Linux Academy

Study Guide

Build Your Own Linux from Scratch

Contents

Section 1

Section 2

Section 3

Section 4

Section 5

Section 6

Section 1

Our Goal

What We are Building

This course walks through the creation of a 64-bit system based on the Linux kernel. Our goal is to produce a small, sleek system well-suited for hosting containers or being employed as a virtual machine.

Because we don't need every piece of functionality under the sun, we're not going to include every piece of software you might find in a typical distro. This distribution is intended to be minimal.

Here is what our end-result will look like:

- 64-bit Linux 4.8 Kernel with GCC 6.2 and glibc 2.24
- A system compatible with both EFI and BIOS hardware
- Bootable with GRUB2
- A VFAT formatted partition for GRUB/UEFI
- A boot partition
- A root partition

What We are Learning

This course provides step-by-step instructions in an effort to build the Linux kernel, the GNU C Standard Library implementation, GCC, and user land binaries from source. The tasks are presented in linear order, and must be followed sequentially, as later tasks have dependencies on early tasks. Do not skip around.

Following this guide as intended will, in turn, enlighten you to many of the "hows" and "whys" of Linux, and assist in your ability to do tasks such as:

- Troubleshooting issues with the kernel
- Troubleshooting issues with user-land software
- Understanding the rationale behind various security systems and measures
- Performance tuning the kernel
- Performance tuning user-land binaries
- Building or "rolling" your own distribution
- Building user-land binaries from source

Required Skills and Knowledge

We make extensive use of VirtualBox in this course. Working knowledge of VirtualBox and a solid foundation in Linux and Linux troubleshooting are essential. If you're not as familiar with VirtualBox as you would like, take a look at the "How to Install CentOS 7 with VirtualBox" lesson in the "Linux Essentials Certification" course. That course, as well, provides the foundational knowledge required for this course.

Standards

As we progress through this course, we will adhere to the FHS (Filesystem Hierarchy Standard) specification, version 3.0. We will adhere (mostly) to the LSB (Linux Standard Base) specification, version 5.0. See the pertinent sections in this guide for more information on these two topics.

Filesystem Hierarchy Standard

We follow the FHS 3.0 specification in this course. The FHS provides guidance as to how the filesystem should be structured in terms of directory structure, partition location, and directory use.

FHS 3.0 specifies four major file categories:

- Static OR variable
- Shareable OR unshareable

It would seem there are two categories above; however, there are not. "Static" and "variable" represent two mutually-exclusive categories, as do "shareable" and "unshareable."

A file must fit into one of these four categories; that is, it must be static or variable, shareable or unsharable or some combination thereof. All files fall into two of the four categories, without exception.

The following directories are required in the primary (or root) hierarchy; their use is as noted.

- **`bin`** • Essential binaries
- **`boot`** • Static boot-related files
- **`dev`** • Device files
- **`etc`** • Host-specific system configuration.
- **`lib`** • Essential shared libraries and kernel modules
- **`media`** • Mount point for removable media
- **`mnt`** • Mount point for temporarily mounting a filesystem

- **opt** • Add-on software
- **run** • Data relevant to running processes; `/var/run` is used more frequently
- **sbin** • Essential system binaries
- **srv** • Data for services provided
- **tmp** • Temporary files
- **usr** • Secondary hierarchy; identical to primary (root) hierarchy
- **var** • Variable (non+static) data

User home directories are located in `/usr/home`, which is linked to `/home`. This standard also specifies in detail which binaries are required; more information regarding this may be found at: http://refspecs.linuxfoundation.org/

Linux Standard Base

We follow the LSB Core Specification for the 64-bit x86 platform, as outlined at: https://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-AMD64/LSB-Core-AMD64/book1.html

The LSB standard is expansive, and outside of the scope of this course. Adherence and deviance from the standard will be pointed out in the course where we feel it is important to do so.

A Word on Linux

"Linux" as a term refers to two things: First, it refers specifically to the Linux kernel. Second, in a broader sense, it refers to the various packagings of the Linux kernel with other programs to provide the functionality required of a complete operating system.

Sound strange? It's not — it's one of the things that makes Linux so versatile. The kernel itself manages the hardware, memory, and the other parts of a computer system which are typically opaque to installed programs.

Programs installed to provide additional functionality are referred to as "user-land" or "the user-land." The combination of kernel and user-land constitute what are referred to as "distributions," many of which we are familiar: Red Hat Enterprise Linux, Ubuntu, Arch, Fedora, and so on.

In a broad sense, the term "Linux" refers to the operating systems created by the pairing of kernel and user-land, but the term is ambiguous. "Distribution," on the other hand, refers to the pairing of the kernel with a user-land built to some specification. Ubuntu, for instance, varies quite a bit from CentOS 6. Both of these are separate distributions of Linux.

Unlike operating systems, which are built in a monolithic fashion (where the user-land and kernel are

tightly-coupled, such as FreeBSD, VMS, Windows, etc.), Linux allows for variations on theme which number into the thousands. The term "distro" better fits these variations, as each one is not entirely unique from the next (because of the shared kernel) but may differ substantially in terms of the user-land.

Section 2

Prerequisites: Build System Specifications

Kernel/Distro Version

We use Fedora Core 24 in this course. You can run any distribution with a 4.x kernel, but be aware that doing so may introduce inconsistencies into the build process. Some distributions may may use older or incompatible versions of the development tools.

Clean Install

We strongly recommend that you use a clean install for the build system.

Notice we undertake the whole of this course in a virtual machine running in VirtualBox; we do this to facilitate the building of Linux in a clean environment.

Any virtualization environment will do, provided you have access to the console, as it may be necessary at various points. Actual hardware is also acceptable, if those resources are readily available to you.

Build System Disk Partitions

The build system — the virtual machine we use to build Linux — uses the following disk layout. This output is from the `parted print all` command:

```
Number  Start   End     Size    Type     File system     Flags
1       1049kB  525MB   524MB   primary  ext4            boot
2       525MB   19.3GB  18.8GB  primary  ext4
3       19.3GB  21.5GB  2147MB  primary  linux-swap(v1)
```

Whilst we walk through the creation of the destination drive (where our newly-built distribution will be installed) in the videos, you should have the proficiency to install Linux and the necessary tools prior to undertaking this course. It is strongly recommended that you take the "Linux Essentials" course on LinuxAcademy.com if your Linux skills are not quite at this level.

Development Tools

We'll need GCC, binutils, and other software packages installed with the "development tools" package group. You can select this during the installation process, or you can install using the `group` install option for `yum` or `dnf`. Note that the package group names may differ depending on distribution, but generally, we will need the "development tools" and "C development tools" groups installed.

Listing Package Groups with dnf

```
dnf group list -v
```

Listing Package Groups with yum

```
yum grouplist hidden
```

Installing Package Groups with dnf

```
dnf group install "C Development Tools and Libraries"
dnf group install "Development Tools"
```

Installing Package Groups with yum

```
yum groupinstall "Development Tools"
```

Texinfo

You may also have to install the `texinfo` package to obtain the `make info` functionality:

```
dnf install texinfo
```

MS-DOS Tools

Our first partition needs to be formatted as FAT12 or FAT16 to enable interoperability with GRUB and EFI. For this reason, you need the "dosfstools" package, or similar, installed on your system.

Specific Software Packages and Required Versions

The `nano` editor is used throughout this course; amend commands with your preferred text editor as needed.

Some of the software packages installed require a specific or minimum version. Most often, the most recent version (with patches) will be sufficient.

- bash (`/bin/sh` must be a link to the bash binary)
- binutils

- bison (`/usr/bin/yacc` must be linked to or provided by bison)
- bzip2
- coreutils
- diffutils
- findutils
- gawk (`/usr/sbin/awk` must be a link to the gawk binary)
- gcc 6.2
- glibc 2.24
- grep
- gzip
- Linux kernel 4.x
- m4
- make
- patch
- perl 5.24
- sed
- tar
- texinfo
- xz

yacc and Bison

IMPORTANT

Note that Fedora Core 24 installs `byacc` by default. This means `yacc` is not a link to the bison executable. You can verify this with the following command:

```
rpm -qf `which yacc`
```

If the `yacc` binary is installed by the `byacc` package, execute the following:

Remove the byacc package:

```
dnf erase byacc
```

Re-install bison

```
dnf reinstall bison
```

Link the bison binary:

```
ln -s `which bison` /bin/yacc
```

Call `yacc -V` and make certain the output matches `bison -V`.

Hardware/Virtual Hardware Specifications

CPUs

Many of the packages we compile in this course can benefit from parallel make processes. The `make` command can, in many cases, compile multiple source files simultaneously, provided the build system has more than one CPU.

It is strongly recommended than you allocate at least two CPUs if your build system is virtualized. If you are using hardware, ideally you will have at least two physical CPU cores to speed up the build process.

In particular, build and test times for software like GCC can be reduced a great deal by having `make` execute processes in parallel. In places where you see `-j2` indicated with the `make` command, it is perfectly acceptable to substitute a higher number, up to the number of CPUs allocated to the VM (or, if using hardware, the number of physical cores in the system). So, provided you have four CPU cores available, you are welcome to use `-j4` instead of `-j2`.

RAM

Virtualized or otherwise, you will need at least two gigabytes of *free* RAM available. Do not include swap space in this consideration. Builds may fail if swap use becomes extant.

Build System Disks

In addition to to the system disk, the build system needs an additional block storage device available. We recommend attaching a 20 gigabyte drive to the second port of the SAS or SATA controller of your virtual machine. If using hardware, this device can be a USB drive, a second hard, etc.; however, it must be a local block device physically attached to the system.

Users, Groups, and More

Don't Use root Unless Needed

The most important rule of this course parallels a general principle in *NIX in general: Do *nothing* as the superuser unless it is required.

Particularly when compiling code, it is very possible to crash your system or damage your installation. It is possible to even damage hardware in many cases. Beyond this, if you compile as *root*, you may find it next to impossible to work through the various steps of a process using anything but the *root* account. This sets us up for failure: Forget to set a single environment variable, and you've overwritten critical system libraries or binaries on the build system, only to find no binaries can be executed and the system cannot reboot.

At some stages of the build process, we must use the *root* account to execute commands. This will be noted when necessary.

Creating our User

For this course, we create a user and group to execute our builds. Make sure the user is part of *wheel* group (or equivalent).

First, add our *byol* group:

```
groupadd byol
```

Be sure to add the `-k` flag to the `user add` command to prevent files being copied to the home directory from `/etc/skeleton`:

```
useradd -s /bin/bash -g byol -m -k /dev/null byol
```

We do this to prevent environment variables that might otherwise be appropriate for a user account from being set in our build environment, as these can have unintended consequences.

Setting the Login Environment for the byol User

There should be two bash-related files in the home directory of the *byol* user: `.bash_profile` and `.bashrc`.

The `.bash_profile` file needs to have the following content:

```
exec env -i HOME=$HOME TERM=$TERM PS1='\u:\w\$ ' /bin/bash
```

While the `.bashrc` file needs to have the following content:

```
set +h

umask 022

LC_ALL=POSIX

PATH=/tools/bin:/bin:/usr/bin

export LC_ALL PATH
```

These files are read during login, and set up the shell environment for the user. We set the environment here specifically to avoid inheriting environment variables that might have an adverse effect on our efforts.

