libunwind.
- `-DCMAKE_BUILD_TYPE=Release`: Uses release optimizations for runtime libraries.
- `DESTDIR="$LBI_ROOT" ninja -C build-runtimes install`: Installs into the target tree without writing to the host filesystem.

For additional runtime configuration details and accepted CMake options, see upstream documentation:

- https://llvm.org/docs/CMake.html

6.1 Introduction

6.1. Introduction

Chapter 6 begins the cross-compilation phase that produces a minimal working target system used as the base for chapters 7 and 8.

Open standalone page

> **Goal:** move from toolchain staging to cross-compiling a minimal, runnable target userspace that later chapters can extend without rebuilding the foundation every time something sneezes.

Chapter 5 established the compiler and runtime pieces needed to stop leaning on the host toolchain. Chapter 6 is where those pieces are used in earnest to assemble a minimal working system in the target tree.

This chapter is intentionally about **minimum viable system behavior**, not feature completeness. The objective is a small, coherent base that can boot into useful maintenance and build workflows, then serve as the platform for chapters 7 and 8.

What This Chapter Is Trying to Achieve

By the end of chapter 6, the target tree should contain enough core userland to support predictable operation and follow-on package work, including:

- essential runtime libraries and foundational tools in the intended layout;
- consistent target-side linkage against the cross-built stack from chapter 5;

- a stable baseline that later package groups can assume without restaging bootstrap pieces.

What This Chapter Is Not Trying to Do

Chapter 6 is not the place to chase polish, optional tooling, or every package someone on the internet once called “tiny but mandatory.” If a component is not required to establish the minimal working base for chapter 7 and chapter 8, it can wait.

> In short: this chapter builds the floor, not the chandelier.

Why the Separation Matters for Chapters 7 and 8

Keeping chapter 6 focused on minimal system viability makes later chapters cleaner:

- chapter 7 will get us into a working shell with its own userland and LLVM toolchain, so we can build the full system from there.;
- chapter 8 will be the actual system building.

That separation reduces troubleshooting noise and makes regressions easier to localize when something inevitably gets creative at build time.

6.2 om4 6.7

6.2. om4 6.7

om4 provides a compact m4 macro processor in the target userspace, using its custom configure flow with only supported options.

Open standalone page

> **Input assumption:** `om4-6.7.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-3-Clause
- ISC

Dependencies:

- musl (libc)

om4 is an OpenBSD-derived implementation of the m4 macro processor. we need it to provide a small, predictable `m4` binary for package build flows in later chapters.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf om4-6.7.tar.gz
cd om4-6.7
```

Configure om4 (Supported Flags Only)

This package uses a custom `configure` script and does **not** support the full flag set used by `lbi_configure`. It also insists on running a compiler test binary during configuration, so use a temporary static-link compiler command for `configure`, then switch back to the normal cross compiler for `make`.

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang --target=$LBI_TARGET --sysroot=
CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
./configure \
    --prefix=/system \
    --bindir=/system/binaries \
    --mandir=/system/documentation/man-pages \
    --enable-m4
```

Apply the Parser Compatibility Fix

Upstream `parser.y` calls `exit(1)` but does not include `<stdlib.h>`. Add it before building so modern Clang modes do not fail with an undeclared-function error.

```
grep -q '^#include <stdlib.h>$' parser.y || \
    sed -i '/^#include <stdint.h>$/a #include <stdlib.h>' parser.y
```

Build and Install om4

```
make -j1 CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"
make CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" install DESTDIR="$LBI_ROO
```

Command Explanations

- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang --target=$LBI_TARGET --sysroot=$LBI_ROOT -static"`: Uses the pass-1 target-prefixed Clang compiler and forces static link for `configure` probe binaries, so they can execute on a glibc host while still probing the target musl toolchain.

- `CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the configured triple and sysroot while allowing optional local C flag tuning.
- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"`: Keeps link behavior on the target triple/sysroot and preserves optional local linker flags.
- `--prefix=/system`: Sets package install prefix under the target `/system` tree.
- `--bindir=/system/binaries`: Installs the executable into the book's binary path layout.
- `--mandir=/system/documentation/man-pages`: Installs manual pages under the documentation tree.
- `--enable-m4`: Installs the program name as `m4` instead of `om4`.
- `grep -q ... || sed -i ... parser.y`: Adds the missing `<stdlib.h>` include only if needed, keeping the source tree compatible with modern Clang behavior.
- `make -j1 CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Reverts to the normal cross compiler command and serializes the build to avoid `yacc` multi-target race conditions in this package's generated `Makefile`.

The upstream script also accepts `--enable-static`, but that is not required for this pass.

6.3 ncurses 6.6-20260418

6.3. ncurses 6.6-20260418

ncurses installs terminal handling libraries and terminfo tooling, with a host-built `tic` used during target installation.

Open standalone page

> **Input assumption:** `ncurses-6.6-20260418.tgz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- X11-style (ncurses)

Dependencies:

- musl (libc)
- libcxx (c++ library, for the C++ bindings)

ncurses is a terminal handling library and terminfo toolkit. we need it to provide curses interfaces and terminal capability data for later userspace packages in this book.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf ncurses-6.6-20260418.tgz
cd ncurses-6.6-20260418
```

Build Host `tic`

```
mkdir -pv build
pushd build
  ../configure --prefix="$LBI_ROOT/system/tools" AWK=gawk
  make -C include
  make -C progs tic
  install -vm755 progs/tic "$LBI_ROOT/system/tools/bin/tic"
popd
```

Configure, Build, and Install ncurses

Use `lbi_configure` for the target configure phase instead of a manual full-path `./configure` invocation.

```
lbi_configure \
    --host="$LBI_TARGET" \
    --build="$(./config.guess)" \
    --with-manpage-format=normal \
    --with-shared \
    --without-normal \
    --with-cxx-shared \
    --without-debug \
    --without-ada \
    --disable-stripping \
    AWK=gawk

make $LWI_MAKE_FLAGS
make DESTDIR="$LBI_ROOT" TIC_PATH="$PWD/build/progs/tic" install
```

Post-install Compatibility Adjustments

```
ln -sv libncursesw.so "$LBI_ROOT/system/libraries/libncurses.so"
sed -e 's/^#if.*XOPEN.*$/#if 1/' \
    -i "$LBI_ROOT/system/headers/curses.h"
```

Command Explanations

- `mkdir -pv build`: Creates a separate host-build directory for the `tic` program without mixing host objects into the target build tree.
- `../configure --prefix="$LBI_ROOT/system/tools" AWK=gawk`: Configures the temporary host-side ncurses tools under the tools prefix and selects the host `gawk` for generated source steps.
- `make -C include`: Builds the generated ncurses headers needed before compiling `tic`.
- `make -C progs tic`: Builds only the host-runnable terminfo compiler instead of building the full host ncurses package.
- `install -vm755 progs/tic "$LBI_ROOT/system/tools/bin/tic"`: Places the host `tic` where the later target install step can reference it explicitly.
- `lbi_configure`: Uses the book's configure helper so ncurses installs into `/system` layout paths instead of upstream defaults.
- `--host="$LBI_TARGET"` and `--build="$(./config.guess)"`: Tell configure this is a cross build from the current host into the target triple.
- `--with-manpage-format=normal`: Installs normal man pages instead of compressed or alternate-format pages.
- `--with-shared` and `--without-normal`: Builds shared ncurses libraries and skips static normal libraries for this pass.
- `--with-cxx-shared`: Builds the C++ ncurses binding as a shared library.
- `--without-debug` and `--without-ada`: Skips debug libraries and Ada bindings that are not needed in the minimal target.
- `--disable-stripping`: Keeps binaries unstripped so symbol removal can remain a deliberate later policy choice.
- `AWK=gawk`: Uses GNU awk for ncurses' generated source rules.
- `make DESTDIR="$LBI_ROOT" TIC_PATH="$PWD/build/progs/tic" install`: Installs into the target tree while forcing the install rule to use the host-built `tic`.
- `ln -sv libncursesw.so "$LBI_ROOT/system/libraries/libncurses.so"`: Provides the conventional `libncurses.so` name as an alias for the wide-character ncurses library.
- `sed -e 's/^#if.*XOPEN.*$/#if 1/' ... curses.h`: Makes the installed header expose X/Open declarations expected by later packages.

6.4 libedit 20251016-3.1

6.4. libedit 20251016-3.1

libedit provides BSD editline functionality used by programs that need interactive line editing and history behavior.

Open standalone page

> **Input assumption:** `libedit-20251016-3.1.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-3-Clause

Dependencies:

- ncurses
- musl (libc)

libedit is a BSD editline and history library. we need it to provide line-editing functionality for interactive target-side programs in later chapters.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf libedit-20251016-3.1.tar.gz
cd libedit-20251016-3.1
```

Configure libedit

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
CPPFLAGS="-D__STDC_ISO_10646__=201706L" \
lbi_configure \
    --host="$LBI_TARGET"
```

Build libedit

```
make $LWI_MAKE_FLAGS
```

Install libedit

```
make install DESTDIR="$LBI_ROOT"
```

Command Explanations

- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang from the temporary toolchain instead of the host compiler.
- `CPPFLAGS="-D__STDC_ISO_10646__=201706L"`: Defines the Unicode character-set macro that libedit checks for wide-character support.

- `lbi_configure --host="$LBI_TARGET"`: Runs the book's configure helper for the `/system` layout while marking this as a cross build for the target triple.
- `make $LWI_MAKE_FLAGS`: Builds libedit using the shared parallel make policy from the build environment.
- `make install DESTDIR="$LBI_ROOT"`: Installs into the target root instead of writing into the host filesystem.

6.6. bheaded 0.0.1-mk2

bheaded provides BSD compatibility headers needed by BSD-origin userland code when building on a musl-based Linux target.

Open standalone page

> **Input assumption:** `bheaded-0.0.1-mk2.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- The Berkeley Artistic License

Dependencies:

- musl (libc)

bheaded is a BSD compatibility header package for non-BSD systems. we need it to provide BSD-style `sys` headers (`cdefs.h`, `queue.h`, `tree.h`) for BSD-origin userland code in this chapter.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf bheaded-0.0.1-mk2.tar.gz
cd bheaded-0.0.1-mk2
```

Fetch Header Sources

```
make $LWI_MAKE_FLAGS main
```

Install bheaded

```
make DESTDIR="$LBI_ROOT" install
```

Command Explanations

- `make $LWI_MAKE_FLAGS main`: Fetches and prepares the BSD compatibility header set using the book's shared make parallelism setting.
- `make DESTDIR="$LBI_ROOT" install`: Installs those headers into the target tree rather than the host system.

6.7 sbase git snapshot

6.7. sbase git snapshot

sbase provides a compact portable Unix utility set for the early target userspace.

Open standalone page

> **Input assumption:** `sbase-c1341583c963.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.
>
> **Source note:** this archive is generated by `scripts/fetch-sources.sh` from upstream `https://git.suckless.org/sbase` at commit `c1341583c96307cb0e6152c963ed23c4d56a4278`.

Licenses:

- MIT/X Consortium-style license

Dependencies:

- musl (libc)
- make (host)
- awk (host)
- yacc-compatible parser generator, such as `yacc` or `byacc` (host)

sbase is a compact portable Unix utility collection from suckless. we need it to provide the early target command set used by later build, install, and troubleshooting steps.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf sbase-c1341583c963.tar.gz
```

```
cd sbase-c1341583c963
```

Build sbase

> **Build note:** sbase builds one generated `bc` source with a yacc-compatible parser generator. Prefer `yacc` when present, and fall back to `byacc`.

```
YACC=${YACC:-yacc}
if ! command -v "$YACC" >/dev/null 2>&1 && command -v byacc >/dev/null 2>&1; the
    YACC=byacc
fi

make $LWI_MAKE_FLAGS \
    YACC="$YACC" \
    CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    AR="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ar" \
    RANLIB="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ranlib" \
    CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"
```

Install sbase

```
make install \
    DESTDIR="$LBI_ROOT" \
    PREFIX=/system \
    MANPREFIX=/system/documentation/man-pages

install -d "$LBI_ROOT/system/binaries"

if [ -d "$LBI_ROOT/system/bin" ]; then
    mv -f "$LBI_ROOT/system/bin/"* "$LBI_ROOT/system/binaries/"
    rmdir "$LBI_ROOT/system/bin"
fi

ln -sf xinstall "$LBI_ROOT/system/binaries/install"
```

Command Explanations

- `YACC=${YACC:-yacc}`: Uses an existing `YACC` setting when present, otherwise defaults to `yacc`.

- `command -v "$YACC" ... && command -v byacc ...`: Falls back to `byacc` when the default yacc-compatible parser generator is not available.
- `make $LWI_MAKE_FLAGS ...`: Builds sbase using the shared make parallelism setting.
- `YACC="$YACC"`: Passes the selected parser generator into sbase's generated-source rules.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Builds target programs with the target-prefixed Clang compiler.
- `AR` and `RANLIB`: Use the matching target-prefixed LLVM archive tools for any static archives built by sbase.
- `CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the selected triple, reads headers and libraries from the target root, and preserves local C flag tuning.
- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"`: Keeps link steps pointed at the target sysroot while preserving local linker flag tuning.
- `make install DESTDIR="$LBI_ROOT" PREFIX=/system MANPREFIX=/system/documentation/man-pages`: Installs under the target `/system` layout and places manual pages in the book's documentation tree.
- `install -d "$LBI_ROOT/system/binaries"`: Ensures the final binary directory exists before moving upstream's default output.
- `mv -f "$LBI_ROOT/system/bin/"* "$LBI_ROOT/system/binaries/"`: Moves programs from upstream's default `bin` directory into the book's `binaries` directory.
- `rmdir "$LBI_ROOT/system/bin"`: Removes the now-empty upstream default directory so the target tree keeps the intended layout.
- `ln -sf xinstall "$LBI_ROOT/system/binaries/install"`: Provides the expected `install` command name from sbase's `xinstall` binary.

6.8. BSD-Diffutils main snapshot

BSD-Diffutils provides BSD-style file and directory difference tools, built here from the upstream main branch snapshot.

Open standalone page

> **Input assumption:** `BSD-Diffutils-main.zip` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

> **Snapshot note:** this package is built from `main.zip`, so upstream source content may change over time.

Licenses:

- BSD-3-Clause

Dependencies:

- musl (libc)
- bmake (for building)
- bheaded (for BSD compatibility headers)

BSD-Diffutils is a BSD-style diff and comparison utility set. we need it to provide file and directory comparison tools for later validation and maintenance steps in the book.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
unzip -q BSD-Diffutils-main.zip
cd BSD-Diffutils-main
```

Apply Compatibility and Layout Patches

```
sed -i \
    -e '1s|^#!/bin/ksh -$|#!/bin/sh|' \
    -e 's|^diff3prog=/usr/libexec/diff3prog$|diff3prog=/systembinaries/diff3prog
    -e 's|^export PATH=.*$|export PATH=/binaries:/systembinaries:/bin:/usr/bin:/
    src/diff3/diff3.ksh

sed -i \
    's|${DESTDIR}/usr/bin/diff3|${DESTDIR}/system/binaries/diff3|' \
    src/diff3/Makefile

sed -i \
    's|char\\s*\\*splice(char \\*, char \\*);|char\t*diff_splice(char *, char *)
    src/diff/diff.h

sed -i \
    's/\bsplice(/diff_splice(/g' \
    src/diff/diff.c src/diff/diffreg.c

sed -i \
    's/u_char ch, \\*p1, \\*p2;/unsigned char ch, *p1, *p2;/' \
    src/cmp/regular.c
```

Build BSD-Diffutils

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
CPPFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -include ../../include/sys/cd
CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
bmake $LWI_MAKE_FLAGS
```

Install BSD-Diffutils

```
install -d "$LBI_ROOT/system/binaries"

for d in cmp diff sdiff; do
    CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    CPPFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -include ../../include/sy
    CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
    bmake -C "src/$d" \
        DESTDIR="$LBI_ROOT" \
        BINDIR=/system/binaries \
        MANDIR=/system/documentation/man-pages \
        STRIP_FLAG= \
        BINOWN="$(id -un)" BINGRP="$(id -gn)" \
        MANOWN="$(id -un)" MANGRP="$(id -gn)" \
        install
done

CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
CPPFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -include ../../include/sys/cd
CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
bmake -C src/diff3 \
    DESTDIR="$LBI_ROOT" \
    BINDIR=/system/systembinaries \
    MANDIR=/system/documentation/man-pages \
    STRIP_FLAG= \
    BINOWN="$(id -un)" BINGRP="$(id -gn)" \
    MANOWN="$(id -un)" MANGRP="$(id -gn)" \
    install
```

Command Explanations

- `unzip -q BSD-Diffutils-main.zip`: Extracts the upstream snapshot archive quietly.
- `sed -i ... src/diff3/diff3.ksh`: Rewrites the helper script to use `/bin/sh`, the book's `diff3prog` location, and the target command search path.
- `sed -i ... src/diff3/Makefile`: Changes the `diff3` install path from the upstream `/usr/bin` default to `/system/binaries`.
- `sed -i ... src/diff/diff.h` and `sed -i ... src/diff/diff.c src/diff/diffreg.c`: Rename the local `splice` helper so it does not collide with Linux's `splice(2)` declaration.
- `sed -i ... src/cmp/regular.c`: Replaces the BSD `u_char` typedef with standard `unsigned char` for portability on musl.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Builds the target utilities with the target-prefixed Clang compiler.
- `CPPFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -include ../../include/sys/cdefs.h"`: Targets the sysroot and force-includes the BSD compatibility `cdefs.h` header needed by this codebase.
- `CFLAGS` and `LDFLAGS`: Keep compile and link steps on the target triple/sysroot while preserving local tuning variables.
- `bmake $LWI_MAKE_FLAGS`: Uses BSD make because this project uses BSD-style makefiles, while still honoring the shared make parallelism setting.
- `install -d "$LBI_ROOT/system/binaries"`: Ensures the destination directory exists before package install rules run.
- `for d in cmp diff sdiff; do ... done`: Installs the normal user-facing comparison tools with the same cross-build settings.
- `bmake -C "src/$d" ... install`: Runs each utility's install rule from its own source directory.
- `BINDIR=/system/binaries` and `MANDIR=/system/documentation/man-pages`: Override BSD make install destinations to match the book's layout.
- `STRIP_FLAG=`: Prevents the install rule from stripping binaries automatically.
- `BINOWN`, `BINGRP`, `MANOWN`, and `MANGRP`: Use the current user and group so the non-root install into `LBI_ROOT` succeeds.
- `bmake -C src/diff3 ... BINDIR=/system/systembinaries`: Installs `diff3prog` support files into the system-binary directory where the patched script expects them.

6.9. file 5.47

file identifies file types by checking filesystem metadata and magic signatures.

Open standalone page

> **Input assumption:** `file-5.47.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-2-Clause

Dependencies:

- musl (libc)

file is a file type identification utility and library. we need it to detect file formats and binaries reliably in later build, packaging, and troubleshooting steps.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf file-5.47.tar.gz
cd file-5.47
```

Configure file

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
lbi_configure \
    --host="$LBI_TARGET"
```

Build file

```
make $LWI_MAKE_FLAGS
```

Install file

```
make install DESTDIR="$LBI_ROOT"
```

Command Explanations

- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler for the cross build.
- `lbi_configure --host="$LBI_TARGET"`: Applies the book's `/system` install layout and tells configure to build for the target triple.
- `make $LWI_MAKE_FLAGS`: Builds with the shared make parallelism policy from the environment.

- `make install DESTDIR="$LBI_ROOT"`: Stages the install into the target root instead of touching the host system.

6.10. bsdgrep master snapshot

bsdgrep provides a FreeBSD-derived grep implementation for Linux targets.

Open standalone page

> **Input assumption:** `bsdgrep-master.zip` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

> **Snapshot note:** this package is built from `master`, so upstream source content may change over time.

Licenses:

- BSD-2-Clause-FreeBSD

Dependencies:

- musl (libc)
- make

bsdgrep is a FreeBSD-derived grep implementation for Linux. we need it to provide `grep`, `egrep`, `fgrep`, and `rgrep` as BSD-style replacements in the target userspace.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
unzip -q bsdgrep-master.zip
cd bsdgrep-master
```

Apply Install Path Patch

```
sed -i \
    -e 's|${DESTDIR}${PREFIX}/bin|${DESTDIR}/system/binaries|g' \
    -e 's|${DESTDIR}${PREFIX}/share/man/man1|${DESTDIR}/system/documentation/man
    Makefile
```

Build bsdgrep

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -DREG_STARTEND=0 $LWI_CFLAGS" \
LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
make $LWI_MAKE_FLAGS
```

Install bsdgrep

```
DESTDIR="$LBI_ROOT" make install
```

Command Explanations

- `unzip -q bsdgrep-master.zip`: Extracts the upstream snapshot archive quietly.
- `sed -i ... Makefile`: Rewrites upstream install destinations from default prefix-relative paths to the book's binary and man-page directories.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Builds bsdgrep with the target-prefixed Clang compiler.
- `CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -DREG_STARTEND=0 $LWI_CFLAGS"`: Targets the selected sysroot and disables `REG_STARTEND` use for compatibility with the target regex implementation.
- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"`: Links against the target sysroot while preserving local linker tuning.
- `make $LWI_MAKE_FLAGS`: Builds with the shared make parallelism setting.
- `DESTDIR="$LBI_ROOT" make install`: Installs into the target root using the patched Makefile paths.

6.11 zlib-ng 2.3.3

6.11. zlib-ng 2.3.3

zlib-ng provides zlib-compatible compression and decompression libraries for the target system.

Open standalone page

> **Input assumption:** `zlib-ng-2.3.3.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- zlib License

Dependencies:

- musl (libc)
- cmake
- ninja

zlib-ng is a deflate compression library with a zlib-compatible API mode. we need it to provide `libz` for packages in later chapters that depend on zlib-compatible compression support.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf zlib-ng-2.3.3.tar.gz
cd zlib-ng-2.3.3
```

Configure zlib-ng

```
lbi_cmake build \
    -DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_AR="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ar" \
    -DCMAKE_RANLIB="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ranlib" \
    -DCMAKE_C_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    -DCMAKE_SHARED_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_C
    -DCMAKE_EXE_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUST
    -DZLIB_COMPAT=ON \
    -DBUILD_TESTING=OFF \
    -DINSTALL_UTILS=OFF
```

Build zlib-ng

```
cmake --build build $LWI_MAKE_FLAGS
```

Install zlib-ng

```
DESTDIR="$LBI_ROOT" cmake --install build
```

Command Explanations

- `lbi_cmake build`: Configures zlib-ng with the book's CMake helper and writes generated files into the `build` directory.

- `-DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler.
- `-DCMAKE_AR` and `-DCMAKE_RANLIB`: Use the matching target-prefixed archive tools.
- `-DCMAKE_C_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the selected triple and sysroot while preserving local C flag tuning.
- `-DCMAKE_SHARED_LINKER_FLAGS` and `-DCMAKE_EXE_LINKER_FLAGS`: Keep shared-library and executable link steps pointed at the target sysroot.
- `-DZLIB_COMPAT=ON`: Builds zlib-ng in zlib-compatible API and library-name mode.
- `-DBUILD_TESTING=OFF`: Skips test programs that are not needed for the target bootstrap.
- `-DINSTALL_UTILS=OFF`: Installs the library and headers without zlib-ng's optional command-line utilities.
- `cmake --build build $LWI_MAKE_FLAGS`: Builds the configured tree using the shared parallel build setting.
- `DESTDIR="$LBI_ROOT" cmake --install build`: Installs into the target root instead of the host filesystem.

6.12 pigz 2.8

6.12. pigz 2.8

pigz is a parallel gzip-compatible compressor and decompressor.

Open standalone page

> **Input assumption:** `pigz-2.8.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- zlib License

Dependencies:

- musl (libc)
- make
- zlib-ng (libz)
- pthreads

pigz is a parallel gzip-compatible compressor and decompressor. we need it to replace `gzip` functionality in the target userspace with multi-core capable compression and decompression.

Replacement Symlinks

To replace gzip-family commands, this package should provide these command names in `PATH`:

- `gzip` -> `pigz`
- `gunzip` -> `unpigz`
- `zcat` -> `unpigz`

Optional additional compatibility names can be added later as needed.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf pigz-2.8.tar.gz
cd pigz-2.8
```

Build pigz

```
make $LWI_MAKE_FLAGS \
    CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    CFLAGS="-O3 -Wall -Wextra -Wno-unknown-pragmas -Wcast-qual --target=$LBI_TAR
    LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -L$LBI_ROOT/system/librari
```

Install pigz

```
install -Dm755 pigz "$LBI_ROOT/system/binaries/pigz"
install -Dm755 unpigz "$LBI_ROOT/system/binaries/unpigz"
install -Dm644 pigz.1 "$LBI_ROOT/system/documentation/man-pages/man1/pigz.1"
```

Install gzip Compatibility Symlinks

```
ln -sf pigz "$LBI_ROOT/system/binaries/gzip"
ln -sf unpigz "$LBI_ROOT/system/binaries/gunzip"
ln -sf unpigz "$LBI_ROOT/system/binaries/zcat"
```

Command Explanations

- `make $LWI_MAKE_FLAGS`: Builds pigz with the shared make parallelism setting.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler.
- `CFLAGS="-O3 ... --target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Keeps upstream's optimization and warning flags while targeting the selected sysroot and preserving local

C flag tuning.

- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -L$LBI_ROOT/system/libraries $LBI_CUSTOM_LDFLAGS"`: Links against target libraries, including the zlib-compatible library installed under `/system/libraries`.
- `install -Dm755 pigz ...`: Installs the compressor binary and creates parent directories as needed.
- `install -Dm755 unpigz ...`: Installs the decompressor binary alongside `pigz`.
- `install -Dm644 pigz.1 ...`: Installs the manual page into the book's man-page tree.
- `ln -sf pigz "$LBI_ROOT/system/binaries/gzip"`: Provides the standard `gzip` command name through pigz.
- `ln -sf unpigz ... gunzip` and `ln -sf unpigz ... zcat`: Provides gzip-family decompression command names through `unpigz`.

6.13 dash 0.5.13.3

6.13. dash 0.5.13.3

dash provides a small POSIX shell for script execution in the target userspace.

Open standalone page

> **Input assumption:** `dash-0.5.13.3.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.
>
> **Source URL:** `http://gondor.apana.org.au/~herbert/dash/files/dash-0.5.13.3.tar.gz`
>
> **Upstream build note:** the release tarball includes a generated `configure` script and `Makefile.in` files, so no autotools bootstrap is needed.

Licenses:

- BSD-3-Clause
- GPL-2.0-or-later for the `mksignames.c` generator input noted by upstream

Dependencies:

- musl (libc)

- make

dash is a small POSIX-compliant Bourne shell implementation. we need it to provide a fast `sh` for scripts and interactive recovery tasks in the minimal target userspace.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf dash-0.5.13.3.tar.gz
cd dash-0.5.13.3
```

Configure dash

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
lbi_configure \
    --host="$LBI_TARGET"
```

Build dash

```
make $LWI_MAKE_FLAGS
```

Install dash

```
make install DESTDIR="$LBI_ROOT"
```

Post-install Shell Symlink

```
ln -sf dash "$LBI_ROOT/system/binaries/sh"
```

Command Explanations

- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler for the shell binary.
- `CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the selected triple and sysroot while preserving local C flag tuning.
- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"`: Keeps link steps in the target sysroot and preserves local linker tuning.
- `lbi_configure --host="$LBI_TARGET"`: Uses the book's configure helper for `/system` paths and marks the build as a cross build.

- `make $LWI_MAKE_FLAGS`: Builds dash with the shared make parallelism setting.
- `make install DESTDIR="$LBI_ROOT"`: Installs into the target root rather than the host filesystem.
- `ln -sf dash "$LBI_ROOT/system/binaries/sh"`: Provides the required `sh` command name through the dash binary.

6.14 oksh 7.8

6.14. oksh 7.8

oksh provides the interactive login shell used when entering the target chroot.

Open standalone page

> **Input assumption:** `oksh-7.8.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/ibara/oksh/releases/download/oksh-7.8/oksh-7.8.tar.gz`
>
> **Upstream build note:** upstream ships a custom `configure` script and documents `./configure`, `make`, and `make install`. for cross builds, upstream documents `--no-thanks` to skip host execution checks.

Licenses:

- Public domain (main shell code)
- BSD and ISC-style licenses (portability code)

Dependencies:

- musl (libc)
- clang/llvm toolchain
- make

oksh is a portable OpenBSD `ksh` implementation. we need it to provide the interactive login shell used for chapter 7 chroot work.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf oksh-7.8.tar.gz
```

```
cd oksh-7.8
```

Configure oksh

> `oksh` does not use GNU autotools `configure` defaults, so `lbi_configure` is not supported here. this section calls upstream's script directly and uses `--no-thanks` for cross-compiling.

```
./configure \
    --no-thanks \
    --disable-curses \
    --prefix=/system \
    --bindir=/system/binaries \
    --mandir=/system/documentation/man-pages \
    --cc="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    --cflags="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"
```

Build oksh

```
make $LWI_MAKE_FLAGS \
    LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"
```

Install oksh

```
make install DESTDIR="$LBI_ROOT"
ln -sf oksh "$LBI_ROOT/system/binaries/ksh"
```

Command Explanations

- `./configure`: Uses oksh's upstream configure script directly because it is not a GNU autotools-compatible script.
- `--no-thanks`: Skips configure checks that try to run target-built test programs during the cross build.
- `--disable-curses`: Builds oksh without curses support for this early target shell.
- `--prefix=/system`, `--bindir=/system/binaries`, and `--mandir=/system/documentation/man-pages`: Install oksh into the book's filesystem layout.
- `--cc="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler.
- `--cflags="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the selected triple/sysroot while preserving local C flag tuning.