After altering these files, upon log out and log in, the output from the `env` command should return output similar to the following:

```
TERM=xterm-256color

LC_ALL=POSIX

PATH=/tools/bin:/bin:/usr/bin

PWD=/home/byol

PS1=\u:\w\$

SHLVL=1

HOME=/home/byol

_=/bin/env
```

Destination Disk

Once you've configured your virtual machine, add an additional disk to it. 20 gigabytes should be sufficient; the disk type doesn't matter, but attaching to the SAS or SATA controller is recommended. We will refer to

this newly attached disk as the "destination disk" from this point forward.

Partitioning the Destination Disk

By the conclusion of this section, the destination disk partition layout should look like this:

```
Number  Start   End     Size    File system     Name     Flags
 1      1049kB  105MB   104MB   fat16           primary  bios_grub
 2      105MB   551MB   446MB   ext4            primary
 3      551MB   19.3GB  18.7GB  ext4            primary
 4      19.3GB  21.5GB  2174MB  linux-swap(v1)  primary
```

The first partition begins at 1MB and extends to 104MB. This partition exists solely for the purposes of booting and is formatted as VFAT. We will refer to this partition as the *EFI/GRUB boot* partition henceforth.

The second partition begins at around 100MB and ends 400MB or so later. This is the *boot* partition.

The third partition begins at around 550MB and ends around 19GB. This is the *root* partition.

This last partition begins around 19MB and extends to the end of the disk. This is the *swap* parition.

The following command creates these partitions using `parted`:

```
parted --script /dev/sdb mklabel gpt mkpart primary 1MiB 100MiB mkpart
primary 100MiB 525MiB mkpart primary 525MiB 19.3GB mkpart primary 19.3GB
100%
```

Setting Flags on the GRUB Boot Partition

The GRUB boot partition needs to have certain flags set so that BIOS-based systems can boot from it. We accomplish this using `parted`:

```
parted --script /dev/sdb set 1 bios_grub on
```

Creating Filesystems on the Destination Disk

Our partition table on our destination block storage device should look like this:

```
Number  Start   End     Size    File system     Name     Flags
 1      1049kB  105MB   104MB   fat16           primary  bios_grub
 2      105MB   551MB   446MB   ext4            primary
 3      551MB   19.3GB  18.7GB  ext4            primary
 4      19.3GB  21.5GB  2174MB  linux-swap(v1)  primary
```

Partition 1

```
mkfs.vfat /dev/sdb1
```

Partitions 2 and 3

The second and third partitions are our boot and root partitions. We will format these as "vanilla" ext4, using the `mkfs.ext4` command; note that your device name may be different:

```
mkfs.ext4 /dev/sdb2
mkfs.ext4 /dev/sdb3
```

Partition 4

The last partition is our swap partition, which we will initialize using the `mkswap` command:

```
mkswap /dev/sdb4
```

Setting Partition and Filesystem Metadata

We're going to set some volume metadata on our filesystems using the `tune2fs` command.

Set the volume label of our second partition on our destination device to *DESTBOOT*:

```
tune2fs -c 1 -L DESTBOOT /dev/sdb2
```

Set the volume label of our third partition to *DESTROOT*:

```
tune2fs -c 1 -L DESTROOT /dev/sdb3
```

Note that we're setting the mount count to `1`, which means the consistency of these filesystems will be checked after they have been mounted once. This is optional, but can be helpful in the event you have to reset your VM or build system and the filesystems aren't cleanly unmounted.

Sanity Check

At this point, you should have:

- A complete build system with all of the necessary software installed to build Linux. Ideally, this system has two or more CPUs and two gigabytes or more of *free* RAM. Do **not** include available swap as free RAM.
- A *byol* user and group. This user should have access to root privileges via `su` or `sudo`.

- A second block storage device mounted on the build system, ideally on the same interface/ controller as the system drive. This is our destination drive and should be partitioned and formatted as described above.

Measure twice, cut once: This is a good time to review all of the sections previous to this one and ensure your build system and destination disk are setup as recommended.

Mount Point(s)

Destination Device: Root and Boot Directories

We're going to mount our destination disk on `/build`, as follows, using the superuser account:

```
mkdir -v /build
export BROOT=/build
mount -t ext4 -L DESTROOT $BROOT
mkdir -v $BROOT/boot
mount -t ext4 -L DESTBOOT $BROOT/boot
```

Adding Our Swap Partition

```
swapon -v /dev/sdb4
```

Exporting the BROOT Environment Variable

We're also going to add an export line to the `.bashrc` file of the *byol* account to export the `BROOT` variable. Our `.bashrc` file should look like this:

```
set +h
umask 022
LC_ALL=POSIX
PATH=/tools/bin:/bin:/usr/bin
BROOT=/build
export BROOT LC_ALL PATH
```

Sanity Check

Measure twice, cut once: you should have the root and boot destination filesystems mounted under `/build`. Also, your swap partition should be active. We can check these with the `mount` and `swapon` commands:

```
mount | grep 'sdb'
```

This should return a listing similar to:

```
/dev/sdb3 on /build type ext4 (rw,relatime,data=ordered)
/dev/sdb2 on /build/boot type ext4 (rw,relatime,data=ordered)
```

`swapon | grep sdb` should output similar:

```
/dev/sdb4 partition   2G   0B   -2
```

Keep in mind that your device names may be different.

Building the Toolchain

We're now ready to build the requisite toolchain needed to build Linux from scratch.

Directories and Directory Permissions

First, let's create a directory in our destination root to hold the source files:

```
mkdir -v $BROOT/source
```

We want the `/usr/src/toolchain` directory to be sticky, which means that only the owner of a file can delete it. We also want the files in this directory to be writable for all users:

```
chmod -v 777 $BROOT/source
```

Additionally, we want to create a directory to contain our compiled tools, which we need to keep aside and apart from the same binaries on the build system:

```
mkdir -v $BROOT/tools
```

We want to link this directory to the build system. The following command looks somewhat strange, but is correct:

```
ln -sv $BROOT/tools /
```

The result should be a symlink from `/tools` in the build system root directory to the `tools` directory on the root of our destination device:

```
ln -sv $BROOT/tools /
'/tools' → '/build/tools'
```

Source Code Files

A list of the software we need to install can be found at: http://linuxfromscratch.org/lfs/view/stable/wget-list.

Change into the `$BROOT/source directory` and download all of these files using two `wget` commands:

```
wget http://linuxfromscratch.org/lfs/view/stable/wget-list
wget --input-file=wget-list --no-check-certificate --directory-
prefix=$BROOT/source
```

We do not want the source code files located in our tools directory.

Decompress and Extract the Downloaded Files

All of our source code files will have been compressed with `zx`, `bzip2`, or `gzip`. We can decompress these files by running the following three commands:

```
bunzip2 -d *.bz2
gunzip *.gz
xz -d *.xz
```

Change Ownership of the Build Directory

Execute:

```
chown -v byol:byol $BROOT
```

Patch Files

You'll notice that some of the files listed in the above TXT file are patch files. These patches need to be applied to the source before we compile the relevant package; we touch on this where necessary.

A Word on Errors

Both binutils and GCC must compile without errors. If either built has errors, start over. Depending on the error type you encounter this might mean rebuilding just GCC or rebuilding binutils as well.

Setting the Target Triplet

GCC and the rest of the build toolchain use a string called the "target triplet" to provide information as to the current CPU architecture, vendor, and operating system. The triplet itself is broken into three parts:

machine-vender-os

So We could define our target triplet as follows:

```
x86_64-elf
```

However, we need to build GCC and the binutils toolchain without any external dependencies. To do that, our first build must be targeted for cross-compilation. This will ensure that all of the necessary dependencies are included and none of the shared libraries on the build system are relied upon. So we're going to use the triplet instead of the one above:

```
x86_64-lfs-linux-gnu
```

This target triplet indicates an x86_64 compatible machine and an elf binary compatible operating system. Since we need this value set in our environment, let's add this to the `.bashrc` file of the *byol* user. That file should now look like this:

```
set +h
umask 022
LC_ALL=POSIX
PATH=/tools/bin:/bin:/usr/bin
BROOT=/build
BTARGET=x86_64-lfs-linux-gnu
export BROOT BTARGET LC_ALL PATH
```

The vendor field is optional in the target triplet; we've specified "lfs" here as this vendor code exists to facilitate building for purposes such as ours.

Target Triplets: What's the Point?

The purpose of the target triplet is to provide a means for the system and compilers to determine specific information about any given binary. After cross referencing and disambiguation, the target triplet is actually resolved to three distinct values, which may be quite different from the triplet specified above:

- **Build Platform** • Where the binary was built.
- **Host Platform** • Where the binaries are intended to run.
- **Target Platform** • In the case of compilers, this is for what the compiler will generate code.

So after disambiguation, target triplets are "resolved" into something like this:

```
x86_64-elf-gcc
```

In fact, we could specify this value as our target triplet. This would result, however, in subtle changes in

GCC's handling of code, and the outcome may not be desirable. For example, changing the vendor to the value `pc` will result in the toolchain being cross-compiled with 32-bit compatibility enabled. This, in turn, will result in a different dynamic linker being used both at compile and runtime to link and find shared libraries. If GCC or glibc are configured incorrectly, the result is often a toolchain broken in such that problems do not manifest until the build process is nearly complete.

Build Directory

For many of the builds, we `cd` into the source directory and create a `build` directory. Once changed into this directory, we proceed with the build. This is the recommended way to build the toolchain, as it keeps the original source files from being tampered with by the build process.

If a build directory is not specified, it is not necessary and won't be used.

Configuration

Once in the build directory, the `configure` script is called from the parent directory to determine the options for the tool being built. This also copies the relevant source and configuration files from the original source directory to the build directory created in the previous section.

Deleting Source Directories

For some builds (like GCC and binutils), it is perfectly fine to delete only the `build` directory after compilation and installation are executed. However, erring on the side of caution, it may be wise to delete the entire source directory to prevent misconfiguration down the road.

Section 3

Stage 1: Building a Temporary Toolchain

The first stage of the build process entails building a toolchain capable of producing executables for our target platform. Because our target platform differs (in terms of the target triplet) from the platform on which we are building, we need to first build a toolchain that is can be used in a "standalone" fashion. The ensures that we avoid any dependency on the build system environment or libraries.

Note that for Stage 1 builds, we don't execute the test suites for the software packages installed. The reason for this is simple — the dependencies required to run the tests, such as TCL, have not yet been installed, and the tests will likely fail.

Running the tests ceases to be optional once we reach Stage 3, however.

Compiling Binutils

Create the Build Directory

Change into the binutils source directory under `$BROOT/source`. Create a build directory:

```
mkdir -v build && cd build
```

Configure the Source

Call the "configure" script from within the build directory to configure binutils. We used the following parameters for configure:

```
../configure \
--prefix=/tools \
--with-sysroot=$BROOT \
--with-lib-path=/tools/lib \
--target=$BTARGET  \
--disable-nls \
--disable-werror
```

- **`prefix`** • Tells the configure script where the compiled binaries should be installed.
- **`with-sysroot`** • Tells the configure script to look in the specified directory for the target system libraries.
- **`with-lib-path`** • Specifies which path the linker will be configured to use.
- **`target`** • Because the triplet we've defined is slightly different than what will be determined by `configure` itself, the binutils source needs to be compiled to accommodate cross-linking. This is necessary because we want to ensure a "clean room" build, without any artifacts from the build system.