- `make $LWI_MAKE_FLAGS LDFLAGS=...`: Builds with shared make parallelism and target-sysroot linker flags.
- `make install DESTDIR="$LBI_ROOT"`: Installs into the target root instead of the host system.
- `ln -sf oksh "$LBI_ROOT/system/binaries/ksh"`: Provides the traditional `ksh` command name for the installed oksh binary.

6.15. bfs 4.1

bfs provides a breadth-first, find-compatible file traversal utility for the target userspace.

Open standalone page

> **Input assumption:** `bfs-4.1.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

> **Source archive note:** this release archive is flat (no top-level directory), so extract it into a dedicated `bfs-4.1/` directory.

Licenses:

- oBSD

Dependencies:

- musl (libc)
- make

bfs is a breadth-first `find`-compatible file search utility. we need it to replace `find` in the target userspace with a modern and predictable traversal tool.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
mkdir -pv bfs-4.1
tar -xf bfs-4.1.tar.gz -C bfs-4.1
cd bfs-4.1
```

Configure bfs

> **Configure note:** `bfs` ships a custom `configure` wrapper and does not follow the standard autotools `configure` interface used by `lbi_configure`, so this section uses its native `./configure` command directly.

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
./configure \
    --enable-release \
    --without-libacl \
    --without-libcap \
    --without-libselinux \
    --without-liburing \
    --without-oniguruma
```

Build bfs

```
make $LWI_MAKE_FLAGS
```

Install bfs

```
install -Dm755 bin/bfs "$LBI_ROOT/system/binaries/bfs"
install -Dm644 docs/bfs.1 "$LBI_ROOT/system/documentation/man-pages/man1/bfs.1"
```

Replace `find` with bfs

```
ln -sf bfs "$LBI_ROOT/system/binaries/find"
```

Command Explanations

- `mkdir -pv bfs-4.1`: Creates a dedicated source directory because this release archive does not contain a top-level directory.
- `tar -xf bfs-4.1.tar.gz -C bfs-4.1`: Extracts the flat archive into that dedicated directory.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler.
- `CFLAGS` and `LDFLAGS`: Target the selected triple/sysroot and preserve local compile and link tuning.
- `./configure`: Uses bfs' native configure wrapper because it does not support the full `lbi_configure` option set.

- `--enable-release`: Selects the release build profile.
- `--without-libacl`, `--without-libcap`, `--without-libselinux`, `--without-liburing`, and `--without-oniguruma`: Disable optional dependencies that are not present in the early target userspace.
- `make $LWI_MAKE_FLAGS`: Builds bfs with the shared make parallelism setting.
- `install -Dm755 bin/bfs ...`: Installs the bfs executable and creates the destination directory if needed.
- `install -Dm644 docs/bfs.1 ...`: Installs the manual page into the book's man-page tree.
- `ln -sf bfs "$LBI_ROOT/system/binaries/find"`: Provides the conventional `find` command name through bfs.

6.16 awk 20251225

6.16. awk 20251225

awk provides the One True Awk implementation for POSIX text-processing scripts in the target userspace.

Open standalone page

> **Input assumption:** `awk-20251225.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- Lucent Technologies permissive license

Dependencies:

- musl (libc)
- make
- bison (or another yacc-compatible parser generator)
- host C compiler (for the `maketab` build helper)

awk is a POSIX awk language interpreter from the One True Awk project. we need it to provide `awk` for text processing and script compatibility in later chapters.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf awk-20251225.tar.gz
```

```
cd awk-20251225
```

Build awk

> **Configure note:** this package does not ship a `configure` script, so `lbi_configure` is not supported here. Build settings are passed directly to `make`.

```
make $LWI_MAKE_FLAGS \
    HOSTCC="cc -g -Wall -pedantic -Wcast-qual" \
    CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang --target=$LBI_TARGET --sysr
```

Install awk

```
install -Dm755 a.out "$LBI_ROOT/system/binaries/awk"
install -Dm644 awk.1 "$LBI_ROOT/system/documentation/man-pages/man1/awk.1"
```

Command Explanations

- `make $LWI_MAKE_FLAGS`: Builds awk with the shared make parallelism setting.
- `HOSTCC="cc -g -Wall -pedantic -Wcast-qual"`: Builds the `maketab` helper with the host C compiler because that helper must run during the build.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang --target=$LBI_TARGET --sysroot=$LBI_ROOT ..."`: Builds the final awk binary for the target triple and sysroot.
- `$LWI_CFLAGS` and `$LBI_CUSTOM_LDFLAGS`: Preserve the local compile and link tuning selected in the build environment.
- `install -Dm755 a.out "$LBI_ROOT/system/binaries/awk"`: Installs the generated awk executable under the expected command name and creates parent directories as needed.
- `install -Dm644 awk.1 ...`: Installs the manual page into the book's man-page tree.

6.17 GNU Make 4.4.1

6.17. GNU Make 4.4.1

GNU Make provides the `make` and `gmake` build frontends needed by most target package build systems.

Open standalone page

> **Input assumption:** `make-4.4.1.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.
>
> **Source URL:** `https://gnu.mirror.constant.com/make/make-4.4.1.tar.gz`
>
> **Upstream build note:** GNU Make ships a generated `configure` script in release tarballs and documents a standard `./configure`, `make`, `make install` flow.

Licenses:

- GPL-3.0-or-later

Dependencies:

- musl (libc)
- clang/llvm toolchain
- make (host)

GNU Make is a Makefile build orchestration tool. we need it to provide the target `make` and `gmake` commands used by most package build systems in later chapters.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf make-4.4.1.tar.gz
cd make-4.4.1
```

Configure GNU Make

```
CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS" \
lbi_configure \
    --host="$LBI_TARGET" \
    --without-guile \
    --disable-nls
```

Build GNU Make

```
make $LWI_MAKE_FLAGS
```

Install GNU Make

```
make install DESTDIR="$LBI_ROOT"
ln -sf make "$LBI_ROOT/system/binaries/gmake"
```

Command Explanations

- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler.
- `CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the selected triple and sysroot while preserving local C flag tuning.
- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"`: Keeps link steps in the target sysroot and preserves local linker tuning.
- `lbi_configure --host="$LBI_TARGET"`: Uses the book's configure helper for `/system` paths and marks the build as a cross build.
- `--without-guile`: Disables GNU Make's optional Guile integration.
- `--disable-nls`: Disables native language support so this early build does not need gettext.
- `make $LWI_MAKE_FLAGS`: Builds GNU Make with the shared make parallelism setting.
- `make install DESTDIR="$LBI_ROOT"`: Installs into the target root instead of the host filesystem.
- `ln -sf make "$LBI_ROOT/system/binaries/gmake"`: Provides the common `gmake` command name for projects that explicitly request GNU Make.

6.18 bsdpatch 0.99.1

6.18. bsdpatch 0.99.1

bsdpatch provides the FreeBSD-derived patch utility for applying unified diffs in the target userspace.

Open standalone page

> **Input assumption:** `bsdpatch-v0.99.1.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-2-Clause-FreeBSD

Dependencies:

- musl (libc)

- make

patch is a FreeBSD-derived patch utility made portable by the Chimera Linux bsdpatch project. we need it to apply source patches reliably in later package build flows throughout this book.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf bsdpatch-v0.99.1.tar.gz
cd bsdpatch-0.99.1
```

Build bsdpatch

> **Configure note:** this package does not ship a `configure` script, so `lbi_configure` is not supported here. Build settings are passed directly to `make`.

```
make $LWI_MAKE_FLAGS \
    CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"
```

Install bsdpatch

```
make install \
    DESTDIR="$LBI_ROOT" \
    PREFIX=/system \
    BINDIR=/system/binaries \
    DATADIR=/system/documentation \
    MANDIR=/system/documentation/man-pages/man1 \
    INSTALL_NAME=patch
```

Command Explanations

- `make $LWI_MAKE_FLAGS`: Builds bsdpatch with the shared make parallelism setting.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler.
- `CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the selected triple/sysroot and preserves local C flag tuning.
- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"`: Keeps link steps inside the target sysroot while preserving local linker tuning.

- `make install DESTDIR="$LBI_ROOT"`: Installs into the target root instead of the host system.
- `PREFIX=/system` and `BINDIR=/system/binaries`: Place the installed command in the book's binary layout.
- `DATADIR=/system/documentation` and `MANDIR=/system/documentation/man-pages/man1`: Place support documentation and the manual page in the book's documentation tree.
- `INSTALL_NAME=patch`: Installs the program under the standard `patch` command name.

6.19. bsdsed 0.99.2

bsdsed provides the FreeBSD-derived sed utility for stream editing in the target userspace.

Open standalone page

> **Input assumption:** `bsdsed-v0.99.2.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-2-Clause-FreeBSD

Dependencies:

- musl (libc)
- make

bsdsed is a FreeBSD-derived sed utility made portable by the Chimera Linux bsdsed project. we need it to provide `sed` for stream editing steps used throughout package preparation and build scripts in this book.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf bsdsed-v0.99.2.tar.gz
cd bsdsed-0.99.2
```

Build bsdsed

> **Configure note:** this package does not ship a `configure` script, so `lbi_configure` is not supported here. Build settings are passed directly to `make`.

```
make $LWI_MAKE_FLAGS \
    CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"
```

Install bsdsed

```
make install \
    DESTDIR="$LBI_ROOT" \
    PREFIX=/system \
    BINDIR=/system/binaries \
    DATADIR=/system/documentation \
    MANDIR=/system/documentation/man-pages/man1 \
    INSTALL_NAME=sed
```

Command Explanations

- `make $LWI_MAKE_FLAGS`: Builds bsdsed with the shared make parallelism setting.
- `CC="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"`: Uses the target-prefixed Clang compiler.
- `CFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Targets the selected triple/sysroot and preserves local C flag tuning.
- `LDFLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUSTOM_LDFLAGS"`: Keeps link steps inside the target sysroot while preserving local linker tuning.
- `make install DESTDIR="$LBI_ROOT"`: Installs into the target root instead of the host system.
- `PREFIX=/system` and `BINDIR=/system/binaries`: Place the installed command in the book's binary layout.
- `DATADIR=/system/documentation` and `MANDIR=/system/documentation/man-pages/man1`: Place support documentation and the manual page in the book's documentation tree.
- `INSTALL_NAME=sed`: Installs the program under the standard `sed` command name.

6.20 libarchive 3.8.7

6.20. libarchive 3.8.7

libarchive provides archive libraries and BSD archive tools, including bsdtar, bsdcpio, and bsdunzip.

Open standalone page

> **Input assumption:** `libarchive-3.8.7.tar.xz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-2-Clause (primary)
- BSD-3-Clause, public domain, and other permissive notices in selected files

Dependencies:

- musl (libc)
- cmake
- make
- zlib-ng (libz)

libarchive is a multi-format archive reading and writing library plus BSD archive utilities. we need it to provide `tar`, `cpio`, and `unzip` command compatibility in the target userspace.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf libarchive-3.8.7.tar.xz
cd libarchive-3.8.7
```

Configure libarchive (CMake Path)

> **Working directory requirement:** run this block from the `libarchive-3.8.7` source root (the directory containing `CMakeLists.txt` and `build/version`), not from inside `build/`.

```
lbi_cmake build \
    -DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_AR="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ar" \
    -DCMAKE_RANLIB="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ranlib" \
    -DCMAKE_C_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    -DCMAKE_SHARED_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_C
    -DCMAKE_EXE_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUST
    -DBUILD_SHARED_LIBS=ON \
    -DENABLE_TAR=ON \
    -DENABLE_CPIO=ON \
    -DENABLE_UNZIP=ON \
```

```
    -DENABLE_CAT=OFF \
    -DENABLE_ZLIB=ON \
    -DENABLE_OPENSSL=OFF \
    -DENABLE_BZip2=OFF \
    -DENABLE_LZMA=OFF \
    -DENABLE_ZSTD=OFF \
    -DENABLE_LZ4=OFF \
    -DENABLE_LZO=OFF \
    -DENABLE_LIBB2=OFF \
    -DENABLE_LIBXML2=OFF \
    -DENABLE_EXPAT=OFF \
    -DENABLE_ICONV=OFF \
    -DENABLE_ACL=OFF \
    -DENABLE_XATTR=OFF \
    -DENABLE_TEST=OFF
```

Build libarchive

```
cmake --build build $LWI_MAKE_FLAGS
```

Install libarchive

```
DESTDIR="$LBI_ROOT" cmake --install build
```

Normalize Installed Paths to LBI Layout

> **Path note:** upstream `libarchive` CMake install rules hardcode some destinations (`bin`, `include`, `share/man`). Move those outputs into the book's layout after install.

```
install -d \
    "$LBI_ROOT/system/binaries" \
    "$LBI_ROOT/system/headers" \
    "$LBI_ROOT/system/documentation/man-pages/man1" \
    "$LBI_ROOT/system/documentation/man-pages/man3" \
    "$LBI_ROOT/system/documentation/man-pages/man5"

if [ -d "$LBI_ROOT/system/bin" ]; then
    mv -v "$LBI_ROOT/system/bin/"* "$LBI_ROOT/system/binaries/" 2>/dev/null || t
    rmdir -v "$LBI_ROOT/system/bin" 2>/dev/null || true
```

```
fi

if [ -d "$LBI_ROOT/system/include" ]; then
    mv -v "$LBI_ROOT/system/include/"* "$LBI_ROOT/system/headers/" 2>/dev/null |
    rmdir -v "$LBI_ROOT/system/include" 2>/dev/null || true
fi

for sec in man1 man3 man5; do
    if [ -d "$LBI_ROOT/system/share/man/$sec" ]; then
        mv -v "$LBI_ROOT/system/share/man/$sec/"* \
            "$LBI_ROOT/system/documentation/man-pages/$sec/" 2>/dev/null || true
        rmdir -v "$LBI_ROOT/system/share/man/$sec" 2>/dev/null || true
    fi
done
```

Post-install pkg-config Fix

```
sed -i \
    -e 's|^prefix=.*$|prefix=/system|' \
    -e 's|^exec_prefix=.*$|exec_prefix=${prefix}|' \
    -e 's|^libdir=.*$|libdir=${exec_prefix}/libraries|' \
    -e 's|^includedir=.*$|includedir=${prefix}/headers|' \
    "$LBI_ROOT/system/libraries/pkgconfig/libarchive.pc"
```

Provide `tar`, `cpio`, and `unzip` Commands

```
ln -sf bsdtar "$LBI_ROOT/system/binaries/tar"
ln -sf bsdcpio "$LBI_ROOT/system/binaries/cpio"
ln -sf bsdunzip "$LBI_ROOT/system/binaries/unzip"
```

Command Explanations

- `lbi_cmake build`: Configures libarchive with the book's CMake helper and writes generated files into the `build` directory.
- `-DCMAKE_C_COMPILER`, `-DCMAKE_AR`, and `-DCMAKE_RANLIB`: Use the target-prefixed LLVM compiler and archive tools.
- `-DCMAKE_C_FLAGS`, `-DCMAKE_SHARED_LINKER_FLAGS`, and `-DCMAKE_EXE_LINKER_FLAGS`: Target the selected triple/sysroot and preserve local compile/link tuning.
- `-DBUILD_SHARED_LIBS=ON`: Builds shared libraries for the target system.

- `-DENABLE_TAR=ON`, `-DENABLE_CPIO=ON`, and `-DENABLE_UNZIP=ON`: Builds the archive command-line tools needed for command compatibility.
- `-DENABLE_CAT=OFF`: Skips the optional `bsdcat` tool.
- `-DENABLE_ZLIB=ON`: Enables gzip/deflate support through the installed zlib-compatible library.
- `-DENABLE_OPENSSL=OFF`, `-DENABLE_BZip2=OFF`, `-DENABLE_LZMA=OFF`, `-DENABLE_ZSTD=OFF`, `-DENABLE_LZ4=OFF`, `-DENABLE_LZO=OFF`, `-DENABLE_LIBB2=OFF`, `-DENABLE_LIBXML2=OFF`, `-DENABLE_EXPAT=OFF`, `-DENABLE_ICONV=OFF`, `-DENABLE_ACL=OFF`, and `-DENABLE_XATTR=OFF`: Disable optional dependencies that are not part of this early target userspace.
- `-DENABLE_TEST=OFF`: Skips test programs during the bootstrap build.
- `cmake --build build $LWI_MAKE_FLAGS`: Builds the configured tree using the shared parallel build setting.
- `DESTDIR="$LBI_ROOT" cmake --install build`: Installs into the target root rather than the host filesystem.
- `install -d ...`: Creates the final destination directories required by the book's layout.
- `mv -v "$LBI_ROOT/system/bin/"* ...` and `mv -v "$LBI_ROOT/system/include/"* ...`: Move upstream CMake install outputs into `/system/binaries` and `/system/headers`.
- `for sec in man1 man3 man5; do ... done`: Moves installed manual pages from upstream's `share/man` layout into the book's documentation tree.
- `2>/dev/null || true`: Allows cleanup moves and directory removals to be harmless when an optional directory is empty or absent.
- `sed -i ... libarchive.pc`: Rewrites pkg-config metadata so later builds discover libarchive under `/system/libraries` and `/system/headers`.
- `ln -sf bsdtar ... tar`, `ln -sf bsdcpio ... cpio`, and `ln -sf bsdunzip ... unzip`: Provide standard archive command names through libarchive's BSD tools.

6.21 xz 5.8.3

6.21. xz 5.8.3

xz provides liblzma and command-line tools for .xz and .lzma compression workflows in the target userspace.

Open standalone page

> **Input assumption:** `xz-5.8.3.tar.xz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- oBSD

Dependencies:

- musl (libc)
- cmake
- make

xz is a compression toolkit that provides liblzma and .xz/.lzma command-line tools. we need it to provide `xz` and `liblzma` for compression workflows and later package dependencies.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf xz-5.8.3.tar.xz
cd xz-5.8.3
```

Configure xz (CMake Path)

> **Upstream note:** `INSTALL` documents the CMake options for this package (`XZ_*`, `TUKLIB_*`, and `CMAKE_*`), including `XZ_NLS` and tool toggles.

```
lbi_cmake build \
    -DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_AR="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ar" \
    -DCMAKE_RANLIB="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ranlib" \
    -DCMAKE_INSTALL_PREFIX=/system \
    -DCMAKE_INSTALL_BINDIR=binaries \
    -DCMAKE_INSTALL_SBINDIR=systembinaries \
    -DCMAKE_INSTALL_LIBDIR=libraries \
    -DCMAKE_INSTALL_LIBEXECDIR=systembinaries \
    -DCMAKE_INSTALL_INCLUDEDIR=headers \
    -DCMAKE_INSTALL_SYSCONFDIR=configuration \
    -DCMAKE_INSTALL_LOCALSTATEDIR=variable \
    -DCMAKE_INSTALL_MANDIR=documentation/man-pages \
    -DCMAKE_INSTALL_INFODIR=documentation/info \
    -DCMAKE_INSTALL_DATAROOTDIR=documentation \
    -DCMAKE_INSTALL_DOCDIR=documentation/xz \
    -DCMAKE_C_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    -DCMAKE_SHARED_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_C
    -DCMAKE_EXE_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LBI_CUST
```

```
    -DCMAKE_BUILD_TYPE=Release \
    -DBUILD_SHARED_LIBS=ON \
    -DBUILD_TESTING=OFF \
    -DXZ_NLS=OFF \
    -DXZ_DOXYGEN=OFF
```

Build xz

```
cmake --build build $LWI_MAKE_FLAGS
```

Install xz

```
DESTDIR="$LBI_ROOT" cmake --install build
```

Command Explanations

- `lbi_cmake build`: Configures xz with the book's CMake helper and writes generated files into the `build` directory.
- `-DCMAKE_C_COMPILER`, `-DCMAKE_AR`, and `-DCMAKE_RANLIB`: Use the target-prefixed LLVM compiler and archive tools.
- `-DCMAKE_INSTALL_PREFIX=/system` and the `CMAKE_INSTALL_*DIR` settings: Keep xz's install paths aligned with the book's `/system` layout.
- `-DCMAKE_INSTALL_DATAROOTDIR=documentation` and `-DCMAKE_INSTALL_DOCDIR=documentation/xz`: Place package documentation under the documentation tree instead of `share`.
- `-DCMAKE_C_FLAGS`, `-DCMAKE_SHARED_LINKER_FLAGS`, and `-DCMAKE_EXE_LINKER_FLAGS`: Target the selected triple/sysroot and preserve local compile/link tuning.
- `-DCMAKE_BUILD_TYPE=Release`: Uses the release build profile.
- `-DBUILD_SHARED_LIBS=ON`: Builds shared liblzma libraries for the target.
- `-DBUILD_TESTING=OFF`: Skips test programs during the bootstrap build.
- `-DXZ_NLS=OFF`: Disables native language support so this pass does not depend on gettext.
- `-DXZ_DOXYGEN=OFF`: Skips generated API documentation.
- `cmake --build build $LWI_MAKE_FLAGS`: Builds the configured tree with the shared parallel build setting.
- `DESTDIR="$LBI_ROOT" cmake --install build`: Installs into the target root instead of the host filesystem.

6.22. llvm/clang pass 2 22.1.3

The second llvm/clang pass installs the final compiler toolchain, then refreshes the LLVM runtimes with standalone cross builds and fixes Clang's default search paths for the book's custom filesystem layout.

Open standalone page

> **Input assumption:** `llvm-project-22.1.3.src.tar.xz` is already present in `LBI_SOURCES`, and sections 5.4 and 5.5 completed successfully.

Licenses

- Apache-2.0 WITH LLVM-exception

Dependencies

- musl (libc)
- cmake
- ninja
- zlib-ng (libz)
- llvm/clang pass 1

llvm is a modular compiler infrastructure project. we need it to provide the final compiler backend, linker integration, and toolchain libraries for the target system.

clang is a C and C++ frontend for LLVM. we need it to provide the final `clang` and `clang++` compilers for the target system.

compiler-rt is the LLVM runtime support library project. we need it to provide Clang builtins and startup objects.

libunwind is a platform-independent stack unwinding library. we need it to provide unwind support used by C++ exception handling.

libcxxabi is the LLVM C++ ABI support library. we need it to provide ABI and exception runtime support for libcxx.

libcxx is the LLVM C++ standard library implementation. we need it to provide the C++ standard library used by Clang in later stages.

Extract Sources and Enter the LLVM Build Root

```
cd "$LBI_SOURCES"
tar -xf llvm-project-22.1.3.src.tar.xz
```

```
cd llvm-project-22.1.3.src/llvm
```

Configure llvm/clang Pass 2

> **Cross-build note:** this pass follows LLVM's cross-compilation guidance for the compiler build itself. The top-level `LLVM_ENABLE_RUNTIMES` flow is intentionally not used here, because it tries to configure compiler-rt builtins with the just-built target compiler, which is not host-runnable in this setup.

```
lbi_cmake build-llvm-pass2 \
    -G Ninja \
    -DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_CXX_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang++" \
    -DCMAKE_AR="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ar" \
    -DCMAKE_RANLIB="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ranlib" \
    -DCMAKE_NM="$LBI_ROOT/system/tools/bin/$LBI_TARGET-nm" \
    -DCMAKE_OBJCOPY="$LBI_ROOT/system/tools/bin/$LBI_TARGET-objcopy" \
    -DCMAKE_OBJDUMP="$LBI_ROOT/system/tools/bin/$LBI_TARGET-objdump" \
    -DCMAKE_STRIP="$LBI_ROOT/system/tools/bin/$LBI_TARGET-strip" \
    -DCMAKE_SYSROOT="$LBI_ROOT" \
    -DCMAKE_C_COMPILER_LAUNCHER="ccache" \
    -DCMAKE_CXX_COMPILER_LAUNCHER="ccache" \
    -DCMAKE_FIND_ROOT_PATH="$LBI_ROOT;$LBI_ROOT/system" \
    -DCMAKE_FIND_ROOT_PATH_MODE_PROGRAM=NEVER \
    -DCMAKE_FIND_ROOT_PATH_MODE_LIBRARY=ONLY \
    -DCMAKE_FIND_ROOT_PATH_MODE_INCLUDE=ONLY \
    -DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=ONLY \
    -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY \
    -DCMAKE_C_COMPILER_TARGET="$LBI_TARGET" \
    -DCMAKE_CXX_COMPILER_TARGET="$LBI_TARGET" \
    -DCMAKE_ASM_COMPILER_TARGET="$LBI_TARGET" \
    -DCMAKE_C_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -isystem /system/h
    -DCMAKE_CXX_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -isystem /system
    -DCMAKE_SHARED_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -B/sys
    -DCMAKE_EXE_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -B/system
    -DLLVM_NATIVE_TOOL_DIR="$LBI_ROOT/system/tools/bin" \
    -DLLVM_TABLEGEN="$LBI_ROOT/system/tools/bin/llvm-tblgen" \
    -DCLANG_TABLEGEN="$LBI_ROOT/system/tools/bin/clang-tblgen" \
    -DLLVM_ENABLE_PROJECTS="clang;lld" \
    -DLLVM_INSTALL_BINUTILS_SYMLINKS=ON \
```

```
    -DLLVM_HOST_TRIPLE="$LBI_TARGET" \
    -DLLVM_TARGETS_TO_BUILD="X86" \
    -DLLVM_DEFAULT_TARGET_TRIPLE="$LBI_TARGET" \
    -DDEFAULT_SYSROOT="/" \
    -DCLANG_DEFAULT_CXX_STDLIB=libc++ \
    -DCLANG_DEFAULT_LINKER=lld \
    -DCLANG_DEFAULT_RTLIB=compiler-rt \
    -DCLANG_DEFAULT_UNWINDLIB=libunwind \
    -DLLVM_ENABLE_ZLIB=ON \
    -DLLVM_ENABLE_ZSTD=OFF \
    -DLLVM_ENABLE_LIBXML2=OFF \
    -DLLVM_INCLUDE_TESTS=OFF \
    -DLLVM_BUILD_TESTS=OFF \
    -DCMAKE_BUILD_TYPE=Release
```

Build llvm/clang Pass 2

```
cmake --build build-llvm-pass2 $LWI_MAKE_FLAGS
```

Install llvm/clang Pass 2

```
DESTDIR="$LBI_ROOT" cmake --install build-llvm-pass2
```

Refresh LLVM C++ Runtimes

```
cd "$LBI_SOURCES/llvm-project-22.1.3.src/runtimes"

lbi_cmake build-runtimes-pass2 \
    -G Ninja \
    -DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_CXX_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang++" \
    -DLLVM_CMAKE_DIR="$LBI_SOURCES/llvm-project-22.1.3.src/llvm/build-llvm-pass2
    -DCMAKE_SYSROOT="$LBI_ROOT" \
    -DCMAKE_C_COMPILER_LAUNCHER="ccache" \
    -DCMAKE_CXX_COMPILER_LAUNCHER="ccache" \
    -DCMAKE_FIND_ROOT_PATH="$LBI_ROOT;$LBI_ROOT/system" \
    -DCMAKE_FIND_ROOT_PATH_MODE_PROGRAM=NEVER \
    -DCMAKE_FIND_ROOT_PATH_MODE_INCLUDE=ONLY \
    -DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=ONLY \
    -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY \
```

```
    -DCMAKE_C_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -isystem /system/h
    -DCMAKE_CXX_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -isystem /system
    -DCMAKE_EXE_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -B/system
    -DLLVM_ENABLE_RUNTIMES="libunwind;libcxxabi;libcxx" \
    -DLLVM_ENABLE_LIBXML2=OFF \
    -DLLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF \
    -DLLVM_INCLUDE_TESTS=OFF \
    -DLLVM_INCLUDE_DOCS=OFF \
    -DLIBUNWIND_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBUNWIND_INCLUDE_TESTS=OFF \
    -DLIBUNWIND_INCLUDE_DOCS=OFF \
    -DLIBCXXABI_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBCXXABI_INCLUDE_TESTS=OFF \
    -DLIBCXX_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBCXX_INCLUDE_TESTS=OFF \
    -DLIBCXX_INCLUDE_BENCHMARKS=OFF \
    -DLIBCXX_INCLUDE_DOCS=OFF \
    -DLIBCXX_HAS_MUSL_LIBC=ON \
    -DLIBCXX_HAS_ATOMIC_LIB=OFF \
    -DLIBCXXABI_HAS_CXA_THREAD_ATEXIT_IMPL=OFF \
    -DLIBCXXABI_USE_LLVM_UNWINDER=ON \
    -DLIBCXX_USE_COMPILER_RT=ON \
    -DLIBCXXABI_USE_COMPILER_RT=ON \
    -DLIBUNWIND_USE_COMPILER_RT=ON \
    -DCMAKE_BUILD_TYPE=Release

cmake --build build-runtimes-pass2 $LWI_MAKE_FLAGS
DESTDIR="$LBI_ROOT" cmake --install build-runtimes-pass2
```

Build compiler-rt Builtins and CRT Objects

> **Compiler-rt note:** this follows upstream compiler-rt cross-build guidance for standalone builtins and CRT builds instead of the failing top-level LLVM runtimes wrapper.

```
cd "$LBI_SOURCES/llvm-project-22.1.3.src/compiler-rt"

lbi_cmake build-compiler-rt-pass2 \
    -G Ninja \
    -DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_CXX_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang++" \
```

```
    -DCMAKE_ASM_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang" \
    -DCMAKE_AR="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ar" \
    -DCMAKE_NM="$LBI_ROOT/system/tools/bin/$LBI_TARGET-nm" \
    -DCMAKE_RANLIB="$LBI_ROOT/system/tools/bin/$LBI_TARGET-ranlib" \
    -DLLVM_CMAKE_DIR="$LBI_SOURCES/llvm-project-22.1.3.src/llvm/build-llvm-pass2
    -DCMAKE_SYSROOT="$LBI_ROOT" \
    -DCMAKE_C_COMPILER_LAUNCHER="ccache" \
    -DCMAKE_CXX_COMPILER_LAUNCHER="ccache" \
    -DCMAKE_C_COMPILER_TARGET="$LBI_TARGET" \
    -DCMAKE_CXX_COMPILER_TARGET="$LBI_TARGET" \
    -DCMAKE_ASM_COMPILER_TARGET="$LBI_TARGET" \
    -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY \
    -DCMAKE_C_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -isystem /system/h
    -DCMAKE_CXX_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -isystem /system
    -DCMAKE_ASM_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS" \
    -DCMAKE_EXE_LINKER_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT -B/system
    -DCOMPILER_RT_INSTALL_PATH=/system/lib/clang/22 \
    -DCOMPILER_RT_BUILD_BUILTINS=ON \
    -DCOMPILER_RT_BUILD_CRT=ON \
    -DCOMPILER_RT_BUILD_LIBFUZZER=OFF \
    -DCOMPILER_RT_BUILD_MEMPROF=OFF \
    -DCOMPILER_RT_BUILD_ORC=OFF \
    -DCOMPILER_RT_BUILD_PROFILE=OFF \
    -DCOMPILER_RT_BUILD_CTX_PROFILE=OFF \
    -DCOMPILER_RT_BUILD_SANITIZERS=OFF \
    -DCOMPILER_RT_BUILD_XRAY=OFF \
    -DCOMPILER_RT_DEFAULT_TARGET_ONLY=ON \
    -DCOMPILER_RT_INCLUDE_TESTS=OFF \
    -DLLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF \
    -DCMAKE_BUILD_TYPE=Release

cmake --build build-compiler-rt-pass2 --target builtins $LWI_MAKE_FLAGS
cmake --build build-compiler-rt-pass2 --target crt $LWI_MAKE_FLAGS
DESTDIR="$LBI_ROOT" cmake --install build-compiler-rt-pass2
```

Make Clang Resource Files Visible in the Final Tree

```
mkdir -p "$LBI_ROOT/system/lib/clang/22/lib/$LBI_TARGET"

if [ -f "$LBI_ROOT/system/tools/lib/clang/22/lib/$LBI_TARGET/libclang_rt.builtin
    ln -sf "/system/tools/lib/clang/22/lib/$LBI_TARGET/libclang_rt.builtins.a" \
```

```
        "$LBI_ROOT/system/lib/clang/22/lib/$LBI_TARGET/libclang_rt.builtins.a"
fi
```

Create Compiler and Linker Compatibility Symlinks

```
cd "$LBI_ROOT/system/binaries"

ln -sf ld.lld ld
ln -sf clang cc
ln -sf clang++ c++

ln -sf clang "$LBI_TARGET-clang"
ln -sf clang++ "$LBI_TARGET-clang++"
ln -sf clang "$LBI_TARGET-cc"
ln -sf clang++ "$LBI_TARGET-c++"
ln -sf ld.lld "$LBI_TARGET-ld"
```

Link Clang Startup Objects for musl Compatibility

```
CRTBEGIN_OBJ=$(find "$LBI_ROOT/system/libraries/clang" \
    -type f \( -name 'crtbeginS.o' -o -name 'clang_rt.crtbegin*.o' \) | head -n1
CRTEND_OBJ=$(find "$LBI_ROOT/system/libraries/clang" \
    -type f \( -name 'crtendS.o' -o -name 'clang_rt.crtend*.o' \) | head -n1)

CRT_DIR=$(dirname "$CRTBEGIN_OBJ")

if [ -n "$CRTBEGIN_OBJ" ] && [ -n "$CRTEND_OBJ" ]; then
    ln -sf "$(basename "$CRTBEGIN_OBJ")" "$CRT_DIR/crtbeginS.o"
    ln -sf "$(basename "$CRTEND_OBJ")" "$CRT_DIR/crtendS.o"

    ln -sf "${CRTBEGIN_OBJ#$LBI_ROOT/system}" "$LBI_ROOT/system/libraries/crtbeg
    ln -sf "${CRTEND_OBJ#$LBI_ROOT/system}" "$LBI_ROOT/system/libraries/crtendS.
fi
```

Add Clang Driver Wrapper Defaults

```
mv "$LBI_ROOT/system/binaries/clang" "$LBI_ROOT/system/binaries/clang.real"
mv "$LBI_ROOT/system/binaries/clang++" "$LBI_ROOT/system/binaries/clang++.real"

cat > "$LBI_ROOT/system/binaries/clang" <<'EOF'
```

```
#!/bin/sh
exec /system/binaries/clang.real \
    -isystem /system/headers \
    -B/system/libraries \
    -Wno-unused-command-line-argument \
    -B/system/libraries/clang/22/lib/linux \
    -L/system/libraries \
    "$@"
EOF

cat > "$LBI_ROOT/system/binaries/clang++" <<'EOF'
#!/bin/sh
exec /system/binaries/clang++.real \
    -nostdinc++ \
    -I/system/headers/c++/v1 \
    -isystem /system/libraries/clang/22/include \
    -isystem /system/headers \
    -B/system/libraries \
    -B/system/libraries/clang/22/lib/linux \
    -L/system/libraries \
    -Wl,-rpath,/system/libraries \
    "$@" \
    -Wno-unused-command-line-argument \
    -lc++ \
    -lc++abi \
    -lunwind
EOF

chmod 755 "$LBI_ROOT/system/binaries/clang" "$LBI_ROOT/system/binaries/clang++"

cd "$LBI_ROOT/system/binaries"
ln -sf clang cc
ln -sf clang++ c++
ln -sf clang "$LBI_TARGET-clang"
ln -sf clang++ "$LBI_TARGET-clang++"
ln -sf clang "$LBI_TARGET-cc"
ln -sf clang++ "$LBI_TARGET-c++"
```

Command Explanations

- `LBI_HOST_NINJA=$(command -v ninja || command -v samu || true)`: Finds a host-runnable Ninja-compatible build tool before CMake has a chance to cache a target-side executable.

- `if [ -z "$LBI_HOST_NINJA" ]; then ... exit 1; fi`: Stops early with a clear error if the host does not have `ninja` or `samu`.
- `lbi_cmake build-llvm-pass2`: Uses the book's CMake helper for install layout while placing generated files in the `build-llvm-pass2` directory.
- `-DCMAKE_C_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"` and `-DCMAKE_CXX_COMPILER=...`: Build pass 2 with the pass 1 cross compiler instead of the host compiler.
- `-DCMAKE_AR`, `-DCMAKE_RANLIB`, `-DCMAKE_NM`, `-DCMAKE_OBJCOPY`, `-DCMAKE_OBJDUMP`, and `-DCMAKE_STRIP`: Use the target-prefixed LLVM binutils-compatible tools.
- `-DCMAKE_SYSROOT="$LBI_ROOT"` and the `CMAKE_FIND_ROOT_PATH*` settings: Keep configure checks and dependency lookups inside the target tree while still allowing host build tools such as `cmake` and `ninja`.
- `-DCMAKE_MAKE_PROGRAM="$LBI_HOST_NINJA"`: Pins the Ninja generator to a host-runnable build executor so CMake does not cache a target-side `/usr/bin/ninja` from `LBI_ROOT`.
- `-DCMAKE_C_COMPILER_LAUNCHER="ccache"` and `-DCMAKE_CXX_COMPILER_LAUNCHER="ccache"`: Use ccache for repeated compiler invocations when it is available.
- `-DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY`: Prevent configure probes from requiring runnable target executables.
- `-DCMAKE_C_COMPILER_TARGET`, `-DCMAKE_CXX_COMPILER_TARGET`, and `-DCMAKE_ASM_COMPILER_TARGET`: Tell CMake and Clang which target triple to use for C, C++, and assembly.
- `-DCMAKE_C_FLAGS=... -isystem /system/headers` and `-DCMAKE_CXX_FLAGS=... -isystem /system/headers`: Make the build use the book's custom libc header location instead of assuming `/usr/include`.
- `-DCMAKE_EXE_LINKER_FLAGS=... -B/system/libraries -B/system/libraries/clang/22/lib/linux -L/system/libraries` and the matching shared-linker flags: Make link steps find musl startup files, Clang CRT objects, and libc in the book's custom library layout.
- `-DLLVM_NATIVE_TOOL_DIR`, `-DLLVM_TABLEGEN`, and `-DCLANG_TABLEGEN`: Reuse the pass 1 native helper tools instead of rebuilding host tools in this pass.
- `-DLLVM_ENABLE_PROJECTS="clang;lld"`: Build the final frontend and linker in the main compiler pass.
- `-DLLVM_INSTALL_BINUTILS_SYMLINKS=ON`: Installs LLVM tool aliases for common binutils command names.
- `-DLLVM_HOST_TRIPLE="$LBI_TARGET"`, `-DLLVM_TARGETS_TO_BUILD="X86"`, and `-DLLVM_DEFAULT_TARGET_TRIPLE="$LBI_TARGET"`: Keep this pass focused on the target architecture and make the installed compiler default to the book's target triple.
- `-DDEFAULT_SYSROOT="/"`: Make the installed Clang use the target root layout at runtime instead of baking in the temporary build host path.

- `-DCLANG_DEFAULT_CXX_STDLIB=libc++`, `-DCLANG_DEFAULT_LINKER=lld`, `-DCLANG_DEFAULT_RTLIB=compiler-rt`, and `-DCLANG_DEFAULT_UNWINDLIB=libunwind`: Set the intended default runtime family for later builds.
- `-DLLVM_ENABLE_ZLIB=ON`: Enables zlib support using the target zlib-compatible library.
- `-DLLVM_ENABLE_ZSTD=OFF` and `-DLLVM_ENABLE_LIBXML2=OFF`: Disable optional dependencies that are not part of this pass.
- `-DLLVM_INCLUDE_TESTS=OFF` and `-DLLVM_BUILD_TESTS=OFF`: Skip LLVM tests in the target compiler build.
- `-DCMAKE_BUILD_TYPE=Release`: Uses release optimization for the final target compiler.
- `cmake --build build-llvm-pass2 $LWI_MAKE_FLAGS`: Builds the configured compiler tree with the shared parallel build setting.
- `DESTDIR="$LBI_ROOT" cmake --install build-llvm-pass2`: Installs the final compiler into the target root instead of the host system.
- `LLVM_ENABLE_RUNTIMES` is omitted from the main pass on purpose: the top-level runtime bootstrap tries to configure compiler-rt builtins with the just-built target compiler, which fails in this cross setup because that compiler is not host-runnable.
- The standalone `build-runtimes-pass2` step refreshes `libunwind`, `libcxxabi`, and `libcxx` with the final compiler and the final library/header layout.
- `-DLLVM_CMAKE_DIR="$LBI_SOURCES/llvm-project-22.1.3.src/llvm/build-llvm-pass2/lib/cmake/llvm"`: Points standalone runtime and compiler-rt builds at the just-configured LLVM package files.
- The runtime `CMAKE_FIND_ROOT_PATH*`, `CMAKE_TRY_COMPILE_TARGET_TYPE`, compile flags, and linker flags repeat the same cross-build protections used for the main pass.
- `-DLLVM_ENABLE_RUNTIMES="libunwind;libcxxabi;libcxx"`: Builds only the C++ runtime stack in the standalone runtime pass.
- `-DLLVM_INCLUDE_DOCS=OFF`, `LIBUNWIND_INCLUDE_*`, `LIBCXXABI_INCLUDE_TESTS=OFF`, and `LIBCXX_INCLUDE_*`: Skip tests, benchmarks, and documentation for the runtime rebuild.
- `-DLLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF`: Keep runtime outputs out of target-triple subdirectories where possible so they land in the normal library layout.
- `-DLIBUNWIND_INSTALL_LIBRARY_DIR=/system/libraries`, `-DLIBCXXABI_INSTALL_LIBRARY_DIR=/system/libraries`, and `-DLIBCXX_INSTALL_LIBRARY_DIR=/system/libraries`: Force the runtime libraries into the book's custom `/system/libraries` path.
- `-DLIBCXX_HAS_MUSL_LIBC=ON`: Enables libcxx behavior for the musl target.
- `-DLIBCXX_HAS_ATOMIC_LIB=OFF`: Prevents an external `-latomic` dependency in this pass.
- `-DLIBCXXABI_HAS_CXA_THREAD_ATEXIT_IMPL=OFF`: Avoids relying on the glibc-specific `__cxa_thread_atexit_impl` symbol.
- `-DLIBCXXABI_USE_LLVM_UNWINDER=ON`: Links libcxxabi against libunwind from this runtime set.

- `-DLIBCXX_USE_COMPILER_RT=ON`, `-DLIBCXXABI_USE_COMPILER_RT=ON`, and `-DLIBUNWIND_USE_COMPILER_RT=ON`: Use compiler-rt runtime libraries instead of libgcc-style runtime assumptions.
- `cmake --build build-runtimes-pass2 $LWI_MAKE_FLAGS` and `DESTDIR="$LBI_ROOT" cmake --install build-runtimes-pass2`: Build and install the refreshed C++ runtime stack into the target root.
- `lbi_cmake build-compiler-rt-pass2`: Configures compiler-rt as a standalone cross build.
- `-DCMAKE_ASM_COMPILER="$LBI_ROOT/system/tools/bin/$LBI_TARGET-clang"` and `-DCMAKE_ASM_COMPILER_TARGET="$LBI_TARGET"`: Build assembly runtime objects with the target Clang.
- `-DCMAKE_ASM_FLAGS="--target=$LBI_TARGET --sysroot=$LBI_ROOT $LWI_CFLAGS"`: Applies the target triple and sysroot to assembly compilation.
- `-DCOMPILER_RT_INSTALL_PATH=/system/lib/clang/22`: Install compiler-rt into Clang's resource-directory layout so the driver can find builtins by default.
- `-DCOMPILER_RT_BUILD_BUILTINS=ON` and `-DCOMPILER_RT_BUILD_CRT=ON`: Build the Clang builtins library and CRT startup objects needed by later links.
- `-DCOMPILER_RT_BUILD_LIBFUZZER=OFF`, `-DCOMPILER_RT_BUILD_MEMPROF=OFF`, `-DCOMPILER_RT_BUILD_ORC=OFF`, `-DCOMPILER_RT_BUILD_PROFILE=OFF`, `-DCOMPILER_RT_BUILD_CTX_PROFILE=OFF`, `-DCOMPILER_RT_BUILD_SANITIZERS=OFF`, and `-DCOMPILER_RT_BUILD_XRAY=OFF`: Skip runtime components that are not needed for this target bootstrap.
- `-DCOMPILER_RT_DEFAULT_TARGET_ONLY=ON`: Limits compiler-rt output to the configured target.
- `-DCOMPILER_RT_INCLUDE_TESTS=OFF`: Skips compiler-rt tests.
- `cmake --build build-compiler-rt-pass2 --target builtins $LWI_MAKE_FLAGS` and `--target crt`: Build only the compiler-rt targets needed by the target toolchain.
- `DESTDIR="$LBI_ROOT" cmake --install build-compiler-rt-pass2`: Installs compiler-rt outputs into the target root.
- `mkdir -p "$LBI_ROOT/system/lib/clang/22/lib/$LBI_TARGET"`: Creates the Clang resource-library directory expected by the driver.
- The resource-file symlink step: Covers cases where compiler-rt still lands under `/system/tools/lib/clang/...` while the Clang driver looks under `/system/lib/clang/...`.
- `cd "$LBI_ROOT/system/binaries"`: Runs compiler/linker symlink creation from the target binary directory so the relative links stay simple.
- `ln -sf ld.lld ld`, `ln -sf clang cc`, and `ln -sf clang++ c++`: Provide conventional compiler and linker command names.
- The `$LBI_TARGET-*` symlinks: Preserve cross-style tool names for later build systems that look for target-prefixed compilers.
- `find "$LBI_ROOT/system/libraries/clang" ... | head -n1`: Locates the installed compiler-rt CRT begin/end objects regardless of their exact upstream-generated names.

- `CRT_DIR=$(dirname "$CRTBEGIN_OBJ")`: Records the resource directory containing those CRT objects.
- The `crtbeginS.o` and `crtendS.o` links: Make the Clang CRT objects visible both in the resource directory and in `/system/libraries`, where musl-oriented links and later builds expect them.
- `mv "$LBI_ROOT/system/binaries/clang" ... clang.real` and the matching `clang++` move: Preserve the real compiler binaries before installing wrapper scripts at the public command names.
- `cat > "$LBI_ROOT/system/binaries/clang" <<'EOF'` and the matching `clang++` block: Writes wrapper scripts with fixed default include, startup-object, and library search paths.
- The Clang wrapper scripts: Add the header path and startup-object/library search paths needed by the book's non-FHS layout, so later package builds do not have to export manual `CPPFLAGS`, `CFLAGS`, or `LDFLAGS` just to find musl headers and libc.
- `chmod 755 "$LBI_ROOT/system/binaries/clang" "$LBI_ROOT/system/binaries/clang++"`: Makes the wrapper scripts executable.
- The final compiler symlink block: Recreates `cc`, `c++`, and target-prefixed aliases so they point at the wrapper scripts rather than bypassing them.

7.1 Introduction

7.1. Introduction

Chapter 7 moves into the target environment by entering chroot and building enough of the utility set from within that system to continue the rest of the build.

Open standalone page

> **Goal:** enter the target system with `chroot`, then build the utilities needed in that environment so the rest of the system can be built from inside it.

Chapter 6 established a minimal working base. Chapter 7 transitions from cross-built staging into target-native operation by entering `chroot` and completing the utility builds needed to continue the remaining system work from inside the target tree.

This chapter is focused on two outcomes:

- establish a reliable `chroot` environment that behaves predictably for package builds;
- build the core utilities needed for the remaining system build so the process no longer depends on temporary bootstrap assumptions.

By the end of this chapter, the target system should be capable of continuing package work from within its own environment, with sufficient utility coverage in place to build the rest of the system.

7.2. Reset Target Tree Ownership to root

Before entering chroot, hand ownership of the entire target tree back to root so in-chroot package work runs with expected permissions.

Open standalone page

> **Goal:** make the whole `$LBI_ROOT` tree owned by `root:root` before entering `chroot`.

In earlier setup, ownership of the target tree may have been handed to a normal user to simplify build steps. At this point, before entering `chroot`, ownership should be reset to `root` recursively.

All commands in this section must be run as the `root` user.

Become root

Use whichever privilege method your host uses:

```
# Option 1: sudo
sudo -i

# Option 2: doas
doas -s

# Option 3: su
su -
```

Reset ownership of the whole target tree

```
chown -R root:root "$LBI_ROOT"
```

That command should cover the entire target tree under `$LBI_ROOT`, including directories, files, and symlinks, so ownership is consistent before `chroot` work begins.

> **Safety check:** verify `$LBI_ROOT` is set to the intended mounted target path before pressing Enter. Running recursive `chown` against the wrong path is an excellent way to ruin your day.

Command Explanations

- `sudo -i`, `doas -s`, and `su -`: Show common ways to open a root shell before changing ownership of the target tree.
- `chown -R root:root "$LBI_ROOT"`: Recursively changes the target tree ownership to root so the chroot starts with normal system ownership instead of the build user.

7.3 Create virtual filesystem link targets

7.3. Create virtual filesystem link targets

Create target-side directories and top-level links for runtime virtual filesystems, then mount them before entering chroot.

Open standalone page

> **Goal:** create `$LBI_ROOT/system/devices`, `$LBI_ROOT/system/processes`, `$LBI_ROOT/system/system`, and `$LBI_ROOT/system/run`, link top-level paths to them, and mount the runtime virtual filesystems.

This step prepares mount targets for runtime virtual filesystems in the custom tree layout before entering `chroot`.

All commands in this section should be run as `root`.

Create the target directories

```
mkdir -p "$LBI_ROOT/system/devices"
mkdir -p "$LBI_ROOT/system/processes"
mkdir -p "$LBI_ROOT/system/system"
mkdir -p "$LBI_ROOT/system/run"

mkdir -p "$LBI_ROOT/system/devices/pts"
mkdir -p "$LBI_ROOT/system/devices/shm"
```

Create top-level compatibility links

```
rm -f "$LBI_ROOT/dev"
rm -f "$LBI_ROOT/proc"
rm -f "$LBI_ROOT/sys"
rm -f "$LBI_ROOT/run"
```

```
ln -sf system/devices "$LBI_ROOT/dev"
ln -sf system/processes "$LBI_ROOT/proc"
ln -sf system/system "$LBI_ROOT/sys"
ln -sf system/run "$LBI_ROOT/run"
```

Mount virtual filesystems

```
mount -o bind /dev "$LBI_ROOT/dev"
mount -t devpts devpts "$LBI_ROOT/dev/pts" -o gid=5,mode=0620
mount -t proc proc "$LBI_ROOT/proc"
mount -t sysfs sysfs "$LBI_ROOT/sys"
mount -t tmpfs tmpfs "$LBI_ROOT/run"

if [ -h "$LBI_ROOT/dev/shm" ]; then
    install -d -m 1777 "$LBI_ROOT/system/run/shm"
else
    mount -t tmpfs -o nosuid,nodev tmpfs "$LBI_ROOT/dev/shm"
fi
```

POSIX `sh` helper script

A ready-to-run helper script is provided here:

- scripts/mount-virtual-fs.sh

Run it as `root` with `LBI_ROOT` exported in your environment.

Quick verification

```
ls -ld "$LBI_ROOT/dev" "$LBI_ROOT/proc" "$LBI_ROOT/sys" "$LBI_ROOT/run"
```

Command Explanations

- `mkdir -p "$LBI_ROOT/system/..."`: Creates the target-side mount points used for device, process, sysfs, runtime, pts, and shared-memory filesystems.
- `rm -f "$LBI_ROOT/dev"` and the matching commands: Remove stale top-level placeholders before recreating the compatibility links.
- `ln -sf system/devices "$LBI_ROOT/dev"`: Maps the conventional `/dev` path to the book's `/system/devices` layout.
- `ln -sf system/processes`, `system/system`, and `system/run`: Provide `/proc`, `/sys`, and `/run` compatibility paths for tools that expect standard Linux locations.

- `mount -o bind /dev "$LBI_ROOT/dev"`: Makes the host device tree visible inside the target tree for chroot work.
- `mount -t devpts`, `mount -t proc`, `mount -t sysfs`, and `mount -t tmpfs`: Mount the virtual filesystems needed by shells, process tools, kernel interfaces, and runtime state.
- `install -d -m 1777 "$LBI_ROOT/system/run/shm"`: Creates a world-writable sticky shared-memory directory when `/dev/shm` is a symlink.
- `ls -ld ...`: Verifies that the top-level compatibility paths resolve to the intended targets.

7.4 Copy selected build variables and helper functions into target profile

7.4. Copy selected build variables and helper functions into target profile

Write selected current Linux by Intent build variables and reusable build helper functions into the target profile so chroot starts with the expected tuning and tools.

Open standalone page

> **Goal:** create `$LBI_ROOT/system/configuration/profile` and a `zprofile` link with selected current book variables (defaulting each one to a single-space string when empty or unset) and reusable configure/meson/cmake helper functions.

Before entering `chroot`, write a target profile file that carries selected book variables into the new environment.

This step intentionally excludes these host-path and mirror variables:

- `LBI_ROOT`
- `LBI_TOOLS`
- `LBI_SOURCES`
- `LBI_BOOT`
- `LBI_ESP_MOUNT`

All commands in this section should be run as `root`.

Write the target profile with `cat <<EOF`

```
dollar='$'
cat > "$LBI_ROOT/system/configuration/profile" <<EOF
# Linux by Intent variable handoff profile
```

```
export LBI_ARCH="${LBI_ARCH:- }"
export LBI_TARGET="${LBI_TARGET:- }"
export LWI_MAKE_JOBS="${LWI_MAKE_JOBS:- }"
export LWI_MAKE_FLAGS="${LWI_MAKE_FLAGS:- }"
export LWI_CFLAGS="${LWI_CFLAGS:- }"
export LWI_CXXFLAGS="${LWI_CXXFLAGS:- }"
export LBI_CUSTOM_LDFLAGS="${LBI_CUSTOM_LDFLAGS:- }"
export LBI_BOOTLOADER_ID="${LBI_BOOTLOADER_ID:- }"

lbi_configure() {
    CFLAGS="${LWI_CFLAGS}" \
    CXXFLAGS="${LWI_CXXFLAGS}" \
    LDFLAGS="${LBI_CUSTOM_LDFLAGS}" \
    ./configure \
        --prefix=/system \
        --bindir=/system/binaries \
        --sbindir=/system/systembinaries \
        --libdir=/system/libraries \
        --libexecdir=/system/systembinaries \
        --includedir=/system/headers \
        --sysconfdir=/system/configuration \
        --localstatedir=/system/variable \
        --mandir=/system/documentation/man-pages \
        --infodir=/system/documentation/info \
        "${dollar}@"
}

lbi_meson() {
    lbi_meson_builddir=build

    case "${dollar}{1-}" in
        '') ;;
        -*) ;;
        *)
            lbi_meson_builddir=${dollar}1
            shift
            ;;
    esac

    meson setup "${dollar}lbi_meson_builddir" \
        --prefix=/system \
```

```
        --bindir=/system/binaries \
        --sbindir=/system/systembinaries \
        --libdir=/system/libraries \
        --libexecdir=/system/systembinaries \
        --includedir=/system/headers \
        --sysconfdir=/system/configuration \
        --localstatedir=/system/variable \
        --mandir=/system/documentation/man-pages \
        --infodir=/system/documentation/info \
        "${dollar}@"
}

lbi_cmake() {
    lbi_cmake_builddir=build

    case "${dollar}{1-}" in
        '') ;;
        -*) ;;
        *)
            lbi_cmake_builddir=${dollar}1
            shift
            ;;
    esac

    cmake -S . -B "${dollar}lbi_cmake_builddir" \
        -DCMAKE_INSTALL_PREFIX=/system \
        -DCMAKE_INSTALL_BINDIR=/system/binaries \
        -DCMAKE_INSTALL_SBINDIR=/system/systembinaries \
        -DCMAKE_INSTALL_LIBDIR=/system/libraries \
        -DCMAKE_INSTALL_LIBEXECDIR=/system/systembinaries \
        -DCMAKE_INSTALL_INCLUDEDIR=/system/headers \
        -DCMAKE_INSTALL_SYSCONFDIR=/system/configuration \
        -DCMAKE_INSTALL_LOCALSTATEDIR=/system/variable \
        -DCMAKE_INSTALL_MANDIR=/system/documentation/man-pages \
        -DCMAKE_INSTALL_INFODIR=/system/documentation/info \
        "${dollar}@"
}
EOF
```

Because the heredoc delimiter is unquoted (`<<EOF`), the file receives the **current** values at write time. The `${VAR:- }` form makes each missing or empty value become a single space.