- **`disable-els`** • Disables internationalization, which we don't need at this stage.
- **`disable-werror`** • Keeps warnings from interrupting the compile process.

Compile the Source

```
make -j2
```

Create a Library Directory and Symlink

We need to create the `/tools/lib` directory and symlink `/tools/lib64` to it. This ensures that the proper libraries can be found along both paths:

```
case $(uname -m) in
  x86_64) mkdir -v /tools/lib && ln -sv lib /tools/lib64 ;;
esac
```

Install binutils

```
make install
```

Delete the Build Directory

Be sure to delete the `build` directory once you've run `make install`.

GCC

We are now going to build GCC. Because we're building GCC as a cross-compiler, it will not rely on libraries installed on the build system and instead prefers those packaged with it, or which we explicitly tell `configure` to look for.

Obtain the GCC Dependencies

First, we need to download the GCC dependencies. We can do that using the following commands to copy the relevant software packages into the GCC source directory. Once they're copied, `make` finds and builds these dependencies automatically. From the GCC source directory, execute:

```
tar -xf ../mpfr-3.1.4.tar
mv -v mpfr-3.1.4 mpfr
tar -xf ../gmp-6.1.1.tar
mv -v gmp-6.1.1 gmp
tar -xf ../mpc-1.0.3.tar
mv -v mpc-1.0.3 mpc
```

Note that the software versions may have changed since this guide was written.

Changing the Default Linker

We need to instruct GCC to use the linker previously installed by our binutils install. The following script (which you should be able to copy and paste to a terminal window) finds the relevant header files and changes the locations where `make` looks for certain libraries and files. Execute the following in the GCC source code directory:

```
for file in \
 $(find gcc/config -name linux64.h -o -name linux.h -o -name sysv4.h)
do
  cp -uv $file{,.orig}
  sed -e 's@/lib\(64\)\?\(32\)\?/ld@/tools&@g' \
  -e 's@/usr@/tools@g' $file.orig > $file
  echo '
#undef STANDARD_STARTFILE_PREFIX_1
#undef STANDARD_STARTFILE_PREFIX_2
#define STANDARD_STARTFILE_PREFIX_1 "/tools/lib/"
#define STANDARD_STARTFILE_PREFIX_2 ""' >> $file
  touch $file.orig
done
```

Alternatively, you can save this code as a shell script in GCC source directory and run it by calling that file as an argument to bash.

A more detailed description of what this script does, per the 'Linux From Scratch' online version:

> *"In case the above seems hard to follow, let's break it down a bit. First we find all the files under the GCC/config directory that are named either `linux.h`, `linux64.h` or `sysv4.h`. For each file found, we copy it to a file of the same name but with an added suffix of `.orig`. Then the first `sed` expression prepends `/tools` to every instance of `/lib/ld`, `/lib64/ld`, or `/lib32/ld`, while the second one replaces hard-coded instances of `/usr`. Next, we add our define statements which alter the default startfile prefix to the end of the file. Note that the trailing `/` in `/tools/lib/` is required. Finally, we use `touch` to update the timestamp on the copied files. When used in conjunction with `cp -u`, this prevents unexpected changes to the original files in case the commands are inadvertently run twice."*

Configure

Create the build directory and change into it. Run the `configure` script with the following arguments:

```
../configure                                    \
--target=$BTARGET                             \
--prefix=/tools                               \
--with-glibc-version=2.24                     \
--with-sysroot=$BROOT                           \
--with-newlib                                 \
--without-headers                             \
```

```
--with-local-prefix=/tools                          \
--with-native-system-header-dir=/tools/include      \
--disable-nls                                       \
--disable-shared                                    \
--disable-multilib                                  \
--disable-decimal-float                             \
--disable-threads                                   \
--disable-libatomic                                 \
--disable-libgomp                                   \
--disable-libmpx                                    \
--disable-libquadmath                               \
--disable-libssp                                    \
--disable-libvtv                                    \
--disable-libstdcxx                                 \
--enable-languages=c,c++
```

- **target** • Tells `make` to use the target triplet located in our environment variable.
- **prefix** • Instructs `make` to use the specified path to prefix relative pathnames.
- **with-glibc-version** • Indicates which version of glibc we're going to target.
- **with-sysroot** • Specifies the system root and allows us to specify a different root path than that of the currently-running kernel.
- **with-newlib** • Prevents any code which requires libc from being compiled (because we haven't built libc yet).
- **without-headers** • If building a full-blown cross-compiler – which are not – GCC needs the header files compatible with the target system. This argument instructs `configure` and `make` not to look for them, as we don't need them for this stage.
- **with-local-prefix** • This instructs `configure` and `make` to search the specified directory for include files.
- **with-native-system-header-dir** • This changes the default include path for headers (which is normally `/usr/include`) to `/tools/include`. Without this switch, GCC will look in the default location for include files, and that will break our build since our header files are located in `/tools/include`.
- **disable-shared** • Advises GCC to build static libraries. This is a good idea at this stage simply because it avoids any conflicts that might arise if GCC were built to use shared libraries, as the system linker might attempt to link them with the libraries installed on the build system.
- **disable-decimal** • This GCC extension is not compatible when building GCC for cross compilation.
- **disable-float** • See `disable-decimal`.
- **disable-threads** • See `disable-decimal`.
- **disable-libatomic** • See `disable-decimal`.

- **`disable-libgomp`** • See `disable-decimal`.
- **`disable-libmpx`** • See `disable-decimal`.
- **`disable-libquadmath`** • See `disable-decimal`.
- **`disable-libssp`** • See `disable-decimal`.
- **`disable-libvtv`** • See `disable-decimal`.
- **`disable-libstdcxx`** • See `disable-decimal`.
- **`disable-multilib`** • This functionality isn't supported on the x86_64 platform.
- **`enable-languages`** • For this stage, we need GCC to compile only C and C++. This disables the compilation of compilers for other languages.

Make GCC

```
make -j2
```

This will take some time.

Install

```
make install
```

Install Kernel Header Files

The Linux kernel ships with a set of header files which provide programmers a software interface to the kernel. These are used primarily by libc (or the GNU implementation, glibc).

Extracting the Kernel Header Files

Change to the kernel source directory:

```
$BROOT/source/linux-4.x
```

Issue the following command:

```
make mrproper
```

We now call the `make` command, specifying the header file output path; we do this in two steps because `make` wipes any files from the destination directory during this step. First, we extract the files:

```
make INSTALL_HDR_PATH=dest headers_install
```

And second, we copy them to the proper location for libc:

```
cp -rv dest/include/* /tools/include
```

Building glibc

Configure

Run the `configure` script with the following arguments. Note the exported variables; be sure to unset these when the build process is complete!

The `libc_cv_forced_unwind` variable impacts the handling of the `--force-unwind` configuration parameter. The linker we installed when we compiled `binutils` is cross-compiled and cannot make use of the `--force-unwind` option unless glibc is present. This variable disables this test.

The `libc_cv_c_cleanup` variables instruct configure to disable the test for this functionality, again for the same reasons given for `libc_cv_forced_unwind`.

The commands for the configuration process should look like this:

```
export libc_cv_forced_unwind=yes
export libc_cv_c_cleanup=yes
../configure                             \
  --prefix=/tools                        \
  --host=$BTARGET                        \
  --build=$(../scripts/config.guess)     \
  --enable-kernel=2.6.32                 \
  --with-headers=/tools/include
```

- **`prefix`** • Instructs `make` to use the specified path to prefix's relative pathnames.
- **`host`** • Tells `make` to use the target triplet located in our environment variable.
- **`build`** • Combined with the host flag, this instructs glibc's build system to configure itself to cross compile, using the cross-linker and compiler in `/tools`.
- **`enable-kernel`** • Instructs glibc to use workarounds for this specific version (and later) of the kernel.
- **`with-headers`** • Specifies the location of header files, so libc knows what features the kernel has and can configure itself accordingly.

Make

According to the Linux From Scratch website, glibc sometimes breaks during parallel makes. We're going to disable parallel compilation for glibc by executing make as follows:

```
make -j1
make -j1 install
```

Delete the Build Directory

Be sure to delete the `build` directory once you've run `make install`. Additionally, unset the environment variables defined during our configuration:

```
unset libc_cv_forced_unwind
unset libc_cv_c_cleanup
```

Sanity Check

At this point, it is imperative that we stop and check our temporary toolchain to ensure that it has been built correctly. The online version of the "Linux From Scratch" website provides us a neat and sure method of testing that our toolchain is installed correctly.

Testing the Toolchain

We can test the temporary toolchain to check it compiles and links code and object code correctly with the following commands:

```
echo 'int main(){}' > dummy.c
$BTARGET-gcc dummy.c
readelf -l a.out | grep ': /tools'
```

Note that this should be done as the *byol* user.

If everything is working as it should be, the `grep` command should return this output:

```
[Requesting program interpreter: /tools/lib64/ld-linux-x86-64.so.2]
```

If the output is different — particularly if the program interpreter is located on the build system's filesystem hierarchy (`lib/ld-linux-x86-64.so.2`, etc.) - then something has gone wrong.

Delete the build directories, check environment variables and start over. Continuing will result in a toolchain that is broken in odd ways which are not immediately apparent.

Be sure to delete all of the `dummy.*` files when you're done testing.

Building libstdc++

libstdc++ is Part of GCC

This software is part of the GCC sources, so we'll need to be in the `gcc` source directory, in an empty `build` directory, before executing the following.

Configure

Create the `build` directory as per usual, and run the `configure` script with the following arguments:

```
../libstdc++-v3/configure           \
--host=$BTARGET                  \
--prefix=/tools                  \
--disable-multilib               \
--disable-nls                    \
--disable-libstdcxx-threads      \
--disable-libstdcxx-pch          \
--with-gxx-include-dir=/tools/$BTARGET/include/c++/6.2.0
```

Make

Compile and install:

```
make -j2
make install
```

Section 4

Stage 2: Building with The Temporary Toolchain

At this point, we've created a temporary toolchain capable of compiling and linking executables in a stand-alone sense. Now we need to make an additional compilation pass over these same tools so they are native to our target triplet.

In a sense, Stage 2 creates a "temporary" system, using minimal installations of several familiar programs and libraries like ncurses, bash, and more.

Compiling Binutils: Native Build

Create the Build Directory

Change into the binutils source directory under `$BROOT/source`. Create a build directory:

```
mkdir -v build && cd build
```

Configure the Source

Because we want a native build of binutils, call the target-triplet-specific utilities installed in our first build of binutils:

```
export CC=$BTARGET-gcc
export AR=$BTARGET-ar
export RANLIB=$BTARGET-ranlib
```

Call the `configure` script *from within the build directory* to configure binutils. We use the following parameters for configure:

```
../configure                \
--prefix=/tools             \
--disable-nls               \
--disable-werror            \
--with-lib-path=/tools/lib \
--with-sysroot
```

- **`with-sysroot`** • Specifying no value here enables the linker to find shared libraries required by other objects. Without this flag, `make` may not be able to locate and link some required libraries.
- **`with-lib-path`** • Here we're instructing `make` to use the specified directory explicitly.