Quick verification

```
cat "$LBI_ROOT/system/configuration/profile"
```

Command Explanations

- `dollar='$'`: Stores a literal dollar sign so the generated profile can contain variables that expand later inside the chroot.
- `cat > "$LBI_ROOT/system/configuration/profile" <<EOF`: Writes the target login profile used by later chroot shells.
- `export LBI_ARCH`, `LBI_TARGET`, `LWI_MAKE_JOBS`, `LWI_MAKE_FLAGS`, `LWI_CFLAGS`, and `LWI_CXXFLAGS`: Carries the selected build identity and tuning variables into the target environment.
- `export LBI_CUSTOM_LDFLAGS`: Preserves local linker tuning for package builds that still consume the linker flag variable.
- `lbi_configure`, `lbi_meson`, and `lbi_cmake`: Defines the same install-layout helper functions inside the target profile for chapter 8 builds.
- `export PATH=/system/binaries:/system/systembinaries`: Keeps target commands ahead of any compatibility paths inside the chroot.
- `cat "$LBI_ROOT/system/configuration/profile"`: Prints the generated profile for review before entering the target environment.

7.5 Enter chroot environment

7.5. Enter chroot environment

Enter the target system with a clean environment and a login shell so package work runs inside the target tree.

Open standalone page

> **Goal:** start a clean `chroot` shell in `$LBI_ROOT` using the target layout paths and login profile.

Run this as `root` after completing the previous setup steps in chapter 7.

Enter chroot

```
chroot "$LBI_ROOT" /system/binaries/env -i \
    HOME=/system/charlie \
    TERM="$TERM" \
    SHELL=/bin/oksh \
```

```
    PS1='(chroot) %# ' \
    PATH=/system/binaries:/system/systembinaries \
    /bin/oksh -l
```

The login shell reads `/system/configuration/profile`, which is linked to the build variable profile created in the previous section.

Quick verification

```
pwd
echo "$PATH"
```

Command Explanations

- `chroot "$LBI_ROOT"`: Changes the apparent root directory to the target tree.
- `/system/binaries/env -i`: Starts with an empty environment so host variables do not leak into the target shell.
- `HOME`, `TERM`, `SHELL`, `PS1`, and `PATH`: Rebuild the minimal environment needed for an interactive target shell.
- `/bin/oksh -l`: Starts oksh as a login shell so `/system/configuration/profile` is read.
- `pwd`: Confirms that the shell is operating from the target root.
- `echo "$PATH"`: Confirms that target binary directories are the active command search path.

7.6 Create essential system files

7.6. Create essential system files

Create baseline compatibility links, host identity files, account databases, temporary directories, and initial log files inside chroot.

Open standalone page

> **Goal:** create `mtab`, `hosts`, `passwd`, and `group`, then add a test user, create temporary directories, and initialize core log files with usable default modes. If ownership tools are not yet available in the chroot, defer the `utmp` group assignment until later.

Run these commands as `root` inside `chroot`.

Create `mtab` compatibility link

```
ln -s /system/processes/self/mounts /system/configuration/mtab
```

Create `hosts`

```
printf '127.0.0.1  localhost %s\n::1        localhost\n' "<your_hostname>" > /sy
```

Create `passwd`

```
printf '%s\n' \
"root:x:0:0:root:/system/charlie:/bin/oksh" \
"bin:x:1:1:bin:/dev/null:/system/binaries/false" \
"daemon:x:6:6:Daemon User:/dev/null:/system/binaries/false" \
"messagebus:x:18:18:D-Bus Message Daemon User:/run/dbus:/system/binaries/false" 
"uuidd:x:80:80:UUID Generation Daemon User:/dev/null:/system/binaries/false" \
"nobody:x:65534:65534:Unprivileged User:/dev/null:/system/binaries/false" \
> /system/configuration/passwd
```

Create `group`

```
printf '%s\n' \
"root:x:0:" \
"bin:x:1:daemon" \
"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:" \
"clock:x:14:" \
"cdrom:x:15:" \
"adm:x:16:" \
"messagebus:x:18:" \
"input:x:24:" \
```

```
"mail:x:34:" \
"kvm:x:61:" \
"uuidd:x:80:" \
"wheel:x:97:" \
"users:x:999:" \
"nogroup:x:65534:" \
> /system/configuration/group
```

Add test user and create its home

```
echo "tester:x:101:101::/system/users/tester:/bin/oksh" >> /system/configuration
echo "tester:x:101:" >> /system/configuration/group
mkdir -p /system/users/tester
```

Create temporary directories

```
install -d -m 1777 /tmp /system/variable/tmp
```

Initialize core log files

```
mkdir -p /system/variable/log

touch /system/variable/log/btmp \
      /system/variable/log/lastlog \
      /system/variable/log/faillog \
      /system/variable/log/wtmp

chmod 664 /system/variable/log/lastlog
chmod 600 /system/variable/log/btmp
```

Quick verification

```
ls -l /system/configuration/hosts \
      /system/configuration/passwd \
      /system/configuration/group \
      /system/configuration/mtab

grep '^tester:' /system/configuration/passwd /system/configuration/group

ls -ld /tmp /system/variable/tmp
```

```
ls -l /system/variable/log/btmp \
      /system/variable/log/lastlog \
      /system/variable/log/faillog \
      /system/variable/log/wtmp
```

Command Explanations

- `ln -s /system/processes/self/mounts /system/configuration/mtab`: Provides an `mtab` compatibility path backed by the kernel's live mount table.
- `printf ... > /system/configuration/hosts`: Writes a minimal hosts file with localhost and the chosen hostname.
- `printf ... > /system/configuration/passwd`: Creates the base user account database for root and system users.
- `printf ... > /system/configuration/group`: Creates the base group database with standard administrative and device groups.
- `echo "tester:..." >> ...`: Adds the optional tester user and group entries for non-root checks.
- `mkdir -p /system/users/tester`: Creates the tester account home directory.
- `install -d -m 1777 /tmp /system/variable/tmp`: Creates the root temporary directory and the backing directory for `/var/tmp` with world-writable sticky permissions.
- `mkdir -p /system/variable/log` and `touch ...`: Creates the initial login/accounting log files expected by user-management tools.
- `chmod 664` and `chmod 600`: Set log-file permissions so public status logs and private login-failure logs have appropriate access.
- `ls -l` and `grep`: Verify that the expected configuration and account files exist after creation.

7.7. gettext-tiny 0.3.3

gettext-tiny provides lightweight gettext tooling for builds in chroot, including msgfmt and a musl-compatible libintl flavor.

Open standalone page

> **Input assumption:** `gettext-tiny-0.3.3.tar.xz` is already present in `/sources` from the chapter 4 source staging step.

Licenses:

- MIT

Dependencies:

- musl (libc)
- make
- clang/llvm toolchain

gettext-tiny is a lightweight replacement for key GNU gettext tools and libintl compatibility pieces. we need it to provide `msgfmt` and musl-friendly `libintl` compatibility for package builds inside chapter 7 chroot.

Extract and Enter the Source Tree

```
cd /sources
tar -xf gettext-tiny-0.3.3.tar.xz
cd gettext-tiny-0.3.3
```

Patch the install symlink rule for BSD install

The upstream install rule uses `install -l` for symlinks, but BSD `install` does not support that option. Replace that line with a normal directory creation plus `ln -sf`.

```
awk '
/^[[:space:]]*\$\(INSTALL\) -D -l / {
    print "\tmkdir -p $(patsubst %m4/,%,$(dir $@))"
    print "\tln -sf ../$(subst $(datarootdir)/,,$(datadir))/$< $(patsubst %m4/,%
    next
}
{ print }
' Makefile > Makefile.new
mv Makefile.new Makefile
```

Build gettext-tiny

```
make $LWI_MAKE_FLAGS \
    LIBINTL=musl \
    CPPFLAGS="-I/system/headers" \
    CFLAGS="-I/system/headers" \
    LDFLAGS="-L/system/libraries" \
    CC="cc -B/system/libraries -B/system/libraries/clang/22/lib/linux"
```

Install gettext-tiny

```
make $LWI_MAKE_FLAGS \
    LIBINTL=musl \
    CPPFLAGS="-I/system/headers" \
    CFLAGS="-I/system/headers" \
    LDFLAGS="-L/system/libraries" \
    CC="cc -B/system/libraries -B/system/libraries/clang/22/lib/linux" \
    prefix=/system \
    bindir=/system/binaries \
    includedir=/system/headers \
    libdir=/system/libraries \
    datarootdir=/system/documentation \
    datadir=/system/documentation/gettext-tiny \
    acdir=/system/documentation/aclocal \
    install
```

Command Explanations

- `cd /sources`, `tar -xf`, and `cd gettext-tiny-...`: Enter a fresh gettext-tiny source tree from the staged archive.
- `awk ... Makefile > Makefile.new`: Replaces the upstream `install -l` symlink rule with portable `mkdir` plus `ln -sf` commands.
- `mv Makefile.new Makefile`: Installs the rewritten Makefile after the replacement succeeds.
- `make $LWI_MAKE_FLAGS`: Builds with the shared parallel make policy.
- `LIBINTL=musl`: Uses musl's libc-provided gettext stubs instead of building a separate libintl.
- `CPPFLAGS`, `CFLAGS`, and `LDFLAGS`: Point the build at the target headers and libraries.
- `CC="cc -B..."`: Tells the compiler where to find target startup objects and Clang runtime files.
- `make ... install`: Installs gettext-tiny with explicit `/system` binary, library, header, and data paths.

7.8 byacc 20260126

7.8. byacc 20260126

byacc provides a yacc-compatible LALR(1) parser generator for package builds in the chroot environment.

Open standalone page

> **Input assumption:** `byacc-20260126.tgz` is already present in `/sources` from the chapter 4 source staging step.

Licenses:

- BSD-3-Clause

Dependencies:

- musl (libc)
- make
- clang/llvm toolchain

byacc is a portable Berkeley yacc-compatible parser generator. we need it to provide a broadly compatible `yacc` command for package builds that expect traditional yacc behavior.

Extract and Enter the Source Tree

```
cd /sources
tar -xf byacc-20260126.tgz
cd byacc-20260126
```

Configure byacc

```
./configure \
    --prefix=/system \
    --bindir=/system/binaries \
    --mandir=/system/documentation/man-pages
```

Build byacc

```
make $LWI_MAKE_FLAGS
```

Install byacc

```
make install
```

Command Explanations

- `cd /sources`, `tar -xf`, and `cd byacc-...`: Enter the byacc source tree from the staged archive.
- `./configure --prefix=/system ...`: Configures byacc to install commands and manual pages into the target `/system` layout.
- `make $LWI_MAKE_FLAGS`: Builds byacc with the shared make parallelism setting.
- `make install`: Installs byacc into the active target filesystem from inside the chroot.

7.9 python 3.14.4

7.9. python 3.14.4

python provides the Python 3 runtime and standard library for build tooling and scripts inside chroot.

Open standalone page

> **Input assumption:** `Python-3.14.4.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://www.python.org/ftp/python/3.14.4/Python-3.14.4.tar.xz`

Licenses:

- Python-2.0

Dependencies:

- musl (libc)
- make

python is a general-purpose programming language and runtime. we need it to provide `python3` and the standard library for build scripts and tooling in later chapters.

Extract and Enter the Source Tree

```
cd /sources
tar -xf Python-3.14.4.tar.xz
cd Python-3.14.4
```

Configure python

> **Build note:** no OpenSSL-compatible library is installed at this stage, so this Python build does not provide the optional `_ssl` module.

```
ax_cv_c_float_words_bigendian=no \
lbi_configure \
    --enable-shared \
    --without-ensurepip \
    --without-static-libpython \
    --with-tzpath= \
    --with-platlibdir=libraries
```

Build python

```
make $LWI_MAKE_FLAGS
```

Install python

```
make install
```

Command Explanations

- `cd /sources`, `tar -xf`, and `cd Python-...`: Enter the Python source tree from the staged archive.
- `ax_cv_c_float_words_bigendian=no`: Preanswers Python's float word-order configure probe for the target.
- `lbi_configure`: Applies the book's `/system` install layout to Python's configure step.
- `--enable-shared`: Builds a shared `libpython` for extension modules and embedding users.
- `--without-ensurepip`: Skips bundled pip installation in this bootstrap Python.
- `--without-static-libpython`: Avoids installing the static Python library.
- `--with-tzpath=` and `--with-platlibdir=libraries`: Match the book's timezone and library-directory policy.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install Python into the target environment.

7.10. ubase git snapshot

ubase provides Linux-specific base utilities for the chroot environment.

Open standalone page

> **Input assumption:** `ubase-e8249b49ca3e.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source note:** this archive is generated by `scripts/fetch-sources.sh` from upstream `https://git.suckless.org/ubase` at commit `e8249b49ca3e02032dece5e0cdac3d236667a6d9`.

Licenses:

- MIT/X Consortium-style license

Dependencies:

- musl (libc)
- make
- clang/llvm toolchain

ubase is a Linux-specific base utility collection from suckless. we need it to provide tools such as `mount`, `umount`, `ps`, `stat`, `swapon`, `switch_root`, and related Linux system commands inside the target environment.

Extract and Enter the Source Tree

```
cd /sources
tar -xf ubase-e8249b49ca3e.tar.gz
cd ubase-e8249b49ca3e
```

Build ubase

> **Build note:** upstream ubase links `-lcrypt` by default. musl provides the needed `crypt` interface from libc, so this target build clears `LDLIBS`.

```
make $LWI_MAKE_FLAGS \
    CC=cc \
    AR=ar \
    RANLIB=ranlib \
    CFLAGS="-std=c99 -Wall -Wextra $LWI_CFLAGS" \
```

```
    LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
    LDLIBS=
```

Install ubase

```
make install \
    DESTDIR= \
    PREFIX=/system \
    MANPREFIX=/system/documentation/man-pages \
    CFLAGS="-std=c99 -Wall -Wextra $LWI_CFLAGS" \
    LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
    LDLIBS=

install -d /system/binaries

if [ -d /system/bin ]; then
    mv -f /system/bin/* /system/binaries/
    rmdir /system/bin
fi
```

Command Explanations

- `cd /sources`, `tar -xf`, and `cd ubase-...`: Enter the staged ubase snapshot source tree.
- `make $LWI_MAKE_FLAGS`: Builds ubase with the shared parallel make setting.
- `CC=cc`, `AR=ar`, and `RANLIB=ranlib`: Use the target toolchain now active inside the chroot.
- `CFLAGS="-std=c99 -Wall -Wextra $LWI_CFLAGS"`: Keeps upstream warning settings while preserving local C flag tuning.
- `LDFLAGS="$LBI_CUSTOM_LDFLAGS"` and `LDLIBS=`: Apply local linker tuning and avoid extra default libraries.
- `make install PREFIX=/system MANPREFIX=...`: Installs ubase commands and manual pages into the book's layout.
- `install -d /system/binaries`: Ensures the final binary directory exists for any later command moves or links.

7.11 Cleanup

7.11. Cleanup

Remove temporary bootstrap artifacts and selected documentation so the target system does not carry unnecessary build leftovers.

Open standalone page

> **Run context:** execute these commands as `root` inside the chapter 7 `chroot` environment.
>
> **Scope:** this step intentionally deletes transitional files and directories that were useful during the build but are not part of the intended runtime system layout.

At this point, the base utilities are installed and the temporary scaffolding can be trimmed. This cleanup keeps the final image smaller and avoids carrying artifacts that can confuse later builds.

Remove selected documentation directories

The following documentation paths are not kept in the target layout policy for this book:

- `/system/documentation/man-pages`
- `/system/documentation/xz`

Remove both directories now:

```
rm -rf /system/documentation/man-pages /system/documentation/xz
```

Remove libtool archive files from system libraries

Libtool archive files (`*.la`) are metadata files used primarily during some build/link workflows. They are not needed for the target runtime and can cause incorrect link decisions in some toolchains.

Remove them from `/system/libraries`:

```
find /system/libraries -type f -name '*.la' -exec rm -f -- '{}' ';'
```

Remove temporary tools tree

The `/system/tools` directory was useful during transitional build phases and is no longer required.

```
rm -rf /system/tools
```

Cleanup Result

After this step, the target system keeps only the paths intended for ongoing package work and runtime use, with temporary bootstrap content removed.

Command Explanations

- `rm -rf /system/documentation/man-pages /system/documentation/xz`: Removes temporary documentation when the target image should be kept smaller.
- `find /system/libraries -type f -name '*.la' -exec rm -f -- '{}' ';'`: Deletes libtool archive files that can encode stale build-time paths.
- `rm -rf /system/tools`: Removes the temporary tools prefix after the target toolchain and runtime are self-hosted.

8.1. Introduction

Chapter 8 builds the remaining system packages natively inside the target environment and prepares the system for daily use.

Open standalone page

> **Goal:** finish the userland package set from inside the target system, with consistent layout and predictable behavior for normal operation.

Chapter 7 established a working `chroot` environment with enough tooling to continue package builds from within the target tree. Chapter 8 uses that environment to complete the remaining package work and move from a minimal base to a practical system.

This chapter is about completion and consistency: adding the packages that turn the current environment into a full day-to-day userspace while preserving the directory policy and build conventions used throughout the book.

Expected Outcome

By the end of chapter 8, the target system should have:

- the remaining essential package set installed in the intended layout;
- a coherent runtime environment that does not depend on transitional bootstrap paths;
- a stable baseline suitable for post-build configuration and ongoing maintenance.

Scope Notes

Keep chapter 8 focused on package completion and system usability. Optional customization and local preference tuning can happen afterward, once the core build is done and verified.

8.2. iana-etc 20260409

iana-etc provides current network services and protocol name tables for `/system/configuration`.

Open standalone page

> **Input assumption:** `iana-etc-20260409.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/Mic92/iana-etc/releases/download/20260409/iana-etc-20260409.tar.gz`

Licenses:

- MIT

Dependencies:

- musl (libc)

iana-etc is a generated dataset of current IANA service and protocol assignments. we need it to provide `services` and `protocols` entries used by the target system's network configuration files.

Extract and Enter the Source Tree

```
cd /sources
tar -xf iana-etc-20260409.tar.gz
cd iana-etc-20260409
```

Install iana-etc data files

```
cp services protocols /system/configuration
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to reuse them.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd iana-etc-...`: Enter the staged iana-etc source tree.
- `cp services protocols /system/configuration`: Installs the network service and protocol registries used by libc and network tools.

8.3. musl libc final pass 1.2.6

The final musl pass rebuilds and reinstalls libc with mimalloc integration from inside the target environment.

Open standalone page

> **Input assumption:** `musl-1.2.6.tar.gz`, `mimalloc-v3.3.0.tar.gz`, `musl-1.2.6-mimalloc.patch`, `mimalloc-3.3.0-for-musl.patch`, and `musl-1.2.6-runtime-lib-from-compiler.patch` are already present in `/sources`.
>
> **Source URLs:** `https://musl.libc.org/releases/musl-1.2.6.tar.gz` and `https://github.com/microsoft/mimalloc/archive/refs/tags/v3.3.0.tar.gz`

Licenses:

- MIT (musl)
- MIT (mimalloc)

Dependencies:

- make
- clang/llvm toolchain

musl is a lightweight C standard library implementation for Linux systems. we need it to provide the final installed libc used by the completed chapter 8 target environment.

mimalloc is a general-purpose memory allocator library. we need it to provide the allocator sources integrated into this final musl pass.

Extract musl and Apply the First Patch

```
cd /sources
tar -xf musl-1.2.6.tar.gz
cd musl-1.2.6
patch -Np1 -i ../musl-1.2.6-mimalloc.patch
```

Add mimalloc Upstream Sources (`src` and `include`)

```
mkdir -p src/malloc/mimalloc/upstream
tar -xf ../mimalloc-v3.3.0.tar.gz \
    --strip-components=1 \
```

```
    -C src/malloc/mimalloc/upstream \
    mimalloc-3.3.0/src mimalloc-3.3.0/include

# The second patch updates this file in-place.
cp -v src/malloc/mimalloc/upstream/src/static.c src/malloc/mimalloc/static.c
```

Apply the Second Patch

```
patch -Np1 -i ../mimalloc-3.3.0-for-musl.patch
```

Apply the compiler-rt builtins runtime patch

```
patch -Np1 -i ../musl-1.2.6-runtime-lib-from-compiler.patch
```

Configure, Build, and Install musl (Final Pass)

```
lbi_configure --with-malloc=mimalloc
make $LWI_MAKE_FLAGS
make install
```

WARNING: temporary command breakage can happen

> **WARNING:** this step **MAY** temporarily remove your ability to run commands in `chroot` on some systems.
>
> Do **not** panic.
>
> Exit the `chroot`, ensure `LBI_ROOT` is set, and as `root` run:
>
> ```
> ln -snf ./libc.so \
>     "$LBI_ROOT/system/libraries/ld-musl-${LBI_ARCH}.so.1"
> chmod 755 "$LBI_ROOT/system/libraries/libc.so"
> ```
>
> Then re-enter `chroot` and continue.

Quick verification

```
ls -lh /system/libraries/libc.so \
       "/system/libraries/ld-musl-${LBI_ARCH}.so.1"
```

> After this step is complete, you can remove the extracted source directories and source tarballs from `/sources` if you do not plan to rebuild musl again.

Command Explanations

- `tar -xf musl-1.2.6.tar.gz` and `cd musl-...`: Unpack and enter a clean musl source tree.
- `patch -Np1 -i ../musl-1.2.6-mimalloc.patch`: Applies the local musl integration patch using paths relative to the source root.
- `mkdir -p src/malloc/mimalloc/upstream`: Creates the destination for upstream mimalloc sources inside musl.
- `tar -xf ../mimalloc-v3.3.0.tar.gz --strip-components=1 ...`: Copies only the mimalloc source and include trees needed by the allocator integration.
- `cp -v ... static.c`: Places the mimalloc static allocator entry point where the musl patch expects it.
- `patch -Np1 -i ../mimalloc-...` and `patch -Np1 -i ../musl-...runtime...`: Apply the remaining allocator and compiler-rt runtime patches.
- `lbi_configure --with-malloc=mimalloc`: Configures musl with the book's layout and selects mimalloc as the allocator.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install the final musl pass.
- `ln -snf ./libc.so ... ld-musl-${LBI_ARCH}.so.1`: Provides the musl dynamic-loader soname expected by dynamically linked programs.
- `chmod 755 ... libc.so`: Ensures the dynamic loader/library is executable.

8.4 pigz stage 2 2.8

8.4. pigz stage 2 2.8

Rebuild pigz in the final target environment and provide gzip and bzip2 command-name compatibility links.

Open standalone page

> **Input assumption:** `pigz-2.8.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/madler/pigz/archive/refs/tags/v2.8.tar.gz`

Licenses:

- zlib License

Dependencies:

- musl (libc)
- make

pigz is a parallel gzip-compatible compressor and decompressor. we need it to provide final stage compression tools in chapter 8 and compatibility command names for existing workflows.

Extract and Enter the Source Tree

```
cd /sources
tar -xf pigz-2.8.tar.gz
cd pigz-2.8
```

Build pigz

```
make $LWI_MAKE_FLAGS CC=cc
```

Install pigz

```
install -Dm755 pigz /system/binaries/pigz
install -Dm755 unpigz /system/binaries/unpigz
install -Dm644 pigz.1 /system/documentation/man-pages/man1/pigz.1
```

Install compatibility symlinks

```
ln -sf pigz /system/binaries/gzip
ln -sf unpigz /system/binaries/gunzip
ln -sf unpigz /system/binaries/zcat

ln -sf pigz /system/binaries/bzip2
ln -sf unpigz /system/binaries/bunzip2
ln -sf unpigz /system/binaries/bzcat
```

> The `bzip2`-family links above provide command-name compatibility to pigz/unpigz in this layout.

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild pigz again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd pigz-...`: Enter the staged pigz source tree.
- `make $LWI_MAKE_FLAGS CC=cc`: Builds pigz with the target C compiler and shared make parallelism.
- `install -Dm755 pigz` and `install -Dm755 unpigz`: Install compressor and decompressor commands, creating directories as needed.
- `install -Dm644 pigz.1`: Installs the manual page.
- `ln -sf pigz ... gzip` and `ln -sf unpigz ...`: Provide gzip and bzip2 family compatibility command names.

8.5 xz stage 2 5.8.3

8.5. xz stage 2 5.8.3

Rebuild xz in the final target environment using the autotools configure path via `lbi_configure`.

Open standalone page

> **Input assumption:** `xz-5.8.3.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/tukaani-project/xz/releases/download/v5.8.3/xz-5.8.3.tar.xz`
>
> **Upstream build note:** the upstream `INSTALL` document describes both configure and CMake options for this release. This section uses the autotools `configure` flow through `lbi_configure`.

Licenses:

- oBSD

Dependencies:

- musl (libc)
- make

xz is a compression toolkit that provides liblzma and .xz/.lzma command-line tools. we need it to provide final stage `xz` and `liblzma` tooling in chapter 8.

Extract and Enter the Source Tree

```
cd /sources
tar -xf xz-5.8.3.tar.xz
cd xz-5.8.3
```

Configure xz

> If configure fails with a `CFLAGS contains something that makes -Werror complain` error, use the upstream-documented override below.

```
SKIP_WERROR_CHECK=yes \
lbi_configure \
    --disable-static \
    --enable-shared \
    --disable-nls \
    --docdir=/system/documentation/xz
```

Build xz

```
make $LWI_MAKE_FLAGS
```

Install xz

```
make install
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild xz again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd xz-...`: Enter the staged xz source tree.
- `SKIP_WERROR_CHECK=yes`: Prevents configure from rejecting builds where warnings are not treated as fatal.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--disable-static` and `--enable-shared`: Install shared libraries without static archives.
- `--disable-nls`: Avoids gettext/native language support for this pass.
- `--docdir=/system/documentation/xz`: Places xz documentation in the book's documentation tree.

- `make $LWI_MAKE_FLAGS` and `make install`: Build and install xz in the target environment.

8.6 zstd stage 2 1.5.7

8.6. zstd stage 2 1.5.7

Build and install zstd in the final target environment so zstd compression libraries and tools are available to later packages.

Open standalone page

> **Input assumption:** `zstd-1.5.7.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/facebook/zstd/releases/download/v1.5.7/zstd-1.5.7.tar.gz`
>
> **Upstream build note:** upstream documents GNU make as the maintained build system and supports staged install variables such as `PREFIX`, `BINDIR`, `LIBDIR`, `INCLUDEDIR`, and `PKGCONFIGDIR`.

Licenses:

- BSD-3-Clause

Dependencies:

- musl (libc)
- clang
- make
- zlib-ng
- xz

zstd is a lossless compression library and command-line tool. we need it to provide `libzstd` and the `zstd` utilities used by LLVM and other target packages.

Extract and Enter the Source Tree

```
cd /sources
rm -rf zstd-1.5.7
tar -xf zstd-1.5.7.tar.gz
cd zstd-1.5.7
```

Build zstd

```
CC=clang \
CFLAGS="$LWI_CFLAGS" \
LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
make $LWI_MAKE_FLAGS
```

Install zstd

```
CC=clang \
CFLAGS="$LWI_CFLAGS" \
LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
make install \
    PREFIX=/system \
    BINDIR=/system/binaries \
    LIBDIR=/system/libraries \
    INCLUDEDIR=/system/headers \
    MANDIR=/system/documentation/man-pages \
    PKGCONFIGDIR=/system/libraries/pkgconfig
```

Verify zstd

```
zstd --version
ls /system/binaries/zstd
```

Command Explanations

- `rm -rf zstd-1.5.7`: Removes any previous extracted source tree before rebuilding.
- `tar -xf zstd-1.5.7.tar.gz`: Extracts the staged zstd source archive.
- `CC=clang`: Builds zstd with the target Clang compiler.
- `CFLAGS` and `LDFLAGS`: Apply local compile and link tuning through the environment so zstd's Makefiles can still append required flags such as `-fPIC`, `-shared`, and `-pthread`.
- `make $LWI_MAKE_FLAGS`: Builds zstd with the shared make parallelism setting.
- `make install PREFIX=/system ...`: Installs commands, libraries, headers, man pages, and pkg-config metadata into the book's layout.
- `zstd --version` and `ls /system/binaries/zstd`: Verify that the command and library are available in the correct locations.

8.7. file stage 2 5.47

Rebuild file in the final target environment so file-type detection and libmagic use the finalized userspace.

Open standalone page

> **Input assumption:** `file-5.47.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://mirrors.mit.edu/macports/distfiles/file/file-5.47.tar.gz`

Licenses:

- BSD-2-Clause

Dependencies:

- musl (libc)
- xz (for compressed magic data files)
- zlib-ng (for compressed magic data files)
- make

file is a file type identification utility and library. we need it to provide final stage `file` and `libmagic` behavior in the completed chapter 8 userspace.