Compile the Source

```
make -j2
```

Install binutils

```
make install
```

Prepare the Linker for Upcoming Adjustments

The following commands clean out the build directory for the `ld` utility, and then rebuilds the same. We specify the `LIB_PATH` to override the default value used by the temporary toolchain — this is the default library search path.

```
make -C ld clean
make -C ld LIB_PATH=/usr/lib:/lib
cp -v ld/ld-new /tools/bin
```

Building GCC Natively

Replacing the Internal GCC limits.h Header

GCC initially uses a self-provided `limits.h` file for the building of the temporary toolchain. Now, we want GCC to use a full set of definitions written in the limits headers. To ensure this, we concatenate several of GCC's internal files to create a single `limits.h` file:

```
cat gcc/limitx.h gcc/glimits.h gcc/limity.h > dirname $($BTARGET-gcc
-print-libgcc-file-name)/include-fixed/limits.h
```

Changing the Default Linker (Again)

Once again, we want GCC to use the linker installed in our `/tools` directory:

```
for file in \
 $(find gcc/config -name linux64.h -o -name linux.h -o -name sysv4.h)
do
  cp -uv $file{,.orig}
  sed -e 's@/lib\(64\)\?\(32\)\?/ld@/tools&@g' \
  -e 's@/usr@/tools@g' $file.orig > $file
  echo '
# undef STANDARDSTARTFILEPREFIX1
# undef STANDARDSTARTFILEPREFIX2
# define STANDARDSTARTFILEPREFIX1 "/tools/lib/"
# define STANDARDSTARTFILEPREFIX2 ""' >> $file
  touch $file.orig
done
```

GCC Dependencies

These should already be in place, as this step was required for the Stage 1 build of GCC. However, if for some reason you've deleted the GCC source directory, re-trace the steps for installing the GCC dependencies as described previously.

Configure

Create the build directory and change into it. As with the build of binutils in the previous section, we define some environment variables before compiling:

```
export CC=$BTARGET-gcc                                          \
export CXX=$BTARGET-g++                                         \
export AR=$BTARGET-ar                                           \
export RANLIB=$BTARGET-ranlib                                   \
```

Run the `configure` script with the following arguments:

```
../configure                                       \
--prefix=/tools                                 \
--with-local-prefix=/tools                      \
--with-native-system-header-dir=/tools/include \
--enable-languages=c,c++                        \
--disable-libstdcxx-pch                         \
--disable-multilib                              \
--disable-bootstrap                             \
--disable-libgomp
```

Build GCC

Execute:

```
make -j2
make install
```

Create the cc Symlink

Typically, scripts call `cc` instead cf `gcc` or `clang` to compile code. This allows developers to use different compilers without having to re-edit scripts or write scripts for each compile. The behavior is accomplished by creating a symlink to the compiler executable; in our case, `cc` should be a link to the `gcc` binary:

```
ln -sv gcc /tools/bin/cc
```

Cleanup

Be sure to unset the exported variables above. You may also logout and log back in as the *byol* user to reset the environment.

Sanity Check

Once again, we need to stop and double-check our toolchain to make sure that it can compile and link executables properly.

Execute the following commands:

```
echo 'int main(){}' > dummy.c
$BTARGET-gcc dummy.c
readelf -l a.out | grep ': /tools'
```

Note that this should be done as the *byol* user.

If everything is working as it should be, the `grep` command should return this output:

```
[Requesting program interpreter: /tools/lib64/ld-linux-x86-64.so.2]
```

If the output is different — particularly if the program interpreter is located on the build system's filesystem hierarchy (`lib/ld-linux-x86-64.so.2`, etc.), then something has gone wrong.

Return to the last sanity check and re-trace your steps. Continuing will result in a broken toolchain that will likely not compile or link correctly.

Test Suite Dependencies

The test suite used for our toolchain requires a handful of dependencies. During the first and second compilation passes of binutils GCC, and Glibc, we didn't allow `make` to execute tests — as the necessary programs were missing, there was no point in doing so.

At this stage, we most certainly want to run the tests to make sure our toolchain is set up properly.

We downloaded the source code for the programs used in testing during Stage 1. Now we need to compile and install them. Unless it is otherwise noted, you'll need to extract the source code directories from the appropriate tar file in `/sources`, `cd` into it and execute the `configure` and `make` commands. It is assumed from this point forward that you have extracted the source and are in the resulting directory. Note that most of these utilities do not require the use or creation of a separate `build` directory.

Regarding make and make install

These commands can typically run one after another.

```
make && make install
```

Sometimes, however, there is good reason to separate the commands into two steps, such as when header files or libraries need to be modified before being installed. Where you see the two commands one after another (as in the following example):

```
make
make install
```

You can replace the two commands with a single line:

```
make && make install
```

Be careful, though, to use this only where the `make` and `make install` commands are one after the other, and no additional steps are required.

TCL

Configure:

```
cd unix
./configure --prefix=/tools
```

Compile:

```
make -j2
```

Install:

```
make install
```

Change Permissions

`chmod` the installed library file so we can strip out debugging symbols later:

```
chmod -v u+w /tools/lib/libtcl8.6.so
```

Install Headers

```
make install-private-headers
```

Create Required Symlink

Be sure to replace the `xxx` with the appropriate minor version of TCL in the following command:

```
ln -sv tclsh8.xxx /tools/bin/tclsh
```

Expect

First, we want to force `expect` to use `/bin/stty` to open a terminal as opposed to `/usr/local/bin/stty`; the binary in `/usr/local` is located on the build system (as we don't have a `/usr/local` hierarchy on our destination disk). We don't want `expect` to depend on that binary.

To remedy this, execute:

```
cp -v configure{,.orig}
sed 's:/usr/local/bin:/bin:' configure.orig > configure
```

This uses `sed` to replace the path used by the `configure` script.

Configure

```
./configure --prefix=/tools --with-tcl=/tools/lib --with-tclinclude=/
tools/include
```

Install

We include the `SCRIPTS` variable so `make` will skip including supplemental scripts.

```
make SCRIPTS="" install
```

DejaGNU

Configure

```
./configure --prefix=/tools
```

Compile and Install

```
make install
```

Check

Configure

We include an empty `PKG_CONFIG` variable here to prevent any pre-defined pkg-config options that may lead `make` to link against libraries on the build system.

```
PKG_CONFIG= ./configure --prefix=/tools
```

Compile and Install

```
make
make install
```

Ncurses

Ncurses is a library which provides terminal control routines. It allows for a basic graphical user interface to be used in text-only (e.g., terminal or console) environments.

Configure

We amend the source code here to ensure that the `gawk` command is found before `awk`:

```
sed -i s/mawk// configure
./configure --prefix=/tools \
            --with-shared   \
            --without-debug \
            --without-ada   \
            --enable-widec  \
            --enable-overwrite
```

Compile and Install

```
make
make install
```

Bash

Configure

```
./configure --prefix=/tools --without-bash-malloc
```

We pass the `--without-bash-malloc` parameter to disable bash's internal `malloc()`, which is known to be somewhat buggy.

Compile and Install

```
make
make install
```

Symlink sh

In many distributions, the `sh` command is actually a symlink to the bash executable; ours is no different. Create the symlink with the following command:

```
ln -sv bash /tools/bin/sh
```

bzip

The `bzip` package does not use a configure script, only `make`.

Compile

```
make
```

Install

We pass the `PREFIX` variable here to ensure the resulting binaries are installed in the `/tools` hierarchy.

```
make PREFIX=/tools install
```

Coreutils

The `coreutils` package provides many of the basic userland utilities we use in the shell to manipulate text, files and the environment.

Configure

Notice the `--enable-install-program` parameter. We pass this to `configure` to build the 'hostname' program, which is disabled by default.

```
./configure --prefix=/tools --enable-install-program=hostname
```

Compile and Install

```
make
make install
```

Diffutils, File, Findutils, and Gawk

These software packages use identical configuration and compilation/install steps. Execute these steps for each of these packages — be sure not to skip any.

Configure

```
./configure --prefix=/tools
```

Compile and Install

```
make
make install
```

Gettext

We only need three programs from the `gettext` package at this point: `msgfmt`, `msgmerge` and `xgettext`; hence the unusual `make` commands.

Configure

Note that we pass an empty "EMACS" variable to `configure`; this prevents the detection of Lisp scripts used by emacs, which is known to hang some systems. Be certain to `cd` into the `gettext-tools` directory before running `configure`.

```
cd gettext-tools
EMACS="no" ./configure --prefix=/tools --disable-shared
```

Compile the Binaries

```
make -C gnulib-lib
make -C intl pluralx.c
make -C src msgfmt
make -C src msgmerge
make -C src xgettext
```

Install

Rather than using `make` to install our binaries for us, we copy them into place manually:

```
cp -v src/{msgfmt,msgmerge,xgettext} /tools/bin
```

Grep, Gzip, and M4

These software packages use identical configuration and compilation/install steps. Execute these steps for each of these packages — be sure not to skip any.

Configure

```
./configure --prefix=/tools
```

Compile and Install

```
make
make install
```

Make

Configure

Here we pass the `--without-guile` flag to `configure`. While these libraries may be available on the build system, they are not on our destination, and we don't want `make` to depend on them for that reason.

```
./configure --prefix=/tools --without-guile
```

Compile and Install

```
make
make install
```

Patch

Configure

```
./configure --prefix=/tools
```

Compile and Install

```
make
make install
```

Perl 5

Configure

The Perl source code doesn't use the standard `configure` script. Note the capital `C` in the configuration command:

```
sh Configure -des -Dprefix=/tools -Dlibs=-lm
```

Compile

```
make
```

Install

We only need a few of the libraries we've built, so we'll copy these to our destination manually:

```
cp -v perl cpan/podlators/scripts/pod2man /tools/bin
mkdir -pv /tools/lib/perl5/5.24.0
cp -Rv lib/* /tools/lib/perl5/5.24.0
```

sed, tar, and texinfo

Configure

```
./configure --prefix=/tools
```

Compile and Install

```
make
make install
```

Util-linux

Configure

We include an empty `PKG_CONFIG` variable here to prevent any pre-defined `pkg-config` options that may lead `make` to link against libraries on the build system.

```
./configure --prefix=/tools            \
            --without-python           \
            --disable-makeinstall-chown    \
            --without-systemdsystemunitdir \
            PKG_CONFIG=""
```

- **`without-python`** • This disables the Python bindings, which we don't need at this point.
- **`disable-makeinstall-chown`** • `make` tries to change the owner of the binaries after copying

them into place. This requires root permissions, however, so we disable this.

- **`without-systemdsystemunitdir`** - This instructs `make` to skip the installation of systemd-specific files.

Compile and Install

```
make
make install
```

XZ

Configure

```
./configure --prefix=/tools
```

Compile and Install

```
make
make install
```

Stripping (Optional)

Now that we've got a temporary system built and installed, we can save some space by removing the debug symbols from our binaries and our libraries.