Extract and Enter the Source Tree

```
cd /sources
tar -xf file-5.47.tar.gz
cd file-5.47
```

Configure file

```
lbi_configure
```

Build file

```
make $LWI_MAKE_FLAGS
```

Install file

```
make install
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild file again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd file-...`: Enter the staged file source tree.
- `lbi_configure`: Configures file for the book's layout.
- `make $LWI_MAKE_FLAGS`: Builds with shared make parallelism.
- `make install`: Installs into the active target filesystem.

8.8 bc 7.0.3

8.8. bc 7.0.3

Build and install bc in the final target environment for standards-compatible calculator functionality.

Open standalone page

> **Input assumption:** `bc-7.0.3.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/gavinhoward/bc/releases/download/7.0.3/bc-7.0.3.tar.xz`
>
> **Upstream build note:** this package uses its own `configure.sh` build system and upstream documents `./configure.sh` + `make` + `make install` for POSIX-compatible systems.

Licenses:

- BSD-2-Clause

Dependencies:

- musl (libc)
- make

bc is a POSIX arbitrary-precision calculator implementation with integrated dc mode. we need it to provide a standards-compatible calculator for scripts and interactive numeric work in chapter 8.

Extract and Enter the Source Tree

```
cd /sources
tar -xf bc-7.0.3.tar.xz
cd bc-7.0.3
```

Configure bc

> `bc` does not use autotools `configure`, so `lbi_configure` is not applicable here.

```
CC="cc -std=c99" ./configure.sh \
    --prefix=/system \
    --bindir=/system/binaries \
    --libdir=/system/libraries \
    --datarootdir=/system/documentation \
    --datadir=/system/documentation \
    --mandir=/system/documentation/man-pages \
    --man1dir=/system/documentation/man-pages/man1 \
    --man3dir=/system/documentation/man-pages/man3 \
    --disable-nls
```

Build bc

```
make $LWI_MAKE_FLAGS
```

Install bc

```
make install
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild bc again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd bc-...`: Enter the staged bc source tree.
- `CC="cc -std=c99" ./configure.sh`: Runs bc's custom configure script with a C99 compiler mode.

- `--prefix`, `--bindir`, `--libdir`, and documentation directories: Install bc into the book's `/system` layout.
- `--disable-nls`: Skips native language support for this target build.
- `make $LWI_MAKE_FLAGS`: Builds with shared make parallelism.
- `make install`: Installs bc and dc into the target filesystem.

8.9. pkgconf 2.5.1

Build and install pkgconf in the final target environment to provide pkg-config-compatible dependency queries.

Open standalone page

> **Input assumption:** `pkgconf-2.5.1.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://distfiles.ariadne.space/pkgconf/pkgconf-2.5.1.tar.xz`
>
> **Upstream build note:** upstream documents pkgconf as an autotools-based UNIX build (`./configure`, `make`, `make install`) for release tarballs.

Licenses:

- ISC

Dependencies:

- musl (libc)
- make

pkgconf is a compiler and linker flag resolver for pkg-config metadata files. we need it to provide pkg-config-compatible dependency resolution for later package builds in chapter 8.

Extract and Enter the Source Tree

```
cd /sources
tar -xf pkgconf-2.5.1.tar.xz
cd pkgconf-2.5.1
```

Configure pkgconf

```
lbi_configure \
    --with-system-libdir=/system/libraries \
    --with-system-includedir=/system/headers
```

Build pkgconf

```
make $LWI_MAKE_FLAGS
```

Install pkgconf

```
make install
```

Install pkg-config compatibility symlink

```
ln -sf pkgconf /system/binaries/pkg-config
ln -sf pkgconf.1 /system/documentation/man-pages/man1/pkg-config.1
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild pkgconf again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd pkgconf-...`: Enter the staged pkgconf source tree.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--with-system-libdir=/system/libraries` and `--with-system-includedir=/system/headers`: Teach pkgconf the target's default library and header locations.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install pkgconf.
- `ln -sf pkgconf ... pkg-config`: Provides the common `pkg-config` command name and manual-page alias.

8.10 Shadow 4.19.4

8.10. Shadow 4.19.4

Build and install Shadow in the final target environment to provide user and group account management tools.

Open standalone page

> **Input assumption:** `shadow-4.19.4.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/shadow-maint/shadow/releases/download/4.19.4/shadow-4.19.4.tar.xz`

Licenses:

- BSD 3-Clause

Dependencies:

- musl (libc)
- make

Shadow provides the core user and group account management programs and supporting files. we need it in chapter 8 to create and manage system accounts in the final target environment.

Extract and Enter the Source Tree

```
cd /sources
tar -xf shadow-4.19.4.tar.xz
cd shadow-4.19.4
```

Configure Shadow

```
lbi_configure \
    --disable-static \
    --with-bcrypt \
    --with-yescrypt \
    --without-su \
    --without-libbsd \
    --disable-logind \
    --with-group-name-max-length=32
```

Set BSD-only install lists

> **License policy note:** Shadow 4.19.4 has an upstream `--without-su` switch for `su`. The `vipw`/`vigr` utility is always present in the default install list and is `GPL-2.0-or-later`, so this section uses Automake program and man-page allow-lists to leave it out.

```
LBI_SHADOW_USBINS="\
chgpasswd \
chpasswd \
groupadd \
groupdel \
groupmems \
groupmod \
grpck \
grpconv \
grpunconv \
logoutd \
newusers \
pwck \
pwconv \
pwunconv \
useradd \
userdel \
usermod"

LBI_SHADOW_MANS="\
man1/chage.1 \
man1/chfn.1 \
man8/chgpasswd.8 \
man8/chpasswd.8 \
man1/chsh.1 \
man1/expiry.1 \
man5/faillog.5 \
man8/faillog.8 \
man3/getspnam.3 \
man1/gpasswd.1 \
man8/groupadd.8 \
man8/groupdel.8 \
man8/groupmems.8 \
man8/groupmod.8 \
man8/grpck.8 \
man8/grpconv.8 \
man8/grpunconv.8 \
man5/gshadow.5 \
man1/login.1 \
man5/login.defs.5 \
man8/logoutd.8 \
```

```
man1/newgrp.1 \
man8/newusers.8 \
man8/nologin.8 \
man1/passwd.1 \
man5/passwd.5 \
man8/pwck.8 \
man8/pwconv.8 \
man8/pwunconv.8 \
man1/sg.1 \
man3/shadow.3 \
man5/shadow.5 \
man8/useradd.8 \
man8/userdel.8 \
man8/usermod.8 \
man1/getsubids.1 \
man1/newgidmap.1 \
man1/newuidmap.1 \
man5/subgid.5 \
man5/subuid.5"

export LBI_SHADOW_USBINS LBI_SHADOW_MANS
```

Post-Configure Adjustments

```
sed -i '' -E \
    -e 's@^[[:space:]]*#?[[:space:]]*ENCRYPT_METHOD[[:space:]]+.*@ENCRYPT_METHOD
    -e 's@/var/spool/mail@/var/mail@g' \
    -e '/^[[:space:]]*(ENV_SUPATH|ENV_PATH|ENV_ROOTPATH|PATH)=/ {
        s@/system/systembinaries:@@g
        s@:/system/systembinaries@@g
        s@/system/binaries:@@g
        s@:/system/binaries@@g
    }' \
    etc/login.defs
```

Build Shadow

```
make $LWI_MAKE_FLAGS \
    bin_PROGRAMS=login \
    usbin_PROGRAMS="$LBI_SHADOW_USBINS"
```

Install Shadow

```
make install \
    bindir=/system/binaries \
    ubindir=/system/binaries \
    sbindir=/system/systembinaries \
    usbindir=/system/systembinaries \
    bin_PROGRAMS=login \
    usbin_PROGRAMS="$LBI_SHADOW_USBINS"

make -C man install-man \
    man_MANS="$LBI_SHADOW_MANS"
```

Post-Install Setup

```
pwconv
grpconv
mkdir -p /etc/default
useradd -D --gid 999
passwd root
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild Shadow again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd shadow-...`: Enter the staged Shadow source tree.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--with-bcrypt` and `--with-yescrypt`: Enable modern password hashing methods.
- `--without-su`: Prevents Shadow from building or installing `su`, whose source contains GNU `su` ancestry.
- `--without-libbsd` and `--disable-logind`: Avoid optional dependencies not required here.
- `LBI_SHADOW_USBINS`: Defines the installed system-binary allow-list without `vipw`.
- `LBI_SHADOW_MANS`: Defines the installed man-page allow-list without `vipw.8`, `vigr.8`, or `su.1`.
- `sed -i ... etc/login.defs`: Sets YESCRYPT as the default hash, adjusts mail paths, and removes non-book binary directories from default PATH settings.
- `make $LWI_MAKE_FLAGS bin_PROGRAMS=login usbin_PROGRAMS=...`: Builds Shadow with shared make parallelism while keeping `su` and `vipw` out of the program list.

- `make exec_prefix=/system install ...` and `make -C man install-man ...`: Install only the allowed commands and manual pages into the target layout.
- `mv /system/sbin/* /system/systembinaries/`: Moves administrative binaries into the book's system-binary directory.
- `pwconv` and `grpconv`: Create shadow password and group databases from the existing passwd/group files.
- `useradd -D --gid 999` and `passwd root`: Set default useradd policy and assign the root password.

8.11. ncurses stage 2 6.6-20260418

Rebuild ncurses in the final target environment so curses libraries, terminfo data, and package metadata match the completed userspace.

Open standalone page

> **Input assumption:** `ncurses-6.6-20260418.tgz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://invisible-mirror.net/archives/ncurses/current/ncurses-6.6-20260418.tgz`
>
> **Upstream build note:** the upstream `INSTALL` file and `./configure --help` document the standard `./configure`, `make`, and `make install` flow, plus the wide-character, shared-library, C++ binding, manpage-format, stripping, and pkg-config file switches used below.

Licenses:

- X11-style (ncurses)

Dependencies:

- musl (libc)
- awk
- make
- pkgconf

ncurses is a terminal handling library and terminfo toolkit. we need it to provide final target curses interfaces, terminal capability data, and pkg-config metadata for later software.

Extract and Enter the Source Tree

Remove any previous extracted ncurses tree before unpacking. ncurses generates build-time table sources such as `ncurses/comp_captab.c` and `include/hashsize.h`; stale copies from an earlier failed or cross build can be newer than the unpacked sources and break the final build.

```
cd /sources
rm -rf ncurses-6.6-20260418
tar -xf ncurses-6.6-20260418.tgz
cd ncurses-6.6-20260418
```

Configure ncurses

```
lbi_configure \
    --with-manpage-format=normal \
    --with-shared \
    --without-normal \
    --without-cxx-binding \
    --without-debug \
    --without-ada \
    --enable-widec \
    --disable-stripping \
    --enable-pc-files \
    --with-pkg-config-libdir=/system/libraries/pkgconfig \
    AWK=awk
```

Pre-generate ncurses capability tables

ncurses generates `include/hashsize.h` from the capability tables in `include/Caps` and `include/Caps-ncurses`. In this target environment, the `make` frontend can mis-generate that file as a zero-entry table, which then creates an empty `ncurses/comp_captab.c` and fails with `term.h and comp_captab.c disagree`.

Replace the hash-size generator with an equivalent `awk` implementation, then pre-generate the capability tables before the full build. This keeps the main build from compiling a zero-entry `comp_captab.c`.

```
cat > include/MKhashsize.sh <<'EOF'
#!/bin/sh
echo "/*"
echo " * hashsize.h -- hash and token table constants"
echo " */"
```

```
awk '
    /^[ #]/ { next }
    /^$/ { next }
    /^capalias/ { next }
    /^infoalias/ { next }
    /^userdef/ { next }
    /^used_by/ { next }
    { ++n }
    END {
        print ""
        printf "#define CAPTABSIZE\t%d\n", n
        printf "#define HASHTABSIZE\t(%d * 2)\n", n
    }
' "$@"
EOF
chmod +x include/MKhashsize.sh

sh include/MKhashsize.sh include/Caps include/Caps-ncurses > include/hashsize.h

captabsize=$(sed -n 's/^#define[[:space:]]*CAPTABSIZE[[:space:]]*//p' include/ha
test "$captabsize" -gt 0

rm -f ncurses/comp_captab.c ncurses/comp_userdefs.c

make -C include sources
make -C ncurses make_hash

(
    cd ncurses
    sh -e ./tinfo/MKcaptab.sh \
        awk \
        1 \
        ./tinfo/MKcaptab.awk \
        ../include/Caps \
        ../include/Caps-ncurses \
        > comp_captab.c
    sh -e ./tinfo/MKuserdefs.sh \
        awk \
        1 \
        ../include/Caps \
```

```
        ../include/Caps-ncurses \
        > comp_userdefs.c
)
```

Build ncurses

```
make $LWI_MAKE_FLAGS
```

Install ncurses

```
make install
```

Post-install Compatibility Adjustments

```
ln -sf libncursesw.so /system/libraries/libncurses.so
sed -e 's/^#if.*XOPEN.*$/#if 1/' \
    -i '' /system/headers/curses.h
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild ncurses again.

Command Explanations

- `rm -rf ncurses-...`: Removes stale generated files before rebuilding ncurses.
- `lbi_configure`: Configures ncurses with the book's install layout.
- `--with-shared`, `--without-normal`, and `--enable-widec`: Build shared wide-character ncurses libraries for the final target.
- `--without-cxx-binding`, `--without-debug`, and `--without-ada`: Skip optional bindings and debug outputs not needed here.
- `cat > include/MKhashsize.sh`: Replaces a generated helper script so hash table constants are produced consistently in this target build.
- `make -j1`: Builds serially to avoid ncurses generated-file races.
- `make install`: Installs ncurses into the target filesystem.
- `ln -sf libncursesw.so ... libncurses.so`: Provides the conventional ncurses library name.
- `sed ... curses.h`: Exposes X/Open declarations expected by later packages.

8.12. byacc stage 2 20260126

Rebuild byacc in the final target environment so the yacc-compatible parser generator matches the completed userspace.

Open standalone page

> **Input assumption:** `byacc-20260126.tgz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://invisible-island.net/archives/byacc/byacc-20260126.tgz`
>
> **Upstream build note:** upstream provides a `configure` script with standard installation directory options, followed by `make` and `make install`. The generated makefile installs the program as `yacc`.

Licenses:

- Public domain
- Permissive notices for selected build-support files

Dependencies:

- musl (libc)
- make

byacc is a portable Berkeley yacc-compatible parser generator. we need it to provide the final target `yacc` command for package builds that expect traditional yacc behavior.

Extract and Enter the Source Tree

```
cd /sources
rm -rf byacc-20260126
tar -xf byacc-20260126.tgz
cd byacc-20260126
```

Configure byacc

```
lbi_configure --with-manpage-format=normal
```

Build byacc

```
make $LWI_MAKE_FLAGS
```

Install byacc

```
make install
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild byacc again.

Command Explanations

- `rm -rf byacc-...`: Removes any previous byacc source tree before rebuilding.
- `tar -xf byacc-20260126.tgz`: Extracts the staged byacc archive.
- `lbi_configure --with-manpage-format=normal`: Configures byacc for the book's layout with normal man pages.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install byacc in the final target environment.

8.13 bsdgrep stage 2 master snapshot

8.13. bsdgrep stage 2 master snapshot

Rebuild bsdgrep in the final target environment so grep-family tools match the completed userspace.

Open standalone page

> **Input assumption:** `bsdgrep-master.zip` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/arp242/bsdgrep/archive/refs/heads/master.zip`
>
> **Snapshot note:** this package is built from `master`, so upstream source content may change over time. The source archive staged by this book is pinned by `scripts/sources.b2sums`.
>
> **Upstream build note:** upstream documents the standard `make` and `make install` flow. The Makefile installs `grep`, `egrep`, `fgrep`, and `rgrep` by default.

Licenses:

- BSD-2-Clause-FreeBSD

Dependencies:

- musl (libc)
- make

bsdgrep is a FreeBSD-derived grep implementation for Linux. we need it to provide final target `grep`, `egrep`, `fgrep`, and `rgrep` command behavior for package builds and system use.

Extract and Enter the Source Tree

```
cd /sources
rm -rf bsdgrep-master
unzip -q bsdgrep-master.zip
cd bsdgrep-master
```

Apply Install Path Patch

```
sed -i '' \
    -e 's|${DESTDIR}${PREFIX}/bin|${DESTDIR}/system/binaries|g' \
    -e 's|${DESTDIR}${PREFIX}/share/man/man1|${DESTDIR}/system/documentation/man
    Makefile
```

Build bsdgrep

> `bsdgrep` does not ship a `configure` script, so `lbi_configure` is not applicable here.

```
make $LWI_MAKE_FLAGS \
    CC=cc \
    CFLAGS="-O2 -DREG_STARTEND=0 $LWI_CFLAGS" \
    LDFLAGS="$LBI_CUSTOM_LDFLAGS"
```

Install bsdgrep

```
make install
```

> After this step is complete, you can remove the extracted source directory and source archive from `/sources` if you do not plan to rebuild bsdgrep again.

Command Explanations

- `rm -rf bsdgrep-master` and `unzip -q`: Recreate a clean source tree from the snapshot archive.
- `sed -i ... Makefile`: Rewrites upstream install paths to the book's binary and man-page locations.
- `make $LWI_MAKE_FLAGS CC=cc`: Builds with the target C compiler and shared make parallelism.
- `CFLAGS="-O2 -DREG_STARTEND=0 $LWI_CFLAGS"`: Uses optimization, disables unsupported regex behavior, and preserves local C tuning.
- `LDFLAGS="$LBI_CUSTOM_LDFLAGS"`: Applies local linker tuning.
- `make install`: Installs bsdgrep using the patched paths.

8.14. LibreSSL 4.2.1

Build and install LibreSSL in the final target environment to provide TLS and OpenSSL-compatible cryptography libraries.

Open standalone page

> **Input assumption:** `libressl-4.2.1.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://cloudflare.cdn.openbsd.org/pub/OpenBSD/LibreSSL/libressl-4.2.1.tar.gz`
>
> **Upstream build note:** the official release tarball includes a generated `configure` script. Upstream documents the configure flow as `./configure`, `make check`, and `make install`; this section uses `lbi_configure` and skips the test subtree for the final target build.
>
> **Install note:** LibreSSL installs `cert.pem`, `openssl.cnf`, and `x509v3.cnf` under the configured OpenSSL directory if those files do not already exist.

Licenses:

- ISC
- OpenSSL License
- Original SSLeay License
- Public domain

Dependencies:

- musl (libc)

- make

LibreSSL is a TLS and cryptography library suite derived from OpenSSL. we need it to provide `libcrypto`, `libssl`, `libtls`, and the `openssl` tool for packages that require TLS or OpenSSL-compatible cryptography.

Extract and Enter the Source Tree

```
cd /sources
tar -xf libressl-4.2.1.tar.gz
cd libressl-4.2.1
```

Configure LibreSSL

```
lbi_configure \
    --with-openssldir=/system/configuration/ssl \
    --disable-static \
    --enable-shared \
    --disable-tests
```

Build LibreSSL

```
make $LWI_MAKE_FLAGS
```

Install LibreSSL

```
make install
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild LibreSSL again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd libressl-...`: Enter the staged LibreSSL source tree.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--with-openssldir=/system/configuration/ssl`: Stores TLS configuration and certificates under the book's configuration tree.
- `--disable-static` and `--enable-shared`: Install shared libraries without static archives.
- `--disable-tests`: Skips the test suite for the target build.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install LibreSSL.

8.14.1 ca-certificates 2026-03-19

8.14.1. ca-certificates 2026-03-19

Install a Mozilla-derived CA certificate bundle for TLS clients in the final target environment.

Open standalone page

> **Input assumption:** `cacert-2026-03-19.pem` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://curl.se/ca/cacert-2026-03-19.pem`
>
> **Source note:** curl's CA extract page publishes PEM bundles generated from Mozilla's root certificate store. The matching upstream SHA-256 sidecar for this file is `b6e66569cc3d438dd5abe514d0df50005d570bfc96c14dca8f768d020cb96171`.

Licenses:

- MPL-2.0

Dependencies:

- LibreSSL

ca-certificates is a bundle of public CA root certificates extracted from Mozilla's root store. we need it to let TLS clients such as curl, Cargo, and Python verify HTTPS servers in the final target environment.

Install ca-certificates

LibreSSL was configured with `/system/configuration/ssl` as its OpenSSL directory, and curl was configured to use `/system/configuration/ssl/cert.pem` as its CA bundle. Install the bundle there and add common compatibility names for tools that probe the traditional `/etc/ssl/certs` paths.

```
install -Dm644 /sources/cacert-2026-03-19.pem \
    /system/configuration/ssl/cert.pem

mkdir -p /system/configuration/ssl/certs

ln -sf ../cert.pem /system/configuration/ssl/certs/ca-certificates.crt
ln -sf ../cert.pem /system/configuration/ssl/certs/ca-bundle.crt
```

Add TLS Environment Defaults

These defaults make Cargo and other OpenSSL-using tools prefer the installed bundle even when their built-in certificate path probe does not know this system's layout.

```
grep -q '^export SSL_CERT_FILE=/system/configuration/ssl/cert.pem$' \
    /system/configuration/profile || \
    printf '%s\n' \
        'export SSL_CERT_FILE=/system/configuration/ssl/cert.pem' \
        >> /system/configuration/profile

grep -q '^export CARGO_HTTP_CAINFO=/system/configuration/ssl/cert.pem$' \
    /system/configuration/profile || \
    printf '%s\n' \
        'export CARGO_HTTP_CAINFO=/system/configuration/ssl/cert.pem' \
        >> /system/configuration/profile

export SSL_CERT_FILE=/system/configuration/ssl/cert.pem
export CARGO_HTTP_CAINFO=/system/configuration/ssl/cert.pem
```

Verify ca-certificates

```
test -s /system/configuration/ssl/cert.pem
openssl verify -CAfile /system/configuration/ssl/cert.pem \
    /system/configuration/ssl/cert.pem >/dev/null
```

Command Explanations

- `install -Dm644 /sources/cacert-...pem`: Installs the CA bundle and creates its parent directory.
- `mkdir -p /system/configuration/ssl/certs`: Creates the certificate-directory compatibility location.
- `ln -sf ../cert.pem ...`: Provides common CA bundle filenames used by TLS clients.
- `grep -q ... || printf ... >> /system/configuration/profile`: Adds certificate environment variables only if they are not already present.
- `SSL_CERT_FILE` and `CARGO_HTTP_CAINFO`: Point OpenSSL-compatible tools and Cargo at the installed CA bundle.
- `test -s ...`: Verifies that the installed bundle exists and is not empty.
- `openssl verify -CAfile ...`: Performs a basic verification check against the installed bundle.

8.15. Flex 2.6.4

Build and install Flex in the final target environment to provide lex-compatible scanner generation.

Open standalone page

> **Input assumption:** `flex-2.6.4.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/westes/flex/releases/download/v2.6.4/flex-2.6.4.tar.gz`
>
> **Upstream build note:** the release tarball includes a generated `configure` script, generated parser and scanner sources, and prebuilt manual pages. Upstream documents the normal build flow as `configure`, `make`, and `make install`.
>
> **Tool note:** Flex requires an `m4` implementation that supports `-P`; chapter 6 installs `om4` as the target `m4` command.

Licenses:

- BSD-style Flex license

Dependencies:

- musl (libc)
- m4
- make

Flex is a fast lexical analyzer generator. we need it to provide final target `flex`, `lex`, and `libfl` support for packages that generate scanners during their build.

Extract and Enter the Source Tree

```
cd /sources
tar -xf flex-2.6.4.tar.gz
cd flex-2.6.4
```

Configure Flex

```
lbi_configure \
    --docdir=/system/documentation/flex \
```

```
    --disable-static \
    --enable-shared \
    --disable-nls
```

Build Flex

```
make $LWI_MAKE_FLAGS
```

Install Flex

```
make install
```

Install lex Compatibility Links

```
ln -sf flex /system/binaries/lex
ln -sf flex.1 /system/documentation/man-pages/man1/lex.1
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild Flex again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd flex-...`: Enter the staged Flex source tree.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--docdir=/system/documentation/flex`: Places package documentation in the documentation tree.
- `--disable-static` and `--enable-shared`: Build shared libraries without static archives.
- `--disable-nls`: Avoids gettext/native language support.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install Flex.
- `ln -sf flex ... lex` and `ln -sf flex.1 ... lex.1`: Provide traditional lex command and manual-page aliases.

8.16. SQLite 3.53.0

Build and install SQLite in the final target environment to provide an embedded SQL database engine.

Open standalone page

> **Input assumption:** `sqlite-autoconf-3530000.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://sqlite.org/2026/sqlite-autoconf-3530000.tar.gz`
>
> **Upstream build note:** SQLite's autoconf bundle contains the prebuilt amalgamation, generated headers, the `sqlite3` shell source, and an autosetup-based `configure` script. The upstream README documents the POSIX build flow as `./configure` followed by `make`.
>
> **Feature note:** this build enables column metadata, unlock notification, the dbstat virtual table, and secure delete through `CPPFLAGS`.

Licenses:

- Public domain

Dependencies:

- musl (libc)
- zlib-ng (libz)
- make

SQLite is a self-contained SQL database engine. we need it to provide the `sqlite3` shell, `libsqlite3`, and SQLite headers for packages and system components that store structured data in local database files.

Extract and Enter the Source Tree

```
cd /sources
tar -xf sqlite-autoconf-3530000.tar.gz
cd sqlite-autoconf-3530000
```

Configure SQLite

> The configure script is autosetup-based, but it accepts the directory options provided by `lbi_configure`.

```
CPPFLAGS="-D SQLITE_ENABLE_COLUMN_METADATA=1 \
          -D SQLITE_ENABLE_UNLOCK_NOTIFY=1 \
          -D SQLITE_ENABLE_DBSTAT_VTAB=1 \
          -D SQLITE_SECURE_DELETE=1" \
lbi_configure \
```

```
    --disable-static \
    --enable-shared \
    --disable-readline \
    --soname=legacy
```

Build SQLite

```
make $LWI_MAKE_FLAGS
```

Install SQLite

```
make install
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild SQLite again.

Command Explanations

- `cd /sources`, `tar -xf`, and `cd sqlite-...`: Enter the staged SQLite source tree.
- `CPPFLAGS="-D SQLITE_ENABLE_..."`: Enables selected SQLite features at compile time.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--disable-static` and `--enable-shared`: Build shared SQLite libraries without static archives.
- `--disable-readline`: Avoids a readline dependency.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install SQLite.

8.17 python 3.14.4

8.17. python 3.14.4

Rebuild python in the final target environment with LibreSSL, pip, and optimized runtime support.

Open standalone page

> **Input assumption:** `Python-3.14.4.tar.xz` and `python-3.14.4-libressl-hostflags-guard.patch` are already present in `/sources` from the chapter 4 source staging step and the book's distributed patch files.
>
> **Source URL:** `https://www.python.org/ftp/python/3.14.4/Python-3.14.4.tar.xz`

> **Patch file:** `patches/python-3.14.4-libressl-hostflags-guard.patch`
>
> **Upstream build note:** upstream `configure --help` documents `--enable-optimizations` as the stable optimization path using PGO. The same configure help documents `--with-pkg-config`, `--with-openssl-rpath`, `--with-platlibdir`, `--with-tzpath`, `--with-ensurepip`, and `--without-static-libpython`.
>
> **LibreSSL note:** this build relies on LibreSSL's installed `openssl.pc` file instead of `--with-openssl=/system`, because the book's include directory is `/system/headers`, not `/system/include`.

Licenses:

- Python-2.0

Dependencies:

- musl (libc)
- LibreSSL
- SQLite
- pkgconf
- xz
- zlib-ng
- make

python is a general-purpose programming language and runtime. we need it to provide an optimized final `python3` runtime with `pip3`, `_ssl`, `_hashlib`, and SQLite module support for system tools and package builds.

Extract and Enter the Source Tree

```
cd /sources
rm -rf Python-3.14.4
tar -xf Python-3.14.4.tar.xz
cd Python-3.14.4
```

Apply LibreSSL Patch

```
patch -Np1 -i ../python-3.14.4-libressl-hostflags-guard.patch
```

Configure python

> `--enable-optimizations` makes the build slower because Python performs profile-guided optimization. `--with-ensurepip=install` installs the bundled pip tooling so later Python package sections can use `pip3`.

```
ax_cv_c_float_words_bigendian=no \
lbi_configure \
    --enable-shared \
    --enable-optimizations \
    --with-pkg-config=yes \
    --with-openssl-rpath=auto \
    --with-ensurepip=install \
    --without-static-libpython \
    --with-tzpath= \
    --with-platlibdir=libraries
```

Build python

```
make $LWI_MAKE_FLAGS
```

Install python

```
make install
```

Move pip Scripts to /system/binaries

> `ensurepip` may install `pip3` and `pip3.14` into `/system/bin` instead of the book's configured `/system/binaries` directory.

```
if [ -d /system/bin ]; then
    for pip_script in pip3 pip3.14; do
        if [ -e "/system/bin/$pip_script" ]; then
            mv -f "/system/bin/$pip_script" /system/binaries/
        fi
    done

    rmdir /system/bin 2>/dev/null || true
fi
```

Verify pip

```
pip3 --version
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild python again.

Command Explanations

- `rm -rf Python-3.14.4`: Removes any previous Python source tree before rebuilding.
- `patch -Np1 -i ../python-...patch`: Applies the book's LibreSSL host-flags guard patch.
- `ax_cv_c_float_words_bigendian=no`: Preanswers a Python configure probe for the target.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--enable-shared`, `--enable-optimizations`, and `--with-ensurepip=install`: Build shared Python, enable optimized build behavior, and install pip.
- `--with-pkg-config=yes` and `--with-openssl-rpath=auto`: Use pkgconf metadata and automatic OpenSSL/LibreSSL runtime path handling.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install Python.
- `mv -f /system/bin/pip* /system/binaries/`: Moves pip scripts out of upstream's default `bin` directory.
- `pip3 --version`: Verifies the installed pip command.

8.18. Python-Flit-Core 3.12.0

Build and install flit_core so Python packages using the Flit PEP 517 backend can be built.

Open standalone page

> **Input assumption:** `flit_core-3.12.0.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://pypi.org/packages/source/f/flit-core/flit_core-3.12.0.tar.gz`
>
> **Upstream build note:** upstream `pyproject.toml` declares `requires = []`, `build-backend = "flit_core.buildapi"`, and `backend-path = ["."]`, so the source tree can build its own wheel without build isolation or extra build dependencies.

Licenses:

- BSD-3-Clause
- MIT (vendored tomli)

Dependencies:

- python
- pip

Python-Flit-Core is the distribution-building core of Flit and provides a PEP 517 build backend. we need it to build Python packages that use `flit_core.buildapi` as their build backend.

Extract and Enter the Source Tree

```
cd /sources
rm -rf flit_core-3.12.0
tar -xf flit_core-3.12.0.tar.gz
cd flit_core-3.12.0
```

Build Python-Flit-Core

```
pip3 wheel -w dist --no-cache-dir --no-build-isolation --no-deps $PWD
```

Install Python-Flit-Core

```
pip3 install --no-index --find-links dist flit_core
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild Python-Flit-Core again.

Command Explanations

- `rm -rf flit_core-...`: Removes any previous extracted flit-core tree.
- `tar -xf flit_core-...tar.gz`: Extracts the staged flit-core source archive.
- `pip3 wheel -w dist --no-cache-dir --no-build-isolation --no-deps $PWD`: Builds a local wheel without downloading dependencies or using build isolation.
- `pip3 install --no-index --find-links dist flit_core`: Installs flit-core from the locally built wheel only.

8.19. bfs stage 2 4.1

Rebuild bfs in the final target environment so the find-compatible traversal tool matches the completed userspace.

Open standalone page

> **Input assumption:** `bfs-4.1.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/tavianator/bfs/releases/download/4.1/bfs-4.1.tar.gz`
>
> **Source archive note:** this release archive is flat (no top-level directory), so extract it into a dedicated `bfs-4.1/` directory.
>
> **Upstream build note:** upstream documents `./configure` followed by `make`. The custom configure wrapper supports `--enable-release` and explicit `--without-*` dependency toggles, but it does not support the full `lbi_configure` directory option set.

Licenses:

- oBSD

Dependencies:

- musl (libc)
- make

bfs is a breadth-first `find`-compatible file search utility. we need it to provide final target `bfs` and `find` command behavior in the completed chapter 8 userspace.

Extract and Enter the Source Tree

```
cd /sources
rm -rf bfs-4.1
mkdir -p bfs-4.1
tar -xf bfs-4.1.tar.gz -C bfs-4.1
cd bfs-4.1
```

Configure bfs

> `bfs` does not use an autotools-compatible `configure` script, so `lbi_configure` is not applicable here.

```
CC=cc \
CFLAGS="-O2 $LWI_CFLAGS" \
LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
./configure \
    --enable-release \
    --without-libacl \
    --without-libcap \
    --without-libselinux \
    --without-liburing \
    --without-oniguruma
```

Build bfs

```
make $LWI_MAKE_FLAGS
```

Install bfs

```
install -Dm755 bin/bfs /system/binaries/bfs
install -Dm644 docs/bfs.1 /system/documentation/man-pages/man1/bfs.1
```

Replace `find` with bfs

```
ln -sf bfs /system/binaries/find
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild bfs again.

Command Explanations

- `rm -rf bfs-4.1` and `mkdir -p bfs-4.1`: Recreate the dedicated source directory for bfs' flat archive.
- `tar -xf bfs-4.1.tar.gz -C bfs-4.1`: Extracts the source archive into that directory.
- `CC=cc`, `CFLAGS`, and `LDFLAGS`: Build with the target compiler and local tuning variables.
- `./configure --enable-release`: Configures bfs for a release build.

- `--without-libacl`, `--without-libcap`, `--without-libselinux`, `--without-liburing`, and `--without-oniguruma`: Disable optional dependencies not required in this target system.
- `make $LWI_MAKE_FLAGS`: Builds with shared make parallelism.
- `install -Dm755` and `install -Dm644`: Install the executable and man page.
- `ln -sf bfs /system/binaries/find`: Provides the standard `find` command name.

8.20. bmake 20260406

Install the portable NetBSD make implementation for packages that expect BSD make behavior.

Open standalone page

> **Input assumption:** `bmake-20260406.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://www.crufty.net/ftp/pub/sjg/bmake-20260406.tar.gz`
>
> **Upstream build note:** upstream documents `./bmake/boot-strap` as the preferred bootstrap path and also supports the standard `./configure`, `make`, and `make install` flow. This section uses `lbi_configure` because the generated configure script accepts the book's standard installation directory options, then uses the newly built `bmake` for installation so the book's directory layout is applied consistently.

Licenses:

- BSD-2-Clause

Dependencies:

- musl (libc)
- make

bmake is a portable NetBSD make implementation. we need it to provide BSD make semantics and the accompanying `mk` include files for packages that are written for BSD-style makefiles.

Extract and Enter the Source Tree

```
cd /sources
rm -rf bmake
```

```
tar -xf bmake-20260406.tar.gz
cd bmake
```

Configure bmake

```
lbi_configure \
    --with-default-sys-path=/system/share/mk \
    --with-mksrc=mk \
    --with-filemon=no \
    --without-lua
```

Build bmake

```
make $LWI_MAKE_FLAGS
```

Install bmake

```
MAKESYSPATH=mk \
./bmake -f Makefile install \
    prefix=/system \
    BINDIR.bmake=/system/binaries \
    SHAREDIR.bmake=/system/share \
    MANDIR.bmake=/system/documentation/man-pages \
    STRIP_FLAG=
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild bmake again.

Command Explanations

- `rm -rf bmake` and `tar -xf`: Recreate a clean bmake source tree.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--with-default-sys-path=/system/share/mk`: Sets the default makefile include path.
- `--with-mksrc=mk`: Points bmake at its bundled `mk` files.
- `--with-filemon=no` and `--without-lua`: Disable optional filemon and Lua support.
- `make $LWI_MAKE_FLAGS`: Builds bmake with shared make parallelism.
- `MAKESYSPATH=mk ./bmake -f Makefile install`: Uses the newly built bmake and bundled mk files for installation.

- `BINDIR.bmake`, `SHAREDIR.bmake`, and `MANDIR.bmake`: Install bmake files into the book's layout.

8.21. BSD-Diffutils stage 2 main snapshot

Rebuild BSD-Diffutils in the final target environment so the diff-family tools match the completed userspace.

Open standalone page

> **Input assumption:** `BSD-Diffutils-main.zip` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/littlefly365/BSD-Diffutils/archive/refs/heads/main.zip`
>
> **Snapshot note:** this package is built from `main`, so upstream source content may change over time. The source archive staged by this book is pinned by `scripts/sources.b2sums`.
>
> **Upstream build note:** upstream lists `glibc` or `musl`, `clang` or `gcc`, and `bmake` as requirements. The package uses BSD makefiles and does not ship a `configure` script.

Licenses:

- BSD-3-Clause

Dependencies:

- musl (libc)
- bmake
- bheaded

BSD-Diffutils is a BSD-style diff and comparison utility set. we need it to provide final target `cmp`, `diff`, `diff3`, and `sdiff` tools for validation, package builds, and day-to-day system work.

Extract and Enter the Source Tree

```
cd /sources
rm -rf BSD-Diffutils-main
unzip -q BSD-Diffutils-main.zip
cd BSD-Diffutils-main
```

Remove Bundled Build Outputs

> The staged snapshot contains previously generated object and library files. Remove them before rebuilding so the installed tools are built inside the final target environment.

```
find . \( \
    -name '*.o' -o \
    -name '*.po' -o \
    -name '*.pico' -o \
    -name '*.a' -o \
    -name '*.d' -o \
    -name '.depend' \
\) -exec rm -f {} +
```

Apply Compatibility and Layout Patches

```
sed -i '' \
    -e '1s|^#!/bin/ksh -$|#!/bin/sh|' \
    -e 's|^diff3prog=/usr/libexec/diff3prog$|diff3prog=/system/systembinaries/di
    -e 's|^export PATH=.*$|export PATH=/system/binaries:/system/systembinaries|'
    src/diff3/diff3.ksh

sed -i '' \
    's|${DESTDIR}/usr/bin/diff3|${DESTDIR}/system/binaries/diff3|' \
    src/diff3/Makefile

sed -i '' \
    's|char[[:space:]]*\*splice(char \*, char \*);|char    *diff_splice(char *, 
    src/diff/diff.h

sed -i '' \
    -e 's|= splice(|= diff_splice(|g' \
    -e 's|^splice(char \*dir, char \*file)|diff_splice(char *dir, char *file)|' 
    src/diff/diff.c src/diff/diffreg.c

sed -i '' \
    's|u_char ch, \*p1, \*p2;|unsigned char ch, *p1, *p2;|' \
    src/cmp/regular.c
```

Build BSD-Diffutils

```
for d in common cmp diff diff3 sdiff; do
    CC=cc \
    CPPFLAGS='-D__dead=__attribute__\(\(__noreturn__\)\)' \
    CFLAGS="-O2 $LWI_CFLAGS" \
    LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
    bmake -C "src/$d" $LWI_MAKE_FLAGS
done
```

Install BSD-Diffutils

```
install -d /system/binaries /system/systembinaries

for d in cmp diff sdiff; do
    CC=cc \
    CPPFLAGS='-D__dead=__attribute__\(\(__noreturn__\)\)' \
    CFLAGS="-O2 $LWI_CFLAGS" \
    LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
    bmake -C "src/$d" \
        BINDIR=/system/binaries \
        MANDIR=/system/documentation/man-pages \
        STRIP_FLAG= \
        BINOWN="$(id -un)" BINGRP="$(id -gn)" \
        MANOWN="$(id -un)" MANGRP="$(id -gn)" \
        install
done

CC=cc \
CPPFLAGS='-D__dead=__attribute__\(\(__noreturn__\)\)' \
CFLAGS="-O2 $LWI_CFLAGS" \
LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
bmake -C src/diff3 \
    BINDIR=/system/systembinaries \
    MANDIR=/system/documentation/man-pages \
    STRIP_FLAG= \
    BINOWN="$(id -un)" BINGRP="$(id -gn)" \
    MANOWN="$(id -un)" MANGRP="$(id -gn)" \
    install
```

> After this step is complete, you can remove the extracted source directory and source archive

from `/sources` if you do not plan to rebuild BSD-Diffutils again.

Command Explanations

- `rm -rf BSD-Diffutils-main` and `unzip -q`: Recreate a clean snapshot source tree.
- `find . ... -exec rm -f {} +`: Removes stale build artifacts from previous attempts.
- `sed -i ... diff3.ksh` and `sed -i ... Makefile`: Patch shell, PATH, helper, and install locations for the book's layout.
- `for d in common cmp diff diff3 sdiff; do ... bmake -C ...`: Builds each BSD make subdirectory with target compiler settings.
- `CPPFLAGS=...__dead...`: Provides the BSD `__dead` attribute macro expected by the sources.
- `install -d /system/binaries /system/systembinaries`: Creates the command destination directories.
- `bmake -C "src/$d" ... install`: Installs each user-facing utility into `/system/binaries`.
- `bmake -C src/diff3 ... BINDIR=/system/systembinaries`: Installs `diff3prog` into the system-binary directory used by the wrapper script.

8.22 awk stage 2 20251225

8.22. awk stage 2 20251225

Rebuild One True Awk in the final target environment so POSIX text-processing scripts use the completed userspace.

Open standalone page

> **Input assumption:** `awk-20251225.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:**
> `https://github.com/onetrueawk/awk/archive/refs/tags/20251225.tar.gz`
>
> **Upstream build note:** upstream documents a plain `make` build that produces an executable named `a.out`, then expects the builder to install it as `awk`. The bundled makefile defaults to `bison -d`; this section passes the yacc-compatible command provided earlier in this chapter.

Licenses:

- Lucent Technologies permissive license

Dependencies:

- musl (libc)
- make
- yacc

awk is a POSIX awk language interpreter from the One True Awk project. we need it to provide final target `awk` for text processing and script compatibility in the completed userspace.

Extract and Enter the Source Tree

```
cd /sources
rm -rf awk-20251225
tar -xf awk-20251225.tar.gz
cd awk-20251225
```

Build awk

> `awk` does not ship a `configure` script, so `lbi_configure` is not applicable here.

```
make $LWI_MAKE_FLAGS \
    HOSTCC="cc -g -Wall -pedantic -Wcast-qual" \
    CC="cc $LWI_CFLAGS $LBI_CUSTOM_LDFLAGS" \
    YACC="yacc -d -b awkgram"
```

Install awk

```
install -Dm755 a.out /system/binaries/awk
install -Dm644 awk.1 /system/documentation/man-pages/man1/awk.1
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild awk again.

Command Explanations

- `rm -rf awk-20251225`: Removes any previous awk source tree.
- `tar -xf awk-20251225.tar.gz`: Extracts the staged awk archive.
- `HOSTCC="cc ..."`: Builds host-side helpers that must run during the build.
- `CC="cc $LWI_CFLAGS $LBI_CUSTOM_LDFLAGS"`: Builds the target awk binary with local compile and link tuning.

- `YACC="yacc -d -b awkgram"`: Uses yacc to generate parser sources with awk's expected basename.
- `install -Dm755 a.out ... awk`: Installs the generated executable as `awk`.
- `install -Dm644 awk.1 ...`: Installs the awk manual page.

8.23 patch stage 2 0.99.1

8.23. patch stage 2 0.99.1

Rebuild the FreeBSD-derived patch utility in the final target environment.

Open standalone page

> **Input assumption:** `bsdpatch-v0.99.1.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/chimera-linux/bsdpatch/archive/refs/tags/v0.99.1.tar.gz`
>
> **Upstream build note:** upstream provides a simple portable Makefile. It builds with `make` and supports install-time `PREFIX`, `BINDIR`, `DATADIR`, `MANDIR`, and `INSTALL_NAME` variables.

Licenses:

- BSD-2-Clause-FreeBSD

Dependencies:

- musl (libc)
- make
- awk

patch is a FreeBSD-derived patch utility made portable by the Chimera Linux bsdpatch project. we need it to provide the final target `patch` command for applying source changes during maintenance and later package work.

Extract and Enter the Source Tree

```
cd /sources
rm -rf bsdpatch-0.99.1
tar -xf bsdpatch-v0.99.1.tar.gz
cd bsdpatch-0.99.1
```

Apply musl Compatibility Edits

```
sed -i '' \
    -e 's|dp->d_namlen|strlen(dp->d_name)|' \
    backupfile.c

sed -i '' \
    -e 's|optreset = optind = 1;|optind = 1;|' \
    patch.c

sed -i '' \
    -e 's|fgetln(|lbi_fgetln(|g' \
    inp.c pch.c

sed -i '' \
    '/char[[:space:]]*\*xstrdup/a\
char            *lbi_fgetln(FILE *, size_t *);
' \
    util.h

awk '
    /\/\* Rename a file, copying it if necessary\. \*\// {
        print "char *"
        print "lbi_fgetln(FILE *stream, size_t *len)"
        print "{"
        print "\tstatic char *line;"
        print "\tstatic size_t cap;"
        print "\tssize_t nread;"
        print ""
        print "\tnread = getline(&line, &cap, stream);"
        print "\tif (nread < 0)"
        print "\t\treturn NULL;"
        print "\t*len = (size_t)nread;"
        print "\treturn line;"
        print "}"
        print ""
    }
    { print }
' util.c > util.c.new
mv util.c.new util.c
```

Build patch

> `bsdpatch` does not ship a `configure` script, so `lbi_configure` is not applicable here.

```
make $LWI_MAKE_FLAGS \
    CC=cc \
    CFLAGS="-O2 $LWI_CFLAGS" \
    LDFLAGS="$LBI_CUSTOM_LDFLAGS"
```

Install patch

```
make install \
    PREFIX=/system \
    BINDIR=/system/binaries \
    DATADIR=/system/documentation \
    MANDIR=/system/documentation/man-pages/man1 \
    INSTALL_NAME=patch
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild patch again.

Command Explanations

- `rm -rf bsdpatch-...` and `tar -xf`: Recreate a clean bsdpatch source tree.
- `sed -i ... backupfile.c`: Replaces BSD `d_namlen` usage with portable `strlen(dp->d_name)`.
- `sed -i ... patch.c`: Removes `optreset` usage that is not portable to musl.
- `sed -i ... fgetln`: Renames local compatibility functions to avoid namespace conflicts.
- `make $LWI_MAKE_FLAGS CC=cc`: Builds patch with the target compiler and shared make parallelism.
- `make install PREFIX=/system ... INSTALL_NAME=patch`: Installs the utility under the standard `patch` command name in the book's layout.

8.24 mandoc 1.14.6

8.24. mandoc 1.14.6

Build and install mandoc as the final manual page compiler, viewer, and indexer.

Open standalone page

> **Input assumption:** `mandoc-1.14.6.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://mandoc.bsd.lv/snapshots/mandoc-1.14.6.tar.gz`
>
> **Upstream build note:** upstream documents the portable build as `./configure`, `make`, `make install`, and then `makewhatis`. The configure script is not generated by GNU autotools; it reads `configure.local` for installation paths and feature choices, so `lbi_configure` is not applicable here.

Licenses:

- ISC-style permissive license
- BSD-2-Clause and BSD-3-Clause licenses for imported compatibility files

Dependencies:

- musl (libc)
- make
- zlib-ng

mandoc is a manual page compiler and viewer for `mdoc`, `man`, and roff documents. we need it to provide the final target `man`, `apropos`, `whatis`, `makewhatis`, and `soelim` commands for reading and indexing installed manual pages.

Extract and Enter the Source Tree

```
cd /sources
rm -rf mandoc-1.14.6
tar -xf mandoc-1.14.6.tar.gz
cd mandoc-1.14.6
```

Configure mandoc

```
cat > configure.local <<EOF
PREFIX=/system
BINDIR=/system/binaries
SBINDIR=/system/systembinaries
MANDIR=/system/documentation/man-pages
MANPATH_DEFAULT="/system/documentation/man-pages"
```

```
MANPATH_BASE="/system/documentation/man-pages"
BIN_FROM_SBIN="../binaries"
BINM_PAGER=cat
LN="ln -sf"
CC=cc
CFLAGS="-O2 $LWI_CFLAGS"
LDFLAGS="$LBI_CUSTOM_LDFLAGS"
EOF

./configure
grep '#define BINM_PAGER "cat"' config.h
```

> `BINM_PAGER` is compiled into `mandoc` as the fallback pager. The `grep` check catches stale configure output before installation. At runtime, `MANPAGER` and `PAGER` still override this fallback, so set either variable to `cat` if your shell environment points at a pager that is not installed yet.

Build mandoc

```
make $LWI_MAKE_FLAGS
```

Install mandoc

```
make install
/system/systembinaries/makewhatis /system/documentation/man-pages
MANPAGER=cat man mandoc >/dev/null
```

> This install provides `man`, `apropos`, and `whatis` as links to `mandoc`, installs `soelim`, and installs `makewhatis` in `/system/systembinaries`.

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild mandoc again.

Command Explanations

- `rm -rf mandoc-...` and `tar -xf`: Recreate a clean mandoc source tree.
- `cat > configure.local`: Writes mandoc's supported local configuration file instead of using an autotools-style configure command.

- `PREFIX`, `BINDIR`, `SBINDIR`, and `MANDIR`: Set install paths for the book's `/system` layout.
- `MANPATH_DEFAULT` and `MANPATH_BASE`: Point mandoc tools at the book's man-page tree.
- `make $LWI_MAKE_FLAGS`: Builds mandoc with shared make parallelism.
- `make install`: Installs mandoc tools into the target filesystem.
- `makewhatis /system/documentation/man-pages`: Builds the manual-page database.
- `MANPAGER=cat man mandoc >/dev/null`: Smoke-tests the installed `man` command without invoking an interactive pager.

8.25 libedit stage 2 20251016-3.1

8.25. libedit stage 2 20251016-3.1

Rebuild libedit in the final target environment so interactive programs use the completed curses stack.

Open standalone page

> **Input assumption:** `libedit-20251016-3.1.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://thrysoee.dk/editline/libedit-20251016-3.1.tar.gz`
>
> **Upstream build note:** the release tarball includes a generated autotools `configure` script. Upstream documents the normal build flow as `./configure`, `make`, and `make install`; this section uses that flow through `lbi_configure`.

Licenses:

- BSD-3-Clause

Dependencies:

- musl (libc)
- make
- ncurses

libedit is a BSD editline and history library. we need it to provide final target line-editing and command-history functionality for interactive programs such as the rebuilt shell.

Extract and Enter the Source Tree

```
cd /sources
rm -rf libedit-20251016-3.1
```

```
tar -xf libedit-20251016-3.1.tar.gz
cd libedit-20251016-3.1
```

Configure libedit

```
CPPFLAGS="-D__STDC_ISO_10646__=201706L" \
lbi_configure \
    --disable-static \
    --enable-shared \
    --disable-examples
```

Build libedit

```
make $LWI_MAKE_FLAGS
```

Install libedit

```
make install
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild libedit again.

Command Explanations

- `rm -rf libedit-...` and `tar -xf`: Recreate a clean libedit source tree.
- `CPPFLAGS="-D__STDC_ISO_10646__=201706L"`: Enables wide-character assumptions expected by libedit checks.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--disable-static` and `--enable-shared`: Build shared libedit without static archives.
- `--disable-examples`: Skips example programs.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install libedit.

8.26 dash stage 2 0.5.13.3

8.26. dash stage 2 0.5.13.3

Rebuild dash in the final target environment and link it against the final libedit.

Open standalone page

> **Input assumption:** `dash-0.5.13.3.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `http://gondor.apana.org.au/~herbert/dash/files/dash-0.5.13.3.tar.gz`
>
> **Release note:** this upstream release tarball includes a generated `configure` script and `Makefile.in` files, so no autotools bootstrap is needed.
>
> **Upstream build note:** the generated `configure` script supports `--with-libedit` for editline support and accepts `CC_FOR_BUILD` for the small build-time generators used by the shell parser tables.

Licenses:

- BSD-3-Clause
- GPL-2.0-or-later for the `mksignames.c` generator input noted by upstream

Dependencies:

- musl (libc)
- make
- libedit
- ncurses

dash is a small POSIX-compliant Bourne shell implementation. we need it to provide the final target `dash` and `sh` commands for fast script execution, interactive recovery work, and a shell linked against the final libedit.

Extract and Enter the Source Tree

```
cd /sources
rm -rf dash-0.5.13.3
tar -xf dash-0.5.13.3.tar.gz
cd dash-0.5.13.3
```

Configure dash

```
lbi_configure \
    --with-libedit \
    CC_FOR_BUILD=cc
```

Build dash

```
make $LWI_MAKE_FLAGS
```

Install dash

```
make install
ln -sf dash /system/binaries/sh
```

Verify the Shell

```
dash -c 'echo dash-ok'
sh -c 'echo sh-ok'
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild dash again.

Command Explanations

- `rm -rf dash-...` and `tar -xf`: Recreate a clean dash source tree.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--with-libedit`: Builds dash with line-editing support from libedit.
- `CC_FOR_BUILD=cc`: Uses the target compiler for build helpers inside the self-hosted target.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install dash.
- `ln -sf dash /system/binaries/sh`: Provides the standard `sh` command name.
- `dash -c` and `sh -c`: Smoke-test both command names.

8.27 curl 8.19.0

8.27. curl 8.19.0

Build curl and libcurl in the final target environment using LibreSSL and zlib-ng.

Open standalone page

> **Input assumption:** `curl-8.19.0.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://curl.se/download/curl-8.19.0.tar.xz`

> **Upstream build note:** the release tarball includes a generated `configure` script. Upstream documents the normal Unix flow as `./configure`, `make`, optional `make test`, and `make install`. It also documents that a TLS backend must be selected explicitly unless TLS is disabled.
>
> **Documentation note:** curl's generated command and library man pages use Perl during the build. This system does not include Perl, so this section disables generated docs and the built-in `curl --manual` text.

Licenses:

- curl license

Dependencies:

- musl (libc)
- make
- pkgconf
- LibreSSL
- zlib-ng

curl is a URL transfer tool and client-side transfer library. we need it to provide the final target `curl` command and `libcurl` for fetching network resources over HTTP, HTTPS, FTP, and related protocols.

Extract and Enter the Source Tree

```
cd /sources
rm -rf curl-8.19.0
tar -xf curl-8.19.0.tar.xz
cd curl-8.19.0
```

Configure curl

```
PERL=false \
lbi_configure \
    --with-openssl \
    --with-zlib \
    --with-ca-bundle=/system/configuration/ssl/cert.pem \
    --without-brotli \
    --with-zstd \
    --without-libpsl \
```

```
    --without-libidn2 \
    --without-nghttp2 \
    --without-ngtcp2 \
    --without-nghttp3 \
    --without-quiche \
    --without-libssh2 \
    --without-libssh \
    --without-librtmp \
    --without-libgsasl \
    --without-ldap \
    --without-libuv \
    --without-zsh-functions-dir \
    --without-fish-functions-dir \
    --disable-ldap \
    --disable-ldaps \
    --disable-manual \
    --disable-docs \
    --disable-static \
    --enable-shared
```

> The `--with-openssl` option also covers LibreSSL. With `pkgconf` available, curl uses the installed `openssl.pc` and `zlib.pc` metadata instead of assuming the conventional `/system/include` layout.

Build curl

```
make $LWI_MAKE_FLAGS
```

Install curl

```
make install
```

Verify curl

```
curl --version
curl-config --features
```

> After this step is complete, you can remove the extracted source directory and source tarball from

> `/sources` if you do not plan to rebuild curl again.

Command Explanations

- `rm -rf curl-...` and `tar -xf`: Recreate a clean curl source tree.
- `PERL=false`: Prevents curl's build from depending on Perl helpers.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--with-openssl`, `--with-zlib`, and `--with-ca-bundle=...`: Enable TLS, zlib support, and the installed CA bundle path.
- `--without-brotli`, `--without-zstd`, `--without-libpsl`, and related toggles: Disable optional dependencies not selected for this curl build.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install curl.
- `curl --version` and `curl-config --features`: Verify the command and installed feature metadata.

8.28 samurai stage 2 1.3

8.28. samurai stage 2 1.3

samurai provides a small Ninja-compatible build executor for the target userspace.

Open standalone page

> **Input assumption:** `samurai-1.3.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- ISC
- Apache-2.0

Dependencies:

- musl (libc)
- make

samurai is a ninja-compatible build tool written in C99.

Extract and Enter the Source Tree

```
cd "/sources"
tar -xf samurai-1.3.tar.gz
```

```
cd samurai-1.3
```

Build samurai

> **Configure note:** samurai uses a small POSIX Makefile and does not ship a `configure` script, so `lbi_configure` is not supported here. Build settings are passed directly to `make`.