This command will result in quit a "File format not recognized" warning messages. This is normal — if you examine the file indicated in the message, you can note that the file in question is usually a script file, not a binary.

Please be *very* careful with this step. It is very easy to strip the wrong files and delete most of the libraries we've just built and installed.

THIS STEP IS OPTIONAL.

Strip the Debug Symbols from Our Libraries

```
strip --strip-debug /tools/lib/*
```

Strip the Debug Symbols from Our Binaries

DO NOT run this command against library files! This command uses the build system's strip binary to remove unneeded symbols. Running this against library files deletes them.

```
/usr/bin/strip --strip-unneeded /tools/{,s}bin/*
```

Switching to Root

At this point, we've used the *byol* user for almost every activity, save the creation and mounting of our destination file system.

From this point forward, however, we will undertake all commands as *root*.

Environment Check

Execute `su` and run:

```
env | grep ^B
```

Your output should look something like this:

```
BROOT=/build
BTARGET=x86_64-lfs-linux-gnu
```

While the "BROOT" environment variable is *required*, the "BTARGET" variable is not, and must *not* be set as we continue forward.

Unsetting the BTARGET Environment Variable

Unset this variable with the following command:

```
unset BTARGET
```

Check again to make sure that the variable has been unset:

```
env | grep ^B
```

"BTARGET" should not be anywhere in the output.

Setting the BROOT Environment Variables

If you're missing "BROOT", you can set it easily enough:

```
export BROOT="/build"
```

Changing File Ownership

Now that we've installed our binaries and libraries, we need to change their ownership.

We've built and installed our binaries and libraries using the *byol* user so far. Our destination system, however, does not have any users defined. This poses a minor security risk — a user could be created with a user ID identical to that of the *byol* user on the build system, which would then have access to all our binaries.

The simplest way to restrict access is to make *root* the owner of these files, as the superuser ID is identical across all systems:

```
chown -R root:root $BROOT/tools
```

If we examine these files, we should see that both user and group are set to *root*:

```
root:~# ls -la $BROOT/tools
total 68
drwxr-xr-x 13 root root  4096 Nov 30 18:20 .
drwxr-xr-x  7 byol byol  4096 Nov 29 00:57 ..
drwxr-xr-x  2 root root 12288 Nov 30 19:31 bin
drwxr-xr-x  2 root root  4096 Nov 30 17:25 etc
drwxr-xr-x 40 root root  4096 Nov 30 19:26 include
drwxr-xr-x 11 root root 12288 Nov 30 19:30 lib
lrwxrwxrwx  1 root root     3 Nov 30 17:00 lib64 -> lib
drwxr-xr-x  6 root root  4096 Nov 30 19:16 libexec
drwxr-xr-x  5 root root  4096 Nov 30 18:20 man
drwxr-xr-x  2 root root  4096 Nov 30 19:31 sbin
drwxr-xr-x 16 root root  4096 Nov 30 19:25 share
drwxr-xr-x  3 root root  4096 Nov 30 17:25 var
drwxr-xr-x  5 root root  4096 Nov 30 17:29 x86_64-lfs-linux-gnu
drwxr-xr-x  4 root root  4096 Nov 30 17:50 x86_64-pc-linux-gnu
```

Backing Up

Now that we've got a complete (albeit temporary) system, it would be wise to backup our work. If you're using a virtual machine, exporting the VM as an OVA file is one way of ensuring you've got a patent backup.

You may choose to `tar` the `/build` directory — whatever works. The point is that you'll be able to restore the environment to the state it is in now with minimal effort at some point in the future.

This is important, particularly as a time-saver: Everything we do from this point will alter the binaries and libraries we've built and installed, so there is a possibility things will break irreparably.

DOUBLE CHECK YOUR ENVIRONMENT AS PER "Switching to Root" BEFORE CONTINUING.

Section 5

Stage 3: Native (Full) System Build

Up to this point, we've used the *byol* user to build and install software, with a few notable exceptions. *Be sure to check your environment as noted in "Switching to Root" in the previous section.*

From this point forward, everything we do will be as the *root* user. After creating our virtual and pseudo filesystems, we'll make use of the `chroot` command to set up a working environment for our *destination* system.

To protect our build system, and to enable the use of virtual and pseudo filesystems on our destination, we undertake much of the Stage 3 process in a 'chroot jail.'

A Note Regarding Package Builds

After the final GCC compile pass, it is necessary to compile software packages using fresh source files. This is to prevent any previous configuration from breaking the build. We strongly recommend you re-extract source code from the appropriate tar file before building and installing any of the userland binaries.

The packages are compiled and installed in the order given to ensure that all dependencies are met. They must, therefore, be installed in the order specified.

Required Directories and Devices

Directories

We need to create some directories before mounting filesystems:

```
mkdir -pv $BROOT/{dev,proc,sys,run}
```

The output from this command should look like this:

```
mkdir: created directory '/build/dev'
mkdir: created directory '/build/proc'
mkdir: created directory '/build/sys'
mkdir: created directory '/build/run'
```

Device Nodes

In addition to the directories required for a fully-functioning system, we need to create some device nodes. These are expected to exist by the kernel and many programs. Execute:

```
mknod -m 600 $BROOT/dev/console c 5 1
```

```
mknod -m 666 $BROOT/dev/null c 1 3
```

Mounting Device, Pseudo, and Virtual Filesystems

The kernel and many userland programs expect these filesystems to be mounted; `vmstat` is one such example.

Mounting the dev Filesystem

To populate the `dev` filesystem, we first need to mount it. We do so by using a "bind" mount, which mounts `$BROOT/dev` on to the `/dev` mount of our build system. The result is that `$BROOT/dev` and `/dev` will be mounted to the same node, and display the same information.

Normally, the `dev` filesystem would be populated by `udev` at boot. However, since we haven't installed the `udev` package, and since we haven't booted our destination system, we need to populate this filesystem manually.

Execute the following to bind `$BROOT/dev` to `/dev`:

```
mount -v --bind /dev $BROOT/dev
```

Mounting Other Pseudo and Virtual Filesystems

The `gid` parameter below ensures that the mount is owned by the group whose ID is `5`; we later assign this ID to the 'tty' group when we create our `/etc/groups` file. The 'mode' parameter sets the file mode (permissions) for the device in question.

```
mount -vt devpts devpts $BROOT/dev/pts -o gid=5,mode=620
mount -vt proc proc $BROOT/proc
mount -vt sysfs sysfs $BROOT/sys
mount -vt tmpfs tmpfs $BROOT/run
```

The Special Case of dev/shm

Some distributions link `/dev/shm` to `/run/shm`. Since we've created the "run" filesystem above, we only need to create the directory:

```
if [ -h $BROOT/dev/shm ]; then
   mkdir -pv $BROOT/$(readlink $BROOT/dev/shm)
fi
```

Entering the chroot jail

We're now ready to enter the jail and continue the final build of our distribution.

IF YOU REBOOT, YOU MUST RE-POPULATE `/dev` AND RE-MOUNT THE PSEUDO/VIRTUAL FILESYSTEMS AS NOTED IN THE PREVIOUS SECTION.

Caveats

The jail provides limited functionality — we've only installed the most basic of tools. At this stage, the goals are to simulate a running destination system and protect the build system from damage. This is why environment variables and filesystem mounts are so important.

Entering the Jail

You must be logged in as the superuser to execute the `chroot` command. Note that we specify the `$BROOT` for bash, which will adopt this location as the root directory of the jail. The "HOME", "TERM", "PATH" and prompt environment variables are passed in to set up our environment inside the jail. Other parameters are explained below:

```
chroot "$BROOT" /tools/bin/env -i HOME=/root TERM="$TERM" PS1='\u:\w\$ '
PATH=/bin:/usr/bin:/sbin:/usr/sbin:/tools/bin /tools/bin/bash --login +h
```

- **`-i`** • Instructs `env` to clear all environment variables upon entering the jail.
- **`--login`** • Notifies bash to provide an interactive login.
- **`+h`** • Disables path-having by bash. This is important; note that the `/tools/bin` directory (where we've installed many of our binaries) is last on the path. This means that the tools we install at this stage will be found before tools installed at previous stages, which is the desired behavior.

Your output from this command should be something like:

```
I have no name!:/#
```

Directory Structure

We now need to create the basic directory structure inside our jail, in keeping with the FHS standard.

Creating Directories

```
mkdir -pv /{bin,boot,etc/{opt,sysconfig},home,lib/firmware,mnt,opt}
mkdir -pv /{media/{floppy,cdrom},sbin,srv,var}
```

```
install -dv -m 0750 /root
install -dv -m 1777 /tmp /var/tmp
mkdir -pv /usr/{,local/}{bin,include,lib,sbin,src}
mkdir -pv /usr/{,local/}share/{color,dict,doc,info,locale,man}
mkdir -v  /usr/{,local/}share/{misc,terminfo,zoneinfo}
mkdir -v  /usr/libexec
mkdir -pv /usr/{,local/}share/man/man{1..8}
case $(uname -m) in
 x86_64) ln -sv lib /lib64
  ln -sv lib /usr/lib64
  ln -sv lib /usr/local/lib64 ;;
esac
mkdir -v /var/{log,mail,spool}
ln -sv /run /var/run
ln -sv /run/lock /var/lock
mkdir -pv /var/{opt,cache,lib/{color,misc,locate},local}
```

Take special note of the permissions assigned to the root home directory and the temporary directories.

Required Files

Some userland programs require the existence of specific files before they can be installed or used. We create these files (or links) here. These will be replaced as we build and install the userland.

Creating Files and Links

Execute the following:

```
ln -sv /tools/bin/{bash,cat,echo,pwd,stty} /bin
ln -sv /tools/bin/perl /usr/bin
ln -sv /tools/lib/libgcc_s.so{,.1} /usr/lib
ln -sv /tools/lib/libstdc++.so{,.6} /usr/lib
sed 's/tools/usr/' /tools/lib/libstdc++.la > /usr/lib/libstdc++.la
ln -sv bash /bin/sh
```

The purpose of each is explained below:

- **`/bin/bash`** • Quite a few scripts expect the bash binary to be located here. Creating this symlink keeps those scripts from breaking.
- **`/bin/cat`** • Glibc hard-codes this pathname into its `configure` script.
- **`/bin/echo`** • Used by Glibc's test-suite; this path is also hard-coded.
- **`/bin/pwd`** • Used by Glibc; this path is hard-coded.
- **`/bin/stty`** • This pathname is hard-coded by the `expect` package.