```
make $LWI_MAKE_FLAGS \
    CC="clang" \
    CFLAGS="$LWI_CFLAGS" \
    LDFLAGS="$LBI_CUSTOM_LDFLAGS"
```

Install samurai

```
make install \
    PREFIX=/system \
    BINDIR=/system/binaries \
    MANDIR=/system/documentation/man-pages

ln -sf samu "/system/binaries/ninja"
ln -sf samu.1 "/system/documentation/man-pages/man1/ninja.1"
```

Command Explanations

- `cd /sources`, `tar -xf`, and `cd samurai-...`: Enter the staged samurai source tree.
- `make $LWI_MAKE_FLAGS CC="clang"`: Builds samurai with Clang and shared make parallelism.
- `CFLAGS` and `LDFLAGS`: Apply local compile and link tuning.
- `make install PREFIX=/system ...`: Installs `samu` and its manual page into the book's layout.
- `ln -sf samu /system/binaries/ninja`: Provides the common `ninja` command name through samurai.
- `ln -sf samu.1 ... ninja.1`: Provides the matching manual-page alias.

8.29. CMake 4.3.2

Bootstrap and install CMake in the final target environment without requiring an existing target CMake.

Open standalone page

> **Input assumption:** `cmake-4.3.2.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/Kitware/CMake/releases/download/v4.3.2/cmake-4.3.2.tar.gz`
>
> **Upstream build note:** upstream documents the Unix source build as `./bootstrap`, `make`, and `make install`. The bootstrap script creates a temporary CMake first, so this section does not require an existing target `cmake`.
>
> **License note:** the CMake source tree is BSD-3-Clause. The tarball also contains bundled third-party code and license texts; this section disables the Qt GUI and uses the already installed system `curl`, `zlib`, `liblzma`, and `libarchive` libraries to avoid building the bundled copies of those components.

Licenses:

- BSD-3-Clause

Dependencies:

- musl (libc)
- samurai (ninja)
- C++ compiler
- ncurses
- curl
- zlib-ng
- xz
- libarchive

CMake is a cross-platform build-system generator and project configuration tool. we need it to provide the final target `cmake`, `ccmake`, `ctest`, and `cpack` commands for packages that use CMake build definitions.

Extract and Enter the Source Tree

```
cd /sources
rm -rf cmake-4.3.2
tar -xf cmake-4.3.2.tar.gz
cd cmake-4.3.2
```

Bootstrap CMake

```
CC=/system/binaries/cc CXX=/system/binaries/c++ ./bootstrap \
    --prefix=/system \
    --bindir=binaries \
    --datadir=share/cmake-4.3 \
    --docdir=documentation/cmake \
    --mandir=documentation/man-pages \
    --parallel="$LWI_MAKE_JOBS" \
    --no-qt-gui \
    --no-system-libs \
    --system-curl \
    --system-zlib \
    --system-liblzma \
    --system-libarchive \
    --generator=Ninja \
    -- \
    -DCMAKE_BUILD_TYPE=Release \
    -DBUILD_TESTING=OFF
```

Build CMake

```
ninja $LWI_MAKE_FLAGS
```

Install CMake

```
ninja install
```

Verify CMake

```
cmake --version
ccmake --version
ctest --version
cpack --version
```

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild CMake again.

Command Explanations

- `rm -rf cmake-...` and `tar -xf`: Recreate a clean CMake source tree.
- `CC=/system/binaries/cc CXX=/system/binaries/c++ ./bootstrap`: Bootstraps CMake with the target C and C++ compilers.
- `--prefix=/system` and directory options: Install CMake into the book's layout.
- `--parallel="$LWI_MAKE_JOBS"`: Uses the configured job count during bootstrap.
- `--no-qt-gui`: Skips the Qt GUI frontend.
- `--system-curl` and related system options: Prefer already installed target libraries where selected by the page.
- `ninja $LWI_MAKE_FLAGS` and `ninja install`: Build and install CMake with Ninja.
- `cmake --version`, `ccmake --version`, `ctest --version`, and `cpack --version`: Verify the installed CMake tool suite.

8.30. zlib-ng stage 2 2.3.3

zlib-ng provides zlib-compatible compression and decompression libraries

Open standalone page

> **Input assumption:** `zlib-ng-2.3.3.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- zlib License

Dependencies:

- musl (libc)
- cmake
- ninja (samurai)

zlib-ng is a deflate compression library with a zlib-compatible API mode. This is the final build of zlib-ng.

Extract and Enter the Source Tree

```
cd "$LBI_SOURCES"
tar -xf zlib-ng-2.3.3.tar.gz
cd zlib-ng-2.3.3
```

Configure zlib-ng

```
lbi_cmake build \
    -DCMAKE_C_COMPILER="clang" \
    -DCMAKE_C_FLAGS="$LWI_CFLAGS" \
    -DCMAKE_SHARED_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DCMAKE_EXE_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DZLIB_COMPAT=ON \
    -DBUILD_TESTING=OFF \
    -DINSTALL_UTILS=ON \
    -G Ninja
```

Build zlib-ng

```
cmake --build build $LWI_MAKE_FLAGS
```

Install zlib-ng

```
cmake --install build
```

Remove static library

```
rm -f /system/libraries/libz.a
```

Command Explanations

- `cd "$LBI_SOURCES"`, `tar -xf`, and `cd zlib-ng-...`: Enter the staged zlib-ng source tree.
- `lbi_cmake build`: Configures zlib-ng with the book's CMake helper.
- `-DCMAKE_C_COMPILER="clang"`: Builds with the final target Clang.
- `-DCMAKE_C_FLAGS`, shared-linker flags, and exe-linker flags: Apply local compile and link tuning.
- `-DZLIB_COMPAT=ON`: Builds zlib-compatible headers and library names.
- `-DBUILD_TESTING=OFF`: Skips tests in this target build.
- `-DINSTALL_UTILS=ON`: Installs zlib-ng utilities in the final target environment.
- `cmake --build` and `cmake --install`: Build and install the configured tree.
- `rm -f /system/libraries/libz.a`: Removes the static zlib archive so shared linking remains the default.

8.31. om4 stage 2 6.7

om4 provides a compact m4 macro processor in the target userspace, using its custom configure flow with only supported options.

Open standalone page

> **Input assumption:** `om4-6.7.tar.gz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-3-Clause
- ISC

Dependencies:

- musl (libc)
- yacc
- lex (flex)

om4 is an OpenBSD-derived implementation of the m4 macro processor. we need it to provide a small, predictable `m4` binary for package build flows in later chapters.

Extract and Enter the Source Tree

```
cd "/sources"
tar -xf om4-6.7.tar.gz
cd om4-6.7
```

Configure om4 (Supported Flags Only)

This package uses a custom `configure` script and does **not** support the full flag set used by `lbi_configure`.

```
CC="clang" \
CFLAGS="$LWI_CFLAGS" \
LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
./configure \
    --prefix=/system \
    --bindir=/system/binaries \
    --mandir=/system/documentation/man-pages \
    --enable-m4
```

Apply the Parser Compatibility Fix

Upstream `parser.y` calls `exit(1)` but does not include `<stdlib.h>`. Add it before building so modern Clang modes do not fail with an undeclared-function error.

```
grep -q '^#include <stdlib.h>$' parser.y || {
    awk '
        {
            print
            if (!done && $0 == "#include <stdint.h>") {
                print "#include <stdlib.h>"
                done = 1
            }
        }
    ' parser.y > parser.y.tmp &&
    mv parser.y.tmp parser.y
}
```

Generate the Tokenizer Source

```
lex -t tokenizer.l > tokenizer.c
```

Build om4

```
make -j1 CC="clang"
```

Install om4

```
make CC="clang" install
```

Command Explanations

- `cd /sources`, `tar -xf`, and `cd om4-...`: Enter the staged om4 source tree.
- `CC="clang"`, `CFLAGS`, and `LDFLAGS`: Configure om4 with the final target compiler and local tuning variables.
- `./configure --prefix=/system ... --enable-m4`: Uses om4's custom configure script and installs the command as `m4`.
- `grep -q ... || awk ...`: Adds the missing `<stdlib.h>` include only when it is absent.
- `lex -t tokenizer.l > tokenizer.c`: Regenerates the tokenizer source from the lexer input.
- `make -j1 CC="clang"`: Builds serially to avoid generated-source races.

- `make CC="clang" install`: Installs om4 into the configured target paths.

8.32 libarchive stage 2 3.8.7

8.32. libarchive stage 2 3.8.7

libarchive provides archive libraries and BSD archive tools, including bsdtar, bsdcpio, and bsdunzip.

Open standalone page

> **Input assumption:** `libarchive-3.8.7.tar.xz` is already present in `LBI_SOURCES` from the chapter 4 source staging step.

Licenses:

- BSD-2-Clause (primary)
- BSD-3-Clause, public domain, and other permissive notices in selected files

Dependencies:

- musl (libc)
- cmake
- make
- zlib-ng (libz)

libarchive is a multi-format archive reading and writing library plus BSD archive utilities. we need it to provide `tar`, `cpio`, and `unzip` command compatibility.

Extract and Enter the Source Tree

```
cd "/sources"
tar -xf libarchive-3.8.7.tar.xz
cd libarchive-3.8.7
```

Configure libarchive (CMake Path)

```
lbi_cmake build \
    -DCMAKE_C_FLAGS="$LWI_CFLAGS" \
    -DCMAKE_SHARED_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DCMAKE_EXE_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DBUILD_SHARED_LIBS=ON \
    -DENABLE_TAR=ON \
```

```
    -DENABLE_CPIO=ON \
    -DENABLE_UNZIP=ON \
    -DENABLE_CAT=OFF \
    -DENABLE_ZLIB=ON \
    -DENABLE_OPENSSL=ON \
    -DENABLE_BZip2=OFF \
    -DENABLE_LZMA=OFF \
    -DENABLE_ZSTD=OFF \
    -DENABLE_LZ4=OFF \
    -DENABLE_LZO=OFF \
    -DENABLE_LIBB2=OFF \
    -DENABLE_LIBXML2=OFF \
    -DENABLE_EXPAT=OFF \
    -DENABLE_ICONV=OFF \
    -DENABLE_ACL=OFF \
    -DENABLE_XATTR=OFF \
    -DENABLE_TEST=OFF \
    -G Ninja
```

Build libarchive

```
cmake --build build $LWI_MAKE_FLAGS
```

Install libarchive

```
cmake --install build
```

Normalize Installed Paths to LBI Layout

> **Path note:** upstream `libarchive` CMake install rules hardcode some destinations (`bin`, `include`, `share/man`). Move those outputs into the book's layout after install.

```
if [ -d "/system/bin" ]; then
    mv "/system/bin/"* "/system/binaries/" 2>/dev/null || true
    rmdir "/system/bin" 2>/dev/null || true
fi

if [ -d "/system/include" ]; then
    mv "/system/include/"* "/system/headers/" 2>/dev/null || true
```

```
    rmdir "/system/include" 2>/dev/null || true
fi

for sec in man1 man3 man5; do
    if [ -d "/system/share/man/$sec" ]; then
        mv "/system/share/man/$sec/"* \
            "/system/documentation/man-pages/$sec/" 2>/dev/null || true
        rmdir "/system/share/man/$sec" 2>/dev/null || true
    fi
done
```

Remove static library

```
rm -f "/system/libraries/libarchive.a"
```

Provide `tar`, `cpio`, and `unzip` Commands

```
ln -sf bsdtar "/system/binaries/tar"
ln -sf bsdcpio "/system/binaries/cpio"
ln -sf bsdunzip "/system/binaries/unzip"
```

Command Explanations

- `cd /sources`, `tar -xf`, and `cd libarchive-...`: Enter the staged libarchive source tree.
- `lbi_cmake build`: Configures libarchive with the book's CMake helper.
- `-DBUILD_SHARED_LIBS=ON`: Builds shared libarchive libraries.
- `-DENABLE_TAR`, `-DENABLE_CPIO`, and `-DENABLE_UNZIP`: Build the BSD archive command-line tools.
- `-DENABLE_ZLIB`, `-DENABLE_LZMA`, and `-DENABLE_ZSTD`: Enable selected compression backends available in the final target.
- `cmake --build build $LWI_MAKE_FLAGS` and `cmake --install build`: Build and install libarchive.
- `mv /system/bin/*` and `mv /system/include/*`: Normalize upstream install directories into `/system/binaries` and `/system/headers`.
- `rm -f /system/libraries/libarchive.a`: Removes the static archive after installing the shared library.
- `ln -sf bsdtar`, `bsdcpio`, and `bsdunzip`: Provide standard `tar`, `cpio`, and `unzip` command names.

8.33. GNU Make stage 2 4.4.1

Rebuild GNU Make in the final target environment so target package builds use the finalized `make` and `gmake` binaries.

Open standalone page

> **Input assumption:** `make-4.4.1.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://gnu.mirror.constant.com/make/make-4.4.1.tar.gz`
>
> **Upstream build note:** GNU Make release tarballs ship a generated `configure` script and follow the standard `./configure`, `make`, `make install` build flow.

Licenses:

- GPL-3.0-or-later

Dependencies:

- musl (libc)
- clang
- make

GNU Make is a Makefile build orchestration tool. we need it to provide the finalized target `make` and `gmake` commands for chapter 8 and later native package builds.

Extract and Enter the Source Tree

```
cd /sources
rm -rf make-4.4.1
tar -xf make-4.4.1.tar.gz
cd make-4.4.1
```

Configure GNU Make

```
CC=clang \
CFLAGS="$LWI_CFLAGS" \
LDFLAGS="$LBI_CUSTOM_LDFLAGS" \
lbi_configure \
```

```
    --without-guile \
    --disable-nls
```

Build GNU Make

```
make $LWI_MAKE_FLAGS
```

Install GNU Make

```
make install
ln -sf make /system/binaries/gmake
```

Command Explanations

- `rm -rf make-...` and `tar -xf`: Recreate a clean GNU Make source tree.
- `CC=clang`, `CFLAGS`, and `LDFLAGS`: Configure with the final target compiler and local tuning variables.
- `lbi_configure`: Applies the book's `/system` install layout.
- `--without-guile`: Disables optional Guile integration.
- `--disable-nls`: Avoids gettext/native language support for this build.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install GNU Make.
- `ln -sf make /system/binaries/gmake`: Provides the common `gmake` alias.

8.34 LLVM final 22.1.3

8.34. LLVM final 22.1.3

Build and install the final native LLVM, Clang, LLD, compiler-rt, libunwind, libcxxabi, and libcxx toolchain inside the target chroot.

Open standalone page

> **Input assumption:** `llvm-project-22.1.3.src.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> ## Licenses
>
> - Apache-2.0 WITH LLVM-exception
>
> ## Dependencies
>
> - musl (libc)

- cmake
- samurai or ninja
- zlib-ng (libz)
- zstd
- clang
- lld
- compiler-rt
- libunwind
- libcxxabi
- libcxx

LLVM is a modular compiler infrastructure project. we need it to provide the final native compiler backend, linker integration, binary utilities, and toolchain libraries for the target system.

Clang is a C and C++ frontend for LLVM. we need it to provide the final `clang`, `clang++`, `cc`, and `c++` compilers used by the completed system.

LLD is LLVM's linker. we need it to provide the final `ld.lld` linker and the default `ld` compatibility entry point.

compiler-rt is LLVM's runtime support library project. we need it to provide Clang builtins, CRT objects, and low-level compiler runtime support.

libunwind is a stack unwinding library. we need it to provide unwind support used by C++ exception handling.

libcxxabi is the LLVM C++ ABI support library. we need it to provide ABI and exception runtime support for libcxx.

libcxx is the LLVM C++ standard library implementation. we need it to provide the final C++ standard library used by Clang.

Extract Sources and Enter the LLVM Build Root

```
cd /sources
rm -rf llvm-project-22.1.3.src
tar -xf llvm-project-22.1.3.src.tar.xz
cd llvm-project-22.1.3.src/llvm
```

Configure LLVM, Clang, and LLD

This pass builds the final native compiler and linker. The runtimes are built in separate standalone steps afterward so that `compiler-rt`, `libunwind`, `libcxxabi`, and `libcxx` land in the book's custom filesystem layout predictably.

```
lbi_cmake build-final \
    -G Ninja \
    -DCMAKE_C_COMPILER=clang \
    -DCMAKE_CXX_COMPILER=clang++ \
    -DCMAKE_C_FLAGS="$LWI_CFLAGS -fPIC" \
    -DCMAKE_CXX_FLAGS="$LWI_CXXFLAGS -fPIC" \
    -DCMAKE_EXE_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DCMAKE_SHARED_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DLLVM_ENABLE_PROJECTS="clang;lld" \
    -DLLVM_INSTALL_BINUTILS_SYMLINKS=ON \
    -DLLVM_TARGETS_TO_BUILD="host" \
    -DCLANG_DEFAULT_CXX_STDLIB=libc++ \
    -DCLANG_DEFAULT_LINKER=lld \
    -DCLANG_DEFAULT_RTLIB=compiler-rt \
    -DCLANG_DEFAULT_UNWINDLIB=none \
    -DLLVM_ENABLE_ZLIB=ON \
    -DLLVM_ENABLE_ZSTD=ON \
    -DLLVM_ENABLE_LIBXML2=OFF \
    -DLLVM_INCLUDE_TESTS=OFF \
    -DLLVM_BUILD_TESTS=OFF \
    -DLLVM_INCLUDE_DOCS=OFF \
    -DLLVM_BUILD_LLVM_DYLIB=ON \
    -DLLVM_LINK_LLVM_DYLIB=ON \
    -DLLVM_ENABLE_RTTI=ON \
    -DCLANG_CONFIG_FILE_SYSTEM_DIR=/system/configuration/clang \
    -DLLVM_INCLUDE_BENCHMARKS=OFF \
    -DCMAKE_BUILD_TYPE=Release
```

Build LLVM, Clang, and LLD

```
cmake --build build-final $LWI_MAKE_FLAGS
```

Install LLVM, Clang, and LLD

```
cmake --install build-final
```

Add Final Clang Driver Wrapper Defaults

The final system uses a non-FHS layout. These wrappers make Clang find the target headers, startup objects, compiler-rt files, and C++ runtime libraries without every later package needing custom include and linker flags.

```
mv /system/binaries/clang /system/binaries/clang.real

mv /system/binaries/clang++ /system/binaries/clang++.real



cat > /system/binaries/clang <<EOF
#!/bin/sh
exec /system/binaries/clang.real \\
    --target="$LBI_TARGET" \\
    -isystem /system/headers \\
    -B/system/libraries \\
    -B/system/libraries/clang/22/lib/linux \\
    -B/system/libraries/clang/22/lib/linux \\
    -L/system/libraries \\
    -Wno-unused-command-line-argument \\
    "\$@"
EOF

cat > /system/binaries/clang++ <<EOF
#!/bin/sh
exec /system/binaries/clang++.real \\
    --target="$LBI_TARGET" \\
    -nostdinc++ \\
    -I/system/headers/c++/v1 \\
    -isystem /system/libraries/clang/22/include \\
    -isystem /system/libraries/clang/22/include \\
    -isystem /system/headers \\
    -B/system/libraries \\
    -B/system/libraries/clang/22/lib/linux \\
    -B/system/libraries/clang/22/lib/linux \\
    -L/system/libraries \\
    -Wl,-rpath,/system/libraries \\
    -Wno-unused-command-line-argument \\
    "\$@" \\
    -lc++ \\
    -lc++abi \\
```

```
    -lunwind
EOF

chmod 755 /system/binaries/clang /system/binaries/clang++

cd /system/binaries
ln -sf clang cc
ln -sf clang++ c++
ln -sf clang "$LBI_TARGET-clang"
ln -sf clang++ "$LBI_TARGET-clang++"
ln -sf clang "$LBI_TARGET-cc"
ln -sf clang++ "$LBI_TARGET-c++"
```

Build and Install compiler-rt Builtins and CRT Objects

:::pull-note **Compiler-rt note:** this section builds only the builtins and CRT objects. Sanitizers, libFuzzer, profiling, XRay, ORC, and MemProf are disabled because this book stage only needs the baseline runtime support required by the compiler driver.

```
cd /sources/llvm-project-22.1.3.src/compiler-rt

lbi_cmake build-compiler-rt-final \
    -G Ninja \
    -DCMAKE_C_COMPILER=clang \
    -DCMAKE_CXX_COMPILER=clang++ \
    -DCMAKE_ASM_COMPILER=clang \
    -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY \
    -DCMAKE_C_FLAGS="$LWI_CFLAGS" \
    -DCMAKE_CXX_FLAGS="$LWI_CXXFLAGS" \
    -DCMAKE_ASM_FLAGS="$LWI_CFLAGS" \
    -DCMAKE_EXE_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DCOMPILER_RT_INSTALL_PATH=/system/lib/clang/22 \
    -DCOMPILER_RT_BUILD_BUILTINS=ON \
    -DCOMPILER_RT_BUILD_CRT=ON \
    -DCOMPILER_RT_BUILD_LIBFUZZER=OFF \
    -DCOMPILER_RT_BUILD_MEMPROF=OFF \
    -DCOMPILER_RT_BUILD_ORC=OFF \
    -DCOMPILER_RT_BUILD_PROFILE=OFF \
    -DCOMPILER_RT_BUILD_CTX_PROFILE=OFF \
    -DCOMPILER_RT_BUILD_SANITIZERS=OFF \
```

```
    -DCOMPILER_RT_BUILD_XRAY=OFF \
    -DCOMPILER_RT_INCLUDE_TESTS=OFF \
    -DLLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF \
    -DCMAKE_BUILD_TYPE=Release

cmake --build build-compiler-rt-final --target builtins $LWI_MAKE_FLAGS
cmake --build build-compiler-rt-final --target crt $LWI_MAKE_FLAGS
cmake --install build-compiler-rt-final
```

Build and Install LLVM C++ Runtimes

```
cd /sources/llvm-project-22.1.3.src/runtimes

lbi_cmake build-runtimes-final \
    -G Ninja \
    -DCMAKE_C_COMPILER=clang.real \
    -DCMAKE_CXX_COMPILER=clang++.real \
    -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY \
    -DCMAKE_C_FLAGS="$LWI_CFLAGS" \
    -DCMAKE_CXX_FLAGS="$LWI_CXXFLAGS" \
    -DCMAKE_EXE_LINKER_FLAGS="$LBI_CUSTOM_LDFLAGS" \
    -DLLVM_ENABLE_RUNTIMES="libunwind;libcxxabi;libcxx" \
    -DLLVM_ENABLE_LIBXML2=OFF \
    -DLLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF \
    -DLLVM_INCLUDE_TESTS=OFF \
    -DLLVM_INCLUDE_DOCS=OFF \
    -DLIBUNWIND_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBUNWIND_INCLUDE_TESTS=OFF \
    -DLIBUNWIND_INCLUDE_DOCS=OFF \
    -DLIBCXXABI_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBCXXABI_INCLUDE_TESTS=OFF \
    -DLIBCXX_INSTALL_LIBRARY_DIR=/system/libraries \
    -DLIBCXX_INCLUDE_TESTS=OFF \
    -DLIBCXX_INCLUDE_BENCHMARKS=OFF \
    -DLIBCXX_INCLUDE_DOCS=OFF \
    -DLIBCXX_HAS_MUSL_LIBC=ON \
    -DLIBCXX_HAS_ATOMIC_LIB=OFF \
    -DLIBCXXABI_HAS_CXA_THREAD_ATEXIT_IMPL=OFF \
    -DLIBCXXABI_USE_LLVM_UNWINDER=ON \
    -DLIBCXX_USE_COMPILER_RT=ON \
    -DLIBCXXABI_USE_COMPILER_RT=ON \
```

```
    -DLIBUNWIND_USE_COMPILER_RT=ON \
    -DCMAKE_BUILD_TYPE=Release

cmake --build build-runtimes-final $LWI_MAKE_FLAGS
cmake --install build-runtimes-final
```

Add libgcc Compatibility Links

LLVM's `llvm-libgcc` runtime exists for distributions that replace GCC runtime pieces with compiler-rt and libunwind while still needing the traditional `libgcc.a`, `libgcc_eh.a`, and `libgcc_s.so` names. The final system already built compiler-rt builtins and libunwind separately, so this step installs a small helper that creates those compatibility names in `/system/libraries`.

```
cat > /system/binaries/lbi-refresh-libgcc-links <<'EOF'
#!/bin/sh
set -eu

libdir=/system/libraries
arch=${LBI_ARCH:-$(uname -m)}

case "$arch" in
    x86_64|amd64) compiler_rt_arch=x86_64 ;;
    i?86) compiler_rt_arch=i386 ;;
    aarch64|arm64) compiler_rt_arch=aarch64 ;;
    *) compiler_rt_arch=$arch ;;
esac

builtins=$({ find "$libdir/clang" /system/lib/clang \
    -type f -name "libclang_rt.builtins-${compiler_rt_arch}.a" 2>/dev/null || tr

unwind_static=$({ find "$libdir" \
    -maxdepth 1 -type f -name 'libunwind.a' 2>/dev/null || true; } | head -n1)

if [ -e "$libdir/libunwind.so" ]; then
    unwind_shared=$libdir/libunwind.so
else
    unwind_shared=$({ find "$libdir" \
        -maxdepth 1 \( -type f -o -type l \) -name 'libunwind.so*' 2>/dev/null |
fi

if [ -z "$builtins" ]; then
```

```
    echo "libclang_rt.builtins archive for $compiler_rt_arch was not found" >&2
    exit 1
fi

if [ -z "$unwind_static" ]; then
    echo "libunwind.a was not found" >&2
    exit 1
fi

if [ -z "$unwind_shared" ]; then
    echo "libunwind.so was not found" >&2
    exit 1
fi

ln -sf "$builtins" "$libdir/libgcc.a"
ln -sf "$(basename "$unwind_static")" "$libdir/libgcc_eh.a"
ln -sf "$(basename "$unwind_shared")" "$libdir/libgcc_s.so.1.0"
ln -sf libgcc_s.so.1.0 "$libdir/libgcc_s.so.1"
ln -sf libgcc_s.so.1 "$libdir/libgcc_s.so"
EOF

chmod 755 /system/binaries/lbi-refresh-libgcc-links
/system/binaries/lbi-refresh-libgcc-links
```

Normalize Clang Runtime Layout

Some LLVM runtime builds install files under `/system/lib/clang`, while the book's final compiler wrapper and library layout use `/system/libraries`. Create compatibility links for the resource files and CRT objects.

```
mkdir -p /system/libraries/clang/22/lib/linux
mkdir -p /system/lib/clang/22/lib/linux

if [ -d /system/lib/clang/22 ]; then
    cp -R /system/lib/clang/22/* /system/libraries/clang/22/ 2>/dev/null || true
fi

if [ -d /system/libraries/clang/22 ]; then
    cp -R /system/libraries/clang/22/* /system/lib/clang/22/ 2>/dev/null || true
fi
```

```
CRTBEGIN_OBJ=$(find /system/libraries/clang /system/lib/clang \
    -type f \( -name 'crtbeginS.o' -o -name 'clang_rt.crtbegin*.o' \) 2>/dev/nul

CRTEND_OBJ=$(find /system/libraries/clang /system/lib/clang \
    -type f \( -name 'crtendS.o' -o -name 'clang_rt.crtend*.o' \) 2>/dev/null | |

if [ -n "$CRTBEGIN_OBJ" ] && [ -n "$CRTEND_OBJ" ]; then
    CRT_DIR=$(dirname "$CRTBEGIN_OBJ")

    ln -sf "$(basename "$CRTBEGIN_OBJ")" "$CRT_DIR/crtbeginS.o"
    ln -sf "$(basename "$CRTEND_OBJ")" "$CRT_DIR/crtendS.o"

    ln -sf "$CRTBEGIN_OBJ" /system/libraries/crtbeginS.o
    ln -sf "$CRTEND_OBJ" /system/libraries/crtendS.o
fi
```

Verify the Final Toolchain

Check the compiler driver versions.

```
clang --version
clang++ --version
ld.lld --version
```

Check C compilation and linking.

```
cat > /tmp/lbi-c-test.c <<'EOF'
#include <stdio.h>

int main(void)
{
    puts("c ok");
    return 0;
}
EOF

cc /tmp/lbi-c-test.c -o /tmp/lbi-c-test
/tmp/lbi-c-test
```

Check C++ header ordering, libc++ wrapper headers, exception runtime support, and the final C++ link line.

```
cat > /tmp/lbi-cxx-test.cpp <<'EOF'
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <memory>
#include <new>

struct test_value {
    int value;
};

int main()
{
    auto item = std::make_unique<test_value>();
    item->value = 123;
    std::cout << item->value << std::endl;
    return errno;
}
EOF

c++ /tmp/lbi-cxx-test.cpp -o /tmp/lbi-cxx-test
/tmp/lbi-cxx-test
```

The C++ test should print:

```
123
```

Command Explanations

- `rm -rf llvm-project-...` and `tar -xf`: Recreate a clean LLVM source tree.
- `lbi_cmake build-final -G Ninja`: Configures the final LLVM build with the book's layout and Ninja generator.
- `-DCMAKE_C_COMPILER=clang` and `-DCMAKE_CXX_COMPILER=clang++`: Builds with the current target compiler wrappers.
- `-DCMAKE_C_FLAGS` and `-DCMAKE_CXX_FLAGS`: Apply local tuning and position-independent code where required.
- `-DLLVM_ENABLE_PROJECTS`, `LLVM_TARGETS_TO_BUILD`, and default Clang runtime options: Select the final compiler, linker, target, and runtime defaults.

- `cmake --build build-final $LWI_MAKE_FLAGS` and `cmake --install build-final`: Build and install the final LLVM toolchain.
- `mv clang clang.real` and `cat > clang`: Preserve real compiler binaries and install wrapper scripts with the book's default include/library paths.
- `lbi_cmake build-compiler-rt-final`: Builds compiler-rt builtins and CRT objects against the final compiler.
- `lbi_cmake build-runtimes-final`: Rebuilds libunwind, libcxxabi, and libcxx for the final compiler/runtime layout.
- `cat > /system/binaries/lbi-refresh-libgcc-links`: Installs a helper that refreshes libgcc-compatible links to compiler-rt outputs.
- `cp -R /system/lib/clang/22/* ...`: Mirrors Clang resource files between expected resource-directory layouts.
- `clang --version`, compile tests, and `readelf`: Verify the final C and C++ compiler drivers and dynamic interpreter behavior.

8.35 rustc 1.95.0

8.35. rustc 1.95.0

Build and install the final Rust compiler and Cargo toolchain inside the target chroot.

Open standalone page

> **Input assumption:** `rustc-1.95.0-src.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://static.rust-lang.org/dist/rustc-1.95.0-src.tar.xz`
>
> **Upstream build note:** the 1.95.0 source tree uses `bootstrap.toml` as the primary bootstrap configuration file. Upstream's bootstrap documentation describes `./x.py install` as the source-install command, and the local `src/etc/xhelp` documents `-j, --jobs` for parallelism.
>
> **musl target note:**
> `compiler/rustc_target/src/spec/targets/x86_64_unknown_linux_musl.rs` still sets `base.crt_static_default = true` in this release. This section patches that default to `false` so the installed musl target produces dynamically linked binaries unless a build explicitly asks for static CRT linkage.

Licenses:

- Apache-2.0
- MIT

Dependencies:

- musl (libc)
- clang
- lld
- LLVM
- compiler-rt
- libunwind
- libcxxabi
- libcxx
- python
- cmake
- samurai or ninja
- curl
- pkgconf
- LibreSSL
- zlib-ng

rustc is a compiler for the Rust programming language. we need it to provide the final target `rustc` compiler, standard library, and Rust documentation support for packages that build Rust code.

cargo is Rust's package manager and build driver. we need it to build Rust packages and drive Rust dependency builds in the final target environment.

Extract and Enter the Source Tree

```
cd /sources
rm -rf rustc-1.95.0-src
tar -xf rustc-1.95.0-src.tar.xz
cd rustc-1.95.0-src
```

Patch the musl Target Default

```
cat > /tmp/rustc-1.95.0-musl-dynamic-link.patch <<'EOF'
--- a/compiler/rustc_target/src/spec/targets/x86_64_unknown_linux_musl.rs
+++ b/compiler/rustc_target/src/spec/targets/x86_64_unknown_linux_musl.rs
@@ -18,6 +18,6 @@ pub(crate) fn target() -> Target {
```

```
     base.supports_xray = true;
     // FIXME(compiler-team#422): musl targets should be dynamically linked by d
-    base.crt_static_default = true;
+    base.crt_static_default = false;

     Target {
         llvm_target: "x86_64-unknown-linux-musl".into(),
EOF

patch -Np1 -i /tmp/rustc-1.95.0-musl-dynamic-link.patch
```

Create bootstrap.toml

```
cat > bootstrap.toml <<EOF
change-id = 148671

[build]
build = "x86_64-unknown-linux-musl"
host = ["x86_64-unknown-linux-musl"]
target = ["x86_64-unknown-linux-musl"]
vendor = true
submodules = false
extended = true
tools = ["cargo", "rustdoc"]
jobs = $LWI_MAKE_JOBS

[install]
prefix = "/system"
sysconfdir = "/system/configuration"
docdir = "documentation/rust"
bindir = "binaries"
libdir = "libraries"
mandir = "documentation/man-pages"
datadir = "share"

[llvm]
download-ci-llvm = false
use-libcxx = true

[rust]
channel = "stable"
```

```
download-rustc = false
default-linker = "/system/binaries/cc"
lld = false
bootstrap-override-lld = false
rpath = false
jemalloc = false
codegen-tests = false
llvm-tools = false

[target.x86_64-unknown-linux-musl]
cc = "/system/binaries/cc"
cxx = "/system/binaries/c++"
linker = "/system/binaries/cc"
ar = "/system/binaries/llvm-ar"
ranlib = "/system/binaries/llvm-ranlib"
llvm-config = "/system/binaries/llvm-config"
llvm-has-rust-patches = false
crt-static = false
EOF
```

Build and Install rustc

```
./x.py install -j "$LWI_MAKE_JOBS"
```

Verify rustc

```
rustc --version
cargo --version
rustdoc --version
```

Check that the default musl target now links dynamically.

```
cat > /tmp/lbi-rust-test.rs <<'EOF'
fn main() {
    println!("rust ok");
}
EOF

rustc /tmp/lbi-rust-test.rs -o /tmp/lbi-rust-test
```

```
/tmp/lbi-rust-test
readelf -l /tmp/lbi-rust-test | grep 'Requesting program interpreter'
```

The Rust test should print:

```
rust ok
```

The `readelf` command should show the musl dynamic loader instead of reporting no program interpreter.

> After this step is complete, you can remove the extracted source directory and source tarball from `/sources` if you do not plan to rebuild rustc again.

Command Explanations

- `rm -rf rustc-...` and `tar -xf`: Recreate a clean Rust source tree.
- `cat > /tmp/rustc-...patch`: Writes a small local patch that changes the musl target default away from static CRT linking.
- `patch -Np1 -i /tmp/rustc-...patch`: Applies that target-spec patch to the Rust source tree.
- `cat > bootstrap.toml`: Writes Rust's build configuration for the target host and target.
- `vendor = true` and `submodules = false`: Use vendored dependencies without trying to update submodules.
- `download-ci-llvm = false` and `download-rustc = false`: Build using local toolchain inputs instead of downloading prebuilt components.
- `./x.py install -j "$LWI_MAKE_JOBS"`: Builds and installs Rust using the configured job count.
- `rustc --version`, `cargo --version`, and `rustdoc --version`: Verify the installed Rust tools.
- `rustc /tmp/lbi-rust-test.rs ...`: Builds and runs a small Rust smoke test.
- `readelf -l ... | grep`: Confirms the produced binary requests the expected dynamic interpreter.

8.36 uutils-coreutils 0.8.0

8.36. uutils-coreutils 0.8.0

Build and install the final Rust core utilities after the target Rust toolchain is available.

Open standalone page

> **Input assumption:** `coreutils-0.8.0.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:**
> `https://github.com/uutils/coreutils/archive/refs/tags/0.8.0.tar.gz`

Licenses:

- MIT

Dependencies:

- musl (libc)
- clang
- make
- ca-certificates
- rustc
- cargo

uutils-coreutils is a Rust reimplementation of the GNU core utilities. we need it to provide the final target command set that replaces the temporary early-userland utility coverage after Rust is installed.

Extract and Enter the Source Tree

```
cd /sources
rm -rf coreutils-0.8.0
tar -xf coreutils-0.8.0.tar.gz
cd coreutils-0.8.0
```

Build uutils-coreutils

> **Build option note:** `MANPAGES=n` and `COMPLETIONS=n` avoid the `uudoc` documentation branches, whose install paths do not match the book's manpage and completion layout. `MULTICALL=y` installs one `coreutils` binary with hardlinked applet names, replacing the temporary utility implementations in `/system/binaries`.

```
CARGO_BUILD_JOBS="$LWI_MAKE_JOBS" \
CARGO_BUILD_TARGET="$LBI_ARCH-unknown-linux-musl" \
SSL_CERT_FILE=/system/configuration/ssl/cert.pem \
CARGO_HTTP_CAINFO=/system/configuration/ssl/cert.pem \
```

```
make $LWI_MAKE_FLAGS \
    PROFILE=release \
    MULTICALL=y \
    MANPAGES=n \
    COMPLETIONS=n \
    LOCALES=y \
    PREFIX=/system \
    BINDIR=/system/binaries \
    DATAROOTDIR=/system/share \
    CARGOFLAGS="--locked"
```

Install uutils-coreutils

```
CARGO_BUILD_JOBS="$LWI_MAKE_JOBS" \
CARGO_BUILD_TARGET="$LBI_ARCH-unknown-linux-musl" \
SSL_CERT_FILE=/system/configuration/ssl/cert.pem \
CARGO_HTTP_CAINFO=/system/configuration/ssl/cert.pem \
make install \
    PROFILE=release \
    MULTICALL=y \
    MANPAGES=n \
    COMPLETIONS=n \
    LOCALES=y \
    PREFIX=/system \
    BINDIR=/system/binaries \
    DATAROOTDIR=/system/share \
    CARGOFLAGS="--locked"
```

Verify uutils-coreutils

```
coreutils --help >/dev/null
cat --version
test --help >/dev/null
[ --help >/dev/null
```

> After this step is complete, the temporary `sbase` tools are still useful history for bootstrapping, but the final command names in `/system/binaries` now come from uutils-coreutils.

Command Explanations

- `rm -rf coreutils-...` and `tar -xf`: Recreate a clean uutils-coreutils source tree.
- `CARGO_BUILD_JOBS="$LWI_MAKE_JOBS"`: Uses the shared job count for Cargo builds.
- `CARGO_BUILD_TARGET="$LBI_ARCH-unknown-linux-musl"`: Builds for the musl target triple matching the selected architecture.
- `SSL_CERT_FILE` and `CARGO_HTTP_CAINFO`: Point Cargo and TLS users at the installed CA bundle.
- `make $LWI_MAKE_FLAGS PROFILE=release MULTICALL=y MANPAGES=n`: Builds the release multicall binary without generating man pages.
- `make install ... PREFIX=/system`: Installs coreutils into the book's layout.
- `coreutils --help`, `cat --version`, `test --help`, and `[ --help`: Smoke-test the multicall binary and important applets.

8.37 red 1.0.2

8.37. red 1.0.2

Build and install red as the final Rust sed-compatible stream editor.

Open standalone page

> **Input assumption:** `red-v1.0.2.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:**
> `https://github.com/vyavdoshenko/red/archive/refs/tags/v1.0.2.tar.gz`
>
> **Upstream compatibility note:** upstream describes red as an experimental drop-in replacement for GNU sed and says it currently achieves full functional compatibility with GNU sed behavior. This section installs it as the final `sed` command after the Rust toolchain is available.

Licenses:

- MIT

Dependencies:

- musl (libc)
- clang
- ca-certificates
- rustc

- cargo

red is a Rust implementation of a GNU sed-compatible stream editor. we need it to provide the final target `sed` command for source edits, package build scripts, and routine text-processing work after the Rust toolchain is installed.

Extract and Enter the Source Tree

```
cd /sources
rm -rf red-1.0.2
tar -xf red-v1.0.2.tar.gz
cd red-1.0.2/red
```

Build red

red is a Cargo project and does not ship a Makefile, configure script, or install target. Build the locked release crate directly.

```
SSL_CERT_FILE=/system/configuration/ssl/cert.pem \
CARGO_HTTP_CAINFO=/system/configuration/ssl/cert.pem \
cargo build \
    --release \
    --locked \
    --no-default-features \
    --target "$LBI_ARCH-unknown-linux-musl"
```

Install red

Install the binary as `red`, remove the earlier `sed` implementation, then replace it with a symlink to `red`.

```
install -Dm755 "target/$LBI_ARCH-unknown-linux-musl/release/red" \
    /system/binaries/red

rm -f /system/binaries/sed
rm -f /system/documentation/man-pages/sed.1
ln -sf red /system/binaries/sed
```

Verify red

```
red --version
sed --version
```

```
printf 'one two\n' | sed 's/two/three/'
```

The final command should print:

```
one three
```

> After this step is complete, the earlier `bsdsed` package remains part of the bootstrap path, but `/system/binaries/sed` now resolves to red.

Command Explanations

- `rm -rf red-...` and `tar -xf`: Recreate a clean red source tree.
- `SSL_CERT_FILE` and `CARGO_HTTP_CAINFO`: Point Cargo's TLS stack at the installed CA bundle.
- `cargo build --release --locked --no-default-features`: Builds a reproducible release binary without default optional features.
- `--target "$LBI_ARCH-unknown-linux-musl"`: Builds for the selected musl architecture target.
- `install -Dm755 ... /system/binaries/red`: Installs the red binary into the target command directory.
- `rm -f /system/binaries/sed` and `ln -sf red /system/binaries/sed`: Replaces the earlier sed provider with red.
- `red --version`, `sed --version`, and the substitution pipeline: Verify both command names and basic sed behavior.

8.38 dinit 0.21.0

8.38. dinit 0.21.0

Build and install dinit as the target system service manager and init-capable supervisor.

Open standalone page

> **Input assumption:** `dinit-0.21.0.tar.xz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/davmac314/dinit/releases/download/v0.21.0/dinit-0.21.0.tar.xz`
>
> **Upstream build note:** upstream documents dinit as a C++11 project built with GNU make, a C++ compiler, POSIX utilities, and `m4` for generating manual pages. The release ships a custom

> `configure` script, not an autotools one, and it accepts only its own directory options; this section calls it directly instead of using `lbi_configure`.

Licenses:

- Apache License 2.0

Dependencies:

- musl (libc)
- clang
- make
- om4

dinit is a service supervisor and init-capable dependency manager. we need it to provide the target system's service manager, service control tools, and shutdown command set.

Extract and Enter the Source Tree

```
cd /sources
rm -rf dinit-0.21.0
tar -xf dinit-0.21.0.tar.xz
cd dinit-0.21.0
```

Configure dinit

> `--disable-capabilities` avoids a dependency on libcap, which is not part of the base package set. The remaining Linux service-management features are enabled explicitly so the generated `mconfig` records the intended target behavior.

```
./configure \
    --prefix=/system \
    --exec-prefix=/system \
    --bindir=/system/binaries \
    --sbindir=/system/systembinaries \
    --mandir=/system/documentation/man-pages \
    --syscontrolsocket=/run/dinitctl \
    --disable-capabilities \
    --enable-cgroups \
    --enable-ioprio \
```

```
    --enable-oom-adj \
    CXX=clang++
```

Build dinit

```
make $LWI_MAKE_FLAGS
```

Install dinit

```
make install
```

Verify dinit

```
dinit --version
dinitctl --help >/dev/null
dinit-check --help >/dev/null
shutdown --help >/dev/null
```

> This installs `dinit`, `dinitctl`, `dinit-check`, and `dinit-monitor` in `/system/binaries`; `shutdown`, `halt`, `reboot`, `soft-reboot`, and `poweroff` in `/system/systembinaries`; and the generated manual pages in `/system/documentation/man-pages`.

Command Explanations

- `rm -rf dinit-...` and `tar -xf`: Recreate a clean dinit source tree.
- `./configure`: Uses dinit's custom configure script directly.
- `--prefix`, `--exec-prefix`, `--bindir`, `--sbindir`, and `--mandir`: Install dinit into the book's `/system` layout.
- `--syscontrolsocket=/run/dinitctl`: Sets the runtime control socket location.
- `--disable-capabilities`: Avoids Linux capability library integration for this build.
- `make $LWI_MAKE_FLAGS` and `make install`: Build and install dinit.
- `dinit --version` and the `--help` commands: Verify the installed service manager tools are runnable.

9.1. Introduction

Chapter 9 explains how the finished system becomes a running Linux installation, from firmware handoff through dinit service supervision.

Open standalone page

> **Goal:** understand the boot path well enough to configure the system deliberately instead of treating startup as a pile of magic files.

The previous chapters built the operating system as a filesystem tree. Chapter 9 is about making that tree wake up as a machine: the kernel must be loaded, the root filesystem must become the active system root, runtime filesystems must appear, and a real init process must take responsibility for the rest of userspace.

Boot is mostly a handoff sequence. Each stage does only enough work to start the next stage, and the first long-running userspace process becomes the parent and coordinator for everything that follows.

The Boot Chain

A typical Linux boot begins in firmware. On modern machines that usually means UEFI. The firmware initializes enough hardware to read the EFI System Partition, then starts a bootloader or a directly bootable EFI program.

The bootloader chooses a kernel, passes a kernel command line, and may also load an initramfs. The command line tells the kernel important early facts, such as where the root filesystem is, whether it should initially be mounted read-only, and which program should be used as init if the default is not correct.

The kernel then takes over. It initializes CPU support, memory management, device discovery, filesystems, and driver infrastructure. At this stage there is no normal userspace yet. The kernel is building the environment that will allow userspace to exist.

If an initramfs is used, the kernel unpacks it into memory and starts its early userspace program first. That early userspace can load modules, unlock encrypted storage, assemble RAID or logical volumes, discover the real root filesystem, and then switch into it. A simple system can boot without an initramfs if the kernel already has everything it needs to find and mount the real root filesystem.

Once the real root filesystem is available, the kernel starts process ID 1. By default it looks for common init paths such as `/sbin/init`, but this can be overridden with the `init=` kernel command-line option. From this point on, the kernel is running the system, but init is running userspace.

Early Mounts

The root filesystem is only the beginning. Several important parts of a live Linux system are not ordinary on-disk directories. They are kernel-provided or memory-backed filesystems that must be mounted during early userspace startup.

`/proc` exposes process and kernel state. Tools such as `ps`, service supervisors, shells, and many scripts rely on it to inspect running processes and system information.

`/sys` exposes the kernel device model. It is the main structured view of devices, drivers, buses, block devices, network interfaces, power state, and many tunable kernel attributes.

`/dev` provides device nodes. On a static system it can contain pre-created nodes, but practical Linux systems usually mount a device filesystem there so device nodes can appear dynamically as the kernel discovers hardware.

`/run` is a small runtime state filesystem, normally backed by memory. It holds sockets, pid files, lock files, and other state that should not survive a reboot. dinit's system control socket lives here by default as `/run/dinitctl`.

Other mounts are added as the system becomes more complete. `devpts` supplies pseudo-terminals under `/dev/pts`. `tmpfs` mounts may provide `/tmp` or other volatile scratch locations. Real filesystems for home directories, boot partitions, removable media, or separate system areas are mounted according to local policy.

The important pattern is that init does not merely start daemons. It also establishes the shape of the running system: the root is checked or remounted as needed, kernel filesystems are mounted, runtime directories are prepared, and only then do higher-level services have a stable place to stand.

What dinit Is

dinit is the init system and service manager used by this book's target system. It can run as process ID 1, which makes it the first normal userspace process and the ancestor of system services. It can also supervise services, track dependencies between them, and provide a control interface through `dinitctl`.

dinit starts from service descriptions. A service description says what kind of service is being managed, what command starts it, what must start before it, what should wait for it, and how dinit should treat failures. Some services represent long-running processes. Others represent one-shot setup work, internal milestones, or scripted state changes.

This dependency model matters during boot. Instead of one long startup script where every line must be in exactly the right place, dinit can express relationships directly. A login service can wait for mounted filesystems. A network service can depend on device setup. A final boot target can collect the services that define a usable system.

dinit also remains useful after boot. `dinitctl` can start, stop, restart, enable, disable, and inspect services through the control socket. Because dinit keeps service state in memory, it knows whether a service is starting, started, stopping, failed, or waiting on another dependency.

Chapter Direction

Chapter 9 turns the completed package set into a bootable system. The work ahead is configuration rather than compilation: arrange early mounts, define the initial service graph, connect dinit to the system's chosen init path, and make sure the result is understandable enough to repair from a shell when something goes wrong.

The goal is not to hide boot behind a framework. The goal is to make the machine's startup path explicit, small enough to reason about, and sturdy enough to trust.

9.2. Limine 11.4.1 binary release

Install Limine's bootloader payloads and build its host control utility from the binary-release snapshot.

Open standalone page

> **Input assumption:** `limine-5be26a73d7b7.tar.gz` is already present in `/sources` from the chapter 4 source staging step.
>
> **Source URL:** `https://github.com/Limine-Bootloader/Limine/commit/5be26a73d7b7b4d4477d18be94e1d16e615adf56`
>
> **Source note:** this is the upstream `v11.4.1-binary` snapshot. Unlike the full `limine-11.4.1.tar.xz` source release, it already contains the Limine boot payloads, so this section does not need `nasm`, `mtools`, or Limine's autotools configure path.

Licenses:

- BSD 2-Clause

Dependencies:

- musl (libc)
- clang
- make

limine is a modern bootloader with UEFI and BIOS support. we need it to provide the boot payloads and host-side `limine` utility used to make the target system bootable.

Extract and Enter the Source Tree

```
cd /sources
rm -rf limine-5be26a73d7b7
```

```
tar -xf limine-5be26a73d7b7.tar.gz
cd limine-5be26a73d7b7
```

Clear Build Flags

Limine's binary-release Makefile is intentionally small and provides its own default `CFLAGS`. Clear inherited build variables before compiling the host utility so target-package flags do not leak into this build.

```
unset CFLAGS CPPFLAGS LDFLAGS LIBS
unset CFLAGS_FOR_TARGET CPPFLAGS_FOR_TARGET LDFLAGS_FOR_TARGET
unset NASMFLAGS_FOR_TARGET
```

Build the Limine Utility

```
make $LWI_MAKE_FLAGS CC=clang
```

Install Limine

Install the host utility, the license, and the boot payloads. The files in `/system/share/limine` are not installed to the EFI System Partition yet; later sections will copy the needed payloads into the chosen boot location.

```
install -Dm755 limine /system/binaries/limine
install -Dm644 LICENSE /system/documentation/limine/LICENSE

install -d /system/share/limine
for file in \
    BOOTAA64.EFI \
    BOOTIA32.EFI \
    BOOTLOONGARCH64.EFI \
    BOOTRISCV64.EFI \
    BOOTX64.EFI \
    limine-bios-cd.bin \
    limine-bios-pxe.bin \
    limine-bios.sys \
    limine-uefi-cd.bin
do
    install -m644 "$file" /system/share/limine/
done
```

Verify Limine

```
limine --version
test -f /system/share/limine/BOOTX64.EFI
test -f /system/share/limine/limine-bios.sys
```

The version command should report Limine 11.4.1.

> The installed files are staged for bootloader deployment. A later section will create the Limine configuration and copy the selected EFI payload, usually `BOOTX64.EFI` on x86-64 UEFI systems, to the EFI System Partition.

Command Explanations

- `rm -rf limine-...` and `tar -xf`: Recreate a clean Limine source tree.
- `unset CFLAGS CPPFLAGS LDFLAGS LIBS` and related variables: Clear target package flags that can confuse Limine's firmware-oriented build.
- `make $LWI_MAKE_FLAGS CC=clang`: Builds Limine with Clang and the shared make parallelism setting.
- `install -Dm755 limine`: Installs the Limine command-line tool.
- `install -Dm644 LICENSE`: Installs the license text into the documentation tree.
- `install -d /system/share/limine`: Creates the directory for Limine bootloader support files.
- `for file in ...; do install -Dm644 ...; done`: Installs BIOS and EFI bootloader payload files used by later boot setup.
- `limine --version` and `test -f ...`: Verify the command and required boot files were installed.

9.3 dinit service setup

9.3. dinit service setup

Create the first dinit service descriptions so the target system can mount its runtime filesystems and present a local console login.

Open standalone page

> **Goal:** install a minimal dinit service tree with a `boot` target, early runtime mounts, a writable root transition, and a supervised `tty1` login prompt.

> **Upstream service note:** dinit looks for a service named `boot` by default, reads plain-text service description files, and uses `/etc/dinit.d` as the default system service directory.

This section creates the smallest useful service graph for the first boot. It is not a full distribution policy yet. It is the point where the machine can become interactive under dinit supervision, which makes later boot services easier to add and debug from the system itself.

The login prompt uses `getty` from ubase. That package is already part of the book, is permissively licensed, and provides a local Linux getty that opens a tty, prompts for a login name, and then invokes `/bin/login`.

Create the service directories

```
install -d /system/configuration/dinit.d/scripts
```

The `/etc` compatibility link points at `/system/configuration`, so these files appear to dinit at `/etc/dinit.d`.

Create the boot service

The `boot` service is an internal grouping service. dinit starts it by default, and its dependencies pull in the actual boot work.

```
cat > /system/configuration/dinit.d/boot <<'EOF'
type = internal
depends-on: early-filesystems
depends-on: rootrw
depends-on: loginready
depends-ms: tty1
EOF
```

`tty1` is a milestone dependency because a login prompt should be started during boot, but the whole boot target should not be torn down just because a getty process later exits and is restarted.

Create the early filesystem script

This script mounts the kernel-backed filesystems needed by the rest of early userspace. If an initramfs or kernel setup already mounted one of them, the script leaves it alone.

```
cat > /system/configuration/dinit.d/scripts/early-filesystems <<'EOF'
#!/system/binaries/sh
set -eu
```

```
mounted() {
    [ -r /system/processes/mounts ] || return 1

    while read -r device target type options rest
    do
        [ "$target" = "$1" ] && return 0
    done < /system/processes/mounts

    return 1
}

mkdir -p /proc /sys /dev /run

mounted /proc || mount -t proc proc /system/processes/
mounted /sys || mount -t sysfs sysfs /system/system
mounted /dev || mount -t devtmpfs devtmpfs /system/devices

mkdir -p /dev/pts /dev/shm /run

mounted /dev/pts || mount -t devpts devpts /system/devices/pts -o gid=5,mode=062
mounted /run || mount -t tmpfs tmpfs /system/run

if [ -h /dev/shm ]; then
    install -d -m 1777 /system/run/shm
else
    mounted /dev/shm || mount -t tmpfs -o nosuid,nodev tmpfs /system/devices/shm
fi
EOF

chmod 755 /system/configuration/dinit.d/scripts/early-filesystems
```

Create the early filesystem service

```
cat > /system/configuration/dinit.d/early-filesystems <<'EOF'
type = scripted
command = /system/binaries/sh /system/configuration/dinit.d/scripts/early-filesy
restart = false
options: starts-on-console
EOF
```

`starts-on-console` keeps early mount failures visible during boot, then releases the console once the script finishes.

Create the writable-root service

```
cat > /system/configuration/dinit.d/rootrw <<'EOF'
type = scripted
command = /system/binaries/mount -o remount,rw /
restart = false
depends-on: early-filesystems
options: starts-on-console starts-rwfs
EOF
```

The `starts-rwfs` option tells dinit that writable filesystems are now available. With this book's dinit build, that also gives dinit another opportunity to create its control socket at `/run/dinitctl`.

Create the login readiness service

```
cat > /system/configuration/dinit.d/loginready <<'EOF'
type = internal
depends-on: rootrw
options: runs-on-console
EOF
```

This service marks the point where the system can accept local logins. Holding the console here also keeps dinit's own service-status output from competing with the login prompt after startup reaches this point.

Create the tty1 getty service

```
cat > /system/configuration/dinit.d/tty1 <<'EOF'
type = process
command = /system/binaries/getty /devices/tty1 linux
restart = true
depends-on: loginready
EOF
```

ubase `getty` expects an absolute tty path. It then runs `/bin/login`, which resolves through the book's `/bin` compatibility link to `/system/binaries/login`.

Install init compatibility links

```
ln -sf ../binaries/dinit /system/systembinaries/init
ln -sf ../binaries/dinit /system/systembinaries/dinit
```

These links make both `/sbin/init` and `/sbin/dinit` resolve to the installed dinit binary through the book's `/sbin -> /system/systembinaries` compatibility link.

Check the service tree

```
dinit-check --services-dir /system/configuration/dinit.d boot
```

The check should finish without syntax errors or dependency cycles.

> This setup intentionally starts only one login prompt. Additional virtual terminals can be added later by creating `tty2`, `tty3`, and similar service descriptions and adding them to `boot` as milestone dependencies.

Command Explanations

- `install -d /system/configuration/dinit.d/scripts`: Creates the service directory and script subdirectory for dinit.
- `cat > /system/configuration/dinit.d/boot`: Writes the top-level boot target service.
- `cat > .../scripts/early-filesystems`: Writes the helper script that mounts or verifies early virtual filesystems.
- `mounted()`: Checks `/system/processes/mounts` so the script avoids remounting filesystems that are already present.
- `cat > .../early-filesystems`: Defines the scripted dinit service that runs the early filesystem helper.
- `cat > .../rootrw`: Defines the service that remounts `/` read-write after early filesystems exist.
- `cat > .../loginready` and `cat > .../tty1`: Define the login readiness target and first getty process.
- `ln -sf ../binaries/dinit /system/systembinaries/init`: Provides the conventional `/system/systembinaries/init` entry point for the boot process.
- `dinit-check --services-dir ... boot`: Validates the generated service graph before rebooting into it.

Back to first page