- **`/usr/bin/perl`** - Many scripts expect to find the PERL binary at this location; this keeps them from breaking.
- **`/usr/lib/libgcc_s.so{,.1}`** - This is needed by Glibc to enable POSIX threads.
- **`/usr/lib/libstdc++{,.6}`** - Needed for C++ support by GMP and Glibc's test suite.
- **`/usr/lib/libstdc++.la`** - This prevents GCC from referencing a previously built library of the same name in our `tools` directory.
- **`/bin/sh`** - Some scripts hard-code this binary path.

mtab

The kernel historically exposes the list of mounted filesystems via the `/etc/mtab` file. Some userland binaries expect this information available. To do this, execute:

```
ln -sv /proc/self/mounts /etc/mtab
```

/etc/passwd

The need for this file is obvious, without it, we can't set passwords or log in to our system once we boot it.

```
cat > /etc/passwd << "EOF"
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/dev/null:/bin/false
daemon:x:6:6:Daemon User:/dev/null:/bin/false
messagebus:x:18:18:D-Bus Message Daemon User:/var/run/dbus:/bin/false
nobody:x:99:99:Unprivileged User:/dev/null:/bin/false
EOF
```

/etc/group

This is another file we need to make use of file permissions properly.

```
cat > /etc/group << "EOF"
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:
usb:x:14:
cdrom:x:15:
adm:x:16:
messagebus:x:18:
systemd-journal:x:23:
input:x:24:
mail:x:34:
nogroup:x:99:
users:x:999:
EOF
```

Re-executing Our Login Shell

The following command instructs bash to execute another login. This results in our bash prompt changing, as we've now installed the `/etc/passwd` and `/etc/group` files.

```
exec /tools/bin/bash --login +h
```

Your login prompt should now reflect the user *root*.

Mandatory Log Files

We need to populate `/var/log` with files that are expected by a number of userland utilities.

```
touch /var/log/{btmp,lastlog,faillog,wtmp}
chgrp -v utmp /var/log/lastlog
chmod -v 664  /var/log/lastlog
chmod -v 600  /var/log/btmp
```

Installing the Kernel Headers (Again)

Just as during the second stage build, we need to extract the kernel headers for GCC and other programs that need them. Note that we're installing them in a different location this time.

Extracting the Kernel Header Files

Change into the kernel source directory:

```
/sources/linux-4.x
```

Change into this directory and issue the following command:

```
make mrproper
```

That done, we now call the `make` command, specifying the header file output path. We do this in three steps because `make` wipes any files from the destination directory during this step. First, we extract the files:

```
make INSTALL_HDR_PATH=dest headers_install
```

Now we remove any files that are not specifically needed, which leaves us with only the headers which are intended to be exposed to the userland:

```
find dest/include \( -name .install -o -name ..install.cmd \) -delete
```

And secondly, we copy them to the proper location:

```
cp -rv dest/include/* /usr/include
```

Installing man Pages

In the `man-pages` source directory, execute:

```
make install
```

Building Glibc, Stage 3

This is our final build of glibc!

Patching the Source

To ensure compliance with the FHS, we need to patch the glibc source code. Execute the following commands to copy the patch file to the source directory and apply the patch:

```
cp ../glibc-2.24-fhs-1.patch .
patch -Np1 -i glibc-2.24-fhs-1.patch
```

Configure

```
../configure --prefix=/usr          \
     --enable-kernel=2.6.32 \
     --enable-obsolete-rpc
```

These options are identical to those used to configure glibc in our first stage.

Compile

```
make
```

Test

Unlike previous build stages, we now have all of the dependencies needed to run the tests against our binaries before we install them. *This step is critical!* Any problems with them build toolchain, the environment, library paths, etc., will likely reveal themselves at this point.

```
make check
```

Note that you will have some of the tests fail, in particular those related to the `getaddrinfo` functions. Overall, your output from `make check` should look something like this:

```
UNSUPPORTED: elf/tst-audit10
XPASS: elf/tst-protected1a
XPASS: elf/tst-protected1b
UNSUPPORTED: math/test-double-libmvec-alias-avx2
UNSUPPORTED: math/test-double-libmvec-alias-avx2-main
UNSUPPORTED: math/test-double-libmvec-alias-avx512
UNSUPPORTED: math/test-double-libmvec-alias-avx512-main
UNSUPPORTED: math/test-double-libmvec-sincos-avx2
UNSUPPORTED: math/test-double-libmvec-sincos-avx512
UNSUPPORTED: math/test-float-libmvec-alias-avx2
UNSUPPORTED: math/test-float-libmvec-alias-avx2-main
UNSUPPORTED: math/test-float-libmvec-alias-avx512
UNSUPPORTED: math/test-float-libmvec-alias-avx512-main
UNSUPPORTED: math/test-float-libmvec-sincosf-avx2
UNSUPPORTED: math/test-float-libmvec-sincosf-avx512
FAIL: posix/tst-getaddrinfo4
FAIL: posix/tst-getaddrinfo5
Summary of test results:
  2 FAIL
   2478 PASS
 13 UNSUPPORTED
 43 XFAIL
  2 XPASS
make[1]: *** [Makefile:331: tests] Error 1
make[1]: Leaving directory '/sources/glibc-2.24'
make: *** [Makefile:9: check] Error 2
```

Create /etc/ld.so.conf

`make` will throw an error if this file does not exist when the install is run.

```
touch /etc/ld.so.conf
```

Install

```
make install
```

Install the Configuration File and Runtime for nscd

```
cp -v ../nscd/nscd.conf /etc/nscd.conf
mkdir -pv /var/cache/nscd
```

Installing Locale Files

The following locales should be defined, if only to enable compliance with future tests.

Create the Locale Directory

```
mkdir -pv /usr/lib/locale
```

Install the Locale Definitions

```
localedef -i cs_CZ -f UTF-8 cs_CZ.UTF-8
localedef -i de_DE -f ISO-8859-1 de_DE
localedef -i de_DE@euro -f ISO-8859-15 de_DE@euro
localedef -i de_DE -f UTF-8 de_DE.UTF-8
localedef -i en_GB -f UTF-8 en_GB.UTF-8
localedef -i en_HK -f ISO-8859-1 en_HK
localedef -i en_PH -f ISO-8859-1 en_PH
localedef -i en_US -f ISO-8859-1 en_US
localedef -i en_US -f UTF-8 en_US.UTF-8
localedef -i es_MX -f ISO-8859-1 es_MX
localedef -i fa_IR -f UTF-8 fa_IR
localedef -i fr_FR -f ISO-8859-1 fr_FR
localedef -i fr_FR@euro -f ISO-8859-15 fr_FR@euro
localedef -i fr_FR -f UTF-8 fr_FR.UTF-8
localedef -i it_IT -f ISO-8859-1 it_IT
localedef -i it_IT -f UTF-8 it_IT.UTF-8
localedef -i ja_JP -f EUC-JP ja_JP
localedef -i ru_RU -f KOI8-R ru_RU.KOI8-R
localedef -i ru_RU -f UTF-8 ru_RU.UTF-8
localedef -i tr_TR -f UTF-8 tr_TR.UTF-8
localedef -i zh_CN -f GB18030 zh_CN.GB18030
```

Glibc Configuration (Post-install)

We undertake the following steps to configure glibc after installing.

Add /etc/nsswitch.conf

```
cat > /etc/nsswitch.conf << "EOF"
# Begin /etc/nsswitch.conf
passwd: files
group: files
shadow: files
hosts: files dns
networks: files
protocols: files
services: files
ethers: files
rpc: files
# End /etc/nsswitch.conf
EOF
```

Add Timezone Files and Configure Our Timezone

For this step, we stay in the glibc build directory, and unzip the timezone data files to our current location.

```
tar -xf ../../tzdata2016f.tar.gz
```

Now we need to configure our timezones:

```
export ZONEINFO=/usr/share/zoneinfo
mkdir -pv $ZONEINFO/{posix,right}
for tz in etcetera southamerica northamerica europe africa antarctica  \
  asia australasia backward pacificnew systemv; do
zic -L /dev/null   -d $ZONEINFO       -y "sh yearistype.sh" ${tz}
zic -L /dev/null   -d $ZONEINFO/posix -y "sh yearistype.sh" ${tz}
zic -L leapseconds -d $ZONEINFO/right -y "sh yearistype.sh" ${tz}
done
cp -v zone.tab zone1970.tab iso3166.tab $ZONEINFO
zic -d $ZONEINFO -p Etc/GMT
zic -d $ZONEINFO -p Etc/UTC
unset ZONEINFO
```

Set the Local Timezone

```
cp -v /usr/share/zoneinfo/Etc/UTC /etc/localtime
```

Configuring the Dynamic Loader

The dynamic loader is what allows shared objects (libraries) to be called at run-time by any given binary. In this step, inform glibc what paths to search when configuring the dynamic loader:

```
cat >> /etc/ld.so.conf << "EOF"
# Add an include directory
include /etc/ld.so.conf.d/*.conf
EOF
```

And last, create the directory referenced in `/etc/ld.so.conf`:

```
mkdir -pv /etc/ld.so.conf.d
```

Adjusting the Toolchain

Now that we've got a native glibc installed, we need to reconfigure our toolchain so that it looks for libraries and utilities inside the root filesystem hierarchy instead of looking in `/tools`.

Backup the Existing Linker

Let's backup the existing linker, and replace it with the one we've most recently built:

```
mv -v /tools/bin/{ld,ld-old}
mv -v /tools/$(uname -m)-pc-linux-gnu/bin/{ld,ld-old}
mv -v /tools/bin/{ld-new,ld}
ln -sv /tools/bin/ld /tools/$(uname -m)-pc-linux-gnu/bin/ld
```

Reconfigure GCC

We now need to alter the GCC so that it uses the new dynamic linker, headers and glibc libraries:

```
gcc -dumpspecs | sed -e 's@/tools@@g'                   \
-e '/\*startfile_prefix_spec:/{n;s@.*@/usr/lib/ @}' \
-e '/\*cpp:/{n;s@$@ -isystem /usr/include@}' >      \
dirname $(gcc --print-libgcc-file-name)/specs
```

Test the GCC Configuration

It is imperative that GCC use the proper linker and libraries when compiling. We can check that using the following commands:

```
echo 'int main(){}' > dummy.c
cc dummy.c -v -Wl,--verbose &> dummy.log
readelf -l a.out | grep ': /lib'
```

Your output should be something like this:

```
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Be certain that the interpreter path *does not* include the string 'tools'. If it does, GCC has not been properly reconfigured and will break.

Additionally, we should check to ensure GCC is using the proper glibc startup files:

```
grep -o '/usr/lib.*/crt[1in].*succeeded' dummy.log
```

Your output should resemble something like this:

```
/usr/lib/../lib64/crt1.o succeeded
/usr/lib/../lib64/crti.o succeeded
/usr/lib/../lib64/crtn.o succeeded
```

Installing binutils' Dependencies

The test suite included with binutils depends on the zlib and file packages. Let's install these before moving on to binutils itself.

Installing zlib

Configure

zlib does not make use of a `configure` script.

Compile

```
make
```

Make check

```
make check
```

Install

```
make install
```

Post-configuration

The shared library needs to be moved to `/lib`, and a symlink needs to be created in `/usr/lib`. This is to ensure compliance with FHS:

```
mv -v /usr/lib/libz.so.* /lib
rm /usr/lib/libz.so
ln -sfv /lib/libz.so.1.2.8 /usr/lib/libz.so
```

Installing file

This is another package upon which the binutils test suite depends.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing binutils: Stage 3

This is our third (and final) installation of binutils.

Start with a Fresh Source

First, delete any existing binutils source directory, and re-extract the source code from the tar archive:

```
rm -rf binutils-2.27
tar xf binutils-2.27.tar
```

Check to Ensure Pseudoterminals Work in the Jail

We need this to perform the tests against the binutils build. Execute:

```
expect -c "spawn ls"
```

The output should be:

```
spawn ls
```

Configure and Build

Create the build directory as usual, and issue the following commands to configure, build, test and install binutils:

```
../configure --prefix=/usr --enable-shared --disable-werror
make tooldir=/usr
make -k check
make tooldir=/usr install
```

Installing GMP

The GMP package contains several math libraries. These are used mainly with precision arithmetic.

Unlike glibc and gcc, GMP does not require the use of a build directory.

Configure and Compile

The configure options are explained below. Execute:

```
./configure --prefix=/usr --enable-cxx --disable-static --docdir=/usr/
share/doc/gmp-6.1.1
```

- **prefix** • Tells the configure script where the compiled binaries should be installed.
- **enable-cxx** • Enables C++ support.
- **disable-static** • Disables the generation of static libraries.
- **docdir** • Specifies the prefix for installation documentation.

Compile

```
make && make html
```

Test

```
make check 2>&1 | tee gmp-check-log
```

Ensure that all 190 of the tests have passed using the following command:

```
awk '/# PASS:/{total+=$3} ; END{print total}' gmp-check-log
```

Install

```
make install && make install-html
```

Installing MPFR

Unlike glibc and gcc, MPFR does not require the use of a build directory.

Configure and Compile

The configure options are explained below. Execute:

```
./configure --prefix=/usr --disable-static  --enable-thread-safe
--docdir=/usr/share/doc/mpfr-3.1.4
```

- **`prefix`** • Tells the configure script where the compiled binaries should be installed.
- **`disable-static`** • Disables the generation of static libraries.
- **`enable-thread-safe`** • Enables thread-handling in the library.
- **`docdir`** • Specifies the prefix for installation documentation.

Compile

```
make && make html
```

Test

```
make check
```

Install

```
make install && make install-html
```

Installing MPC

The MPC package installs libraries which handle rounding and high-precision numbers.

Unlike glibc and gcc, MPC does not require the use of a build directory.

Configure and Compile

```
./configure --prefix=/usr --disable-static --docdir=/usr/share/doc/mpc-1.0.3
```

Compile

```
make && make html
```

Test

```
make check
```

Install

```
make install && make install-html
```

Installing GCC

This is the fourth (and final) build of GCC. Make sure you've deleted the existing GCC directory, and re-extracted the source code from the tar file.

Configure and Compile

Create the build directory, as per usual. Note that the `SED` variable here prevents GCC from hard-coding a path to the `sed` binary.

Note as well that we do not copy the `mpc`, `mpfr` or `gap` source packages into the GCC source directory, as we've installed these previously on the system.

Execute the following to begin the build:

```
export SED=sed
```

```
../configure --prefix=/usr --enable-languages=c,c++ --disable-multilib --disable-bootstrap --with-system-zlib
```

- **`prefix`** • Tells the configure script where the compiled binaries should be installed.
- **`enable-languages`** • We only need C and C++ right now. Others can be added later.

- **`disable-multilib`** • This functionality isn't supported on the x86_64 platform.
- **`disable-bootstrap`** • We prevent GCC from performing a bootstrap build. As we've built GCC as a cross compiler, and then built GCC natively, we are now using the native build to compile GCC a third time. A bootstrapped build replicates these steps, which should be unnecessary if all of the testing has been executed as recommended.
- **`with-system-zib`** • This instructs `make` to use the system zlib library instead of that bundled with GCC.

Compile

```
make -j2
```

Test

Before we run the tests, we need to make sure the stack size is large enough to accommodate the testing software. We increase the stack size using the `ulimit` command:

```
ulimit -s 32768
```

And now run our tests:

```
make -k check
```

Check the Test Results

You should have very few errors reported.

```
../contrib/test_summary
```

Install

```
make install
```

Add FHS-Mandated Symlink:

```
ln -sv ../usr/bin/cpp /lib
```

Add a cc Link to the GCC Binary

```
ln -sv gcc /usr/bin/cc
```

Add Compatibility Symlinks

```
install -v -dm755 /usr/lib/bfd-plugins
ln -sfv ../../libexec/gcc/$(gcc -dumpmachine)/6.2.0/liblto_plugin.so /
usr/lib/bfd-plugins/
```

Sanity Check

Once again, we stop here to check the functionality of our toolchain. Execute the following commands:

```
echo 'int main(){}' > dummy.c
cc dummy.c -v -Wl,--verbose &> dummy.log
readelf -l a.out | grep ': /lib'
```

Check GCC Startup

Now we want to check that GCC is using the proper set of files when it starts:

```
grep -o '/usr/lib.*/crt[1in].*succeeded' dummy.log
```

Result:

```
/usr/lib/gcc/i686-pc-linux-gnu/6.2.0/../../../crt1.o succeeded
/usr/lib/gcc/i686-pc-linux-gnu/6.2.0/../../../crti.o succeeded
/usr/lib/gcc/i686-pc-linux-gnu/6.2.0/../../../crtn.o succeeded
```

Check that GCC Uses the Proper Header Files

```
grep -B4 '^ /usr/include' dummy.log
```

Results:

```
##include <...> search starts here:
 /usr/lib/gcc/i686-pc-linux-gnu/6.2.0/include
 /usr/local/include
 /usr/lib/gcc/i686-pc-linux-gnu/6.2.0/include-fixed
 /usr/include
```

Verify the Linker Uses Proper Search Paths

```
grep 'SEARCH.*/usr/lib' dummy.log |sed 's|; |\n|g'
```

You should see the following results:

```
SEARCH_DIR("/usr/x86_64-unknown-linux-gnu/lib64")
SEARCH_DIR("/usr/local/lib64")
SEARCH_DIR("/lib64")
SEARCH_DIR("/usr/lib64")
SEARCH_DIR("/usr/x86_64-unknown-linux-gnu/lib")
SEARCH_DIR("/usr/local/lib")
SEARCH_DIR("/lib")
SEARCH_DIR("/usr/lib");
```

Ensure We're Using the Proper Libc Implementation

```
grep "/lib.*/libc.so.6 " dummy.log
```

Result:

```
attempt to open /lib/libc.so.6 succeeded
```

Ensure GCC Uses the Correct Dynamic Linker

```
grep found dummy.log
```

Result:

```
found ld-linux.so.2 at /lib/ld-linux.so.2
```

Cleanup

```
rm -v dummy.c a.out dummy.log
unset SED
```

Move a Misplaced File

Only execute this step if GCC has built correctly and all of the sanity checks are passed:

```
mkdir -pv /usr/share/gdb/auto-load/usr/lib
mv -v /usr/lib/*gdb.py /usr/share/gdb/auto-load/usr/lib
```

Installing Bzip2

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Patch and Amend the Source

Change into the `bzip2` source directory, and run the following command to patch the source:

```
patch -Np1 -i ../bzip2-1.0.6-install_docs-1.patch
```

Also, we need to amend the source code to ensure that symbolic links are installed using relative paths:

```
sed -i 's@\(ln -s -f \)$(PREFIX)/bin/@\1@' Makefile
```

This amendment ensures man page are installed to the proper location:

```
sed -i "s@(PREFIX)/man@(PREFIX)/share/man@g" Makefile
```

And now we need to reconstruct the `make` file. The following commands cause `make` to use a different file; the makefile we create here adds a `libb2.s` shared library, and links the `bzip2` utilities against it. Execute:

```
make -f Makefile-libbz2_so
make clean
```

Compile and Install

```
make
make PREFIX=/usr install
```

Copy the Shared Binary into the Appropriate Directory and Make Symbolic Links:

```
cp -v bzip2-shared /bin/bzip2
cp -av libbz2.so* /lib
ln -sv ../../lib/libbz2.so.1.0 /usr/lib/libbz2.so
rm -v /usr/bin/{bunzip2,bzcat,bzip2}
ln -sv bzip2 /bin/bunzip2
ln -sv bzip2 /bin/bzcat
```

Installing pkg-config

`pkg-config` does not require the use of a build directory.

Configure and Compile

The configure options are explained below. Execute:

```
./configure --prefix=/usr --with-internal-glib --disable-compile-
warnings --disable-host-tool --docdir=/usr/share/doc/pkg-config-0.29.1
```

- **`prefix`** • Tells the configure script where the compiled binaries should be installed.
- **`with-internal-glib`** • Allows `pkg-config` to use an internal glibc, as a system version is not available.
- **`disable-compile-warnings`** • Prevents the use of compilation flags which could result in build failure.
- **`disable-host-tool`** • Disables the creation of a hard-link to the `pkg-config` binary.

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing ncurses

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Amend the Source

This command prevents the installation of a static library otherwise not handled by `configure`:

```
sed -i '/LIBTOOL_INSTALL/d' c++/Makefile.in
```

Configure and Compile

The configure options are explained below. Execute:

```
./configure --prefix=/usr --mandir=/usr/share/man --with-shared
--without-debug --without-normal --enable-pc-files --enable-widec
```

- **widec** - Causes wide-character libraries to be installed. Wide character libraries are usable by both multibyte and traditional locales.
- **enable-pc-file** - Generates .pc files for `pkg-config`.
- **without-normal** - Disables the building of most static libraries.

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Move the shared libraries to the `/lib` directory:

```
mv -v /usr/lib/libncursesw.so.6* /lib
```

Because we've moved the library files, there is now a symlink to a non-existent file. So we need to re-create it:

```
ln -sfv ../../lib/$(readlink /usr/lib/libncursesw.so) /usr/lib/
libncursesw.so
```

These symlinks make the wide-character libraries accessible to binaries which expect to find the non-wide libraries:

```
for lib in ncurses form panel menu ; do
```

```
rm -vf                     /usr/lib/lib${lib}.so
echo "INPUT(-l${lib}w)" > /usr/lib/lib${lib}.so
ln -sfv ${lib}w.pc        /usr/lib/pkgconfig/${lib}.pc
done
```

This snippet ensures that binaries which look for `-lcurses` at build time are still buildable:

```
rm -vf                     /usr/lib/libcursesw.so
echo "INPUT(-lncursesw)" > /usr/lib/libcursesw.so
ln -sfv libncurses.so      /usr/lib/libcurses.so
```

Install the documentation:

```
mkdir -v       /usr/share/doc/ncurses-6.0
cp -v -R doc/* /usr/share/doc/ncurses-6.0
```

Installing attr

This package does not require the use of a build directory.

Amend the Source

Amend the location of the documentation directory so that it uses a version number:

```
sed -i -e 's|/@pkg_name@|&-@pkg_version@|' include/builddefs.in
```

Prevent `make` from overwriting man pages installed by the `man-pages` package:

```
sed -i -e "/SUBDIRS/s|man[25]||g" man/Makefile
```

Configure and Compile

Execute:

```
./configure --prefix=/usr --bindir=/bin --disable-static
```

Compile

```
make
```

Test

```
make -j1 tests root-tests
```

Install

```
make install install-dev install-lib
```

Change the permissions on the shared library:

```
chmod -v 755 /usr/lib/libattr.so
```

Move the shared library to `/lib`:

```
mv -v /usr/lib/libattr.so.* /lib
```

Which results in a missing library in `/usr/lib`, which we recreate with a symlink:

```
ln -sfv ../../lib/$(readlink /usr/lib/libattr.so) /usr/lib/libattr.so
```

Installing acl

This package does not require the use of a build directory.

Amend the Source

Amend the location of the documentation directory so that it uses a version number:

```
sed -i -e 's|/@pkg_name@|&-@pkg_version@|' include/builddefs.in
```

Fix some broken tests:

```
sed -i "s:| sed.*::g" test/{sbits-restore,cp,misc}.test
```

Fix a bug which causes `getfacl -e` to segfault on long group names:

```
sed -i -e "/TABS-1;/a if (x > (TABS-1)) x = (TABS-1);" libacl/__acl_to_
any_text.c
```

Configure and Compile

Execute:

```
./configure --prefix=/usr --bindir=/bin --disable-static --libexecdir=/
usr/lib
```

Compile

```
make
```

Install

```
make install install-dev install-lib
```

Change the permissions on the shared library:

```
chmod -v 755 /usr/lib/libacl.so
```

Move the shared library to `/lib`:

```
mv -v /usr/lib/libacl.so.* /lib
```

Which results in a missing library in `/usr/lib`, which we recreate with a symlink:

```
ln -sfv ../../lib/$(readlink /usr/lib/libacl.so) /usr/lib/libacl.so
```

Installing libpcap

This package does not require the use of a build directory.

Amend the Source

Prevent `make` from installing a static library:

```
sed -i '/install.*STALIBNAME/d' libcap/Makefile
```

Configure and Compile

The package doesn't make use of a `configure` script:

```
make
```

Install

The `RAISE_SETFCAP` variable defined on the command line prevents the use of `setcap` on the resulting library. This is necessary if the kernel or filesystem doesn't not support extended capabilities.

```
make RAISE_SETFCAP=no prefix=/usr install
chmod -v 755 /usr/lib/libcap.so
```

Move the shared library to `/lib`:

```
mv -v /usr/lib/libcap.so.* /lib
```

Which results in a missing library in `/usr/lib`, which we recreate with a symlink:

```
ln -sfv ../../lib/$(readlink /usr/lib/libcap.so) /usr/lib/libcap.so
```

Installing sed

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Configure

```
./configure --prefix=/usr --bindir=/bin --htmldir=/usr/share/doc/sed-
4.2.2
```

Compile

```
make
make html
```

Test

Note that the 'version' test may fail, resulting in a total of 5 failed tests of 66:

```
make check
```

Install

```
make install
make -C doc install-html
```

Installing shadow

This package does not require the use of a build directory.

Amend the Source

Prevent `make` from installing `groups`-related programs and related man pages. These will be installed by the `coreutils` package.

```
sed -i '/install.*STALIBNAME/d' libcap/Makefile}
find man -name Makefile.in -exec sed -i 's/groups\.1 / /'   {} \;
find man -name Makefile.in -exec sed -i 's/getspnam\.3 / /' {} \;
find man -name Makefile.in -exec sed -i 's/passwd\.5 / /'   {} \;
```

Enable the use of the SHA-512 algorithm for password encryption:

```
sed -i -e 's@#ENCRYPT_METHOD DES@ENCRYPT_METHOD SHA512@' etc/login.defs
```

Change the location of user mailboxes from `/var/spool/mail` to `/var/mail`:

```
sed -i  -e 's@/var/spool/mail@/var/mail@' etc/login.defs
```

This amendment renders the 'user add' binary consistent with LFS:

```
sed -i 's/1000/999/' etc/useradd
```

Configure

```
./configure --sysconfdir=/etc --with-group-name-max-length=32
```

Compile and Install

```
make && make install
```

Move a misplaced binary to its proper location:

```
mv -v /usr/bin/passwd /bin
```

Post-install Configuration

To enable shadowed passwords, run the following command:

```
pwconv
```

To enable shadowed groups, execute this command:

```
grpconv
```

Set the root Password

Now that we've installed the `/etc/passwd` file and the utilities to manage passwords, change the *root* password.

If you receive an error message about *root* not existing in `/etc/passwd`, there is likely a problem with the file.

Installing psmisc

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Install

```
make install
```

Move

Move the `killall` and `fuser` binaries to FHS-compliant locations:

```
mv -v /usr/bin/fuser   /bin
mv -v /usr/bin/killall /bin
```

Installing IANA-etc

This package does not require the use of a build directory.

Configure

This package does not make use of a configuration script.

Compile

```
make
```

Install

```
make install
```

Installing M4

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

M4 may fail the 'test-update-copyright' test; you check this by using `cat` to check the contents of `tests/test-suite.log`.

```
make check
```

Install

```
make install
```

Installing bison

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --docdir=/usr/share/doc/bison-3.0.4
```

Compile

```
make
```

Install

```
make install
```

Installing flex

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --docdir=/usr/share/doc/flex-2.6.1
```

Compile

```
make
```

Install

```
make install
```

Compatibility Symlink

`flex` is predated by a software package by the name of `lex`. Some userland binaries may be expecting `lex` to be installed, and may not call `flex`. However, this software package does provide for the emulation of `lex` using a symlink:

```
ln -sv flex /usr/bin/lex
```

Installing grep

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --bindir=/bin
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing readline

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Patch and Amend the Source

Change to the readline source directory, and run the following command to patch the source:

```
patch -Np1 -i ../readline-6.3-upstream_fixes-3.patch
```

Reinstalling this package results in old libraries being moved, with the suffix `.old` attached to their filename.

This can trigger bugs in `ldconfig`. We can avoid this behavior by removing the renaming:

```
sed -i '/MV.*old/d' Makefile.in
sed -i '/{OLDSUFF}/c:' support/shlib-install
```

Configure

```
./configure --prefix=/usr  --disable-static --docdir=/usr/share/doc/
readline-6.3
```

Compile and Install

```
make SHLIB_LIBS=-lncurses
make SHLIB_LIBS=-lncurses install
```

Copy the Shared Binary into the Appropriate Directory and Make Symbolic Links:

```
mv -v /usr/lib/lib{readline,history}.so.* /lib
ln -sfv ../../lib/$(readlink /usr/lib/libreadline.so) /usr/lib/
libreadline.so
ln -sfv ../../lib/$(readlink /usr/lib/libhistory.so ) /usr/lib/
libhistory.so
```

Install the Documentation

```
install -v -m644 doc/*.{ps,pdf,html,dvi} /usr/share/doc/readline-6.3
```

Installing bash

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Patch

```
patch -Np1 -i ../bash-4.3.30-upstream_fixes-3.patch
```

Configure

```
./configure --prefix=/usr --docdir=/usr/share/doc/bash-4.3.30 --without-
bash-malloc --with-installed-readline
```

Compile

```
make
```

Test

First, we need to change the permissions on the source tree so that the *nobody* user can write:

```
chown -Rv nobody
```

Now, execute the tests as the *nobody* user:

```
su nobody -s /bin/bash -c "PATH=$PATH make tests"
```

Install

```
make install
mv -vf /usr/bin/bash /bin
```

To Use the Newly Installed bash:

```
exec /bin/bash --login +h
```

Installing bc

This package does not require the use of a build directory.

Patch

```
patch -Np1 -i ../bc-1.06.95-memory_leak-1.patch
```

Configure

```
./configure --prefix=/usr --with-readline --mandir=/usr/share/man
--infodir=/usr/share/info
```

Compile

```
make
```

Test

```
echo "quit" | ./bc/bc -l Test/checklib.b
```

Install

```
make install
```

Installing lib tool

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

Libtool will fail several tests, due to unmet dependencies on the `automake` package.

```
make check
```

Install

```
make install
```

Installing GBDM

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --disable-static --enable-libgdbm-compat
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing Gperf

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --docdir=/usr/share/doc/gperf-3.0.4
```

Compile

```
make
```

Test

```
make -j1 check
```

Install

```
make install
```

Installing expat

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --disable-static
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
install -v -dm755 /usr/share/doc/expat-2.2.0
install -v -m644 doc/*.{html,png,css} /usr/share/doc/expat-2.2.0
```

Installing inetutils

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --localstatedir=/var --disable-logger
--disable-whois --disable-rcp  --disable-rexec  --disable-rlogin
--disable-rsh  --disable-servers
```

- **`disable-logger`** • This prevents the installation of the logger program; this same program is provided (albeit, a more recent version) by the `util-linux` package.

- **`disable-whois`** • Disables the building of the `whois` client.
- **`disable-rcp/rexec/rlogin/rsh`** • Disables the building of obsolete programs whose functionality has been replaced by OpenSSH.
- **`disable-servers`** • This disables the installation of a number of network services included in the package. Most of these are insecure.

Compile

```
make
```

Test

```
make check
```

Install

```
make install
mv -v /usr/bin/{hostname,ping,ping6,traceroute} /bin
mv -v /usr/bin/ifconfig /sbin
```

Installing PERL

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Pre-configuration

Install the `/etc/hosts` file, which is required by PERL's configuration files:

```
echo -e "127.0.0.1\tlocalhost\t$(hostname)" > /etc/hosts
```

Export the following variables:

```
export BUILD_ZLIB=False
export BUILD_BZIP2=0
```

Configure

PERL uses a special configuration script, as we saw previously:

```
sh Configure -des -Dprefix=/usr -Dvendorprefix=/usr -Dman1dir=/usr/
share/man/man1 -Dman3dir=/usr/share/man/man3 -Dpager="/usr/bin/less
-isR" -Duseshrplib
```

- **Dvendorprefix=/usr** • Notifies PERL where packages should install their PERL modules.
- **Dpager="/usr/bin/less -isR"** • Instructs PERL to use `less` as a pager instead of `more`.
- **Dman3dir=/usr/share/man/man3** • PERL depends on `groff`, which is not installed yet. This directive instructs `make` to build man pages anyway.
- **Duseshrplib** • Build a shared library.

Make

```
make
```

Test

```
make -k test
```

Install

```
make install
```

Cleanup

```
unset BUILD_ZLIB BUILD_BZIP2
```

Installing XML::Parser

This is a PERL module which incorporates the functionality of the expat binary into PERL.

Configuration

```
perl Makefile.PL
```

Compile

```
make
```

Test

```
make test
```

Install

```
make install
```

Installing intltool

This package does not require the use of a build directory.

Amend the Source

PERL version 5.22 and up throw a warning unless the source is amended as follows:

```
sed -i 's:\\\${:\\\$\\{:' intltool-update.in
```

Configuration

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
install -v -Dm644 doc/I18N-HOWTO /usr/share/doc/intltool-0.51.0/I18N-
HOWTO
```

Installing autoconf

This package does not require the use of a build directory.

Configuration

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing automake

This package does not require the use of a build directory.

Amend the Source Code

This fixes warnings thrown by PERL 5.22 and above.

```
sed -i 's:/\\\${:/\\\$\\{:' bin/automake.in
```

Configuration

```
./configure --prefix=/usr --docdir=/usr/share/doc/automake-1.15
```

Compile

```
make
```

Test

Some of the tests link to the wrong version of flex; we use the sed command to fix this:

```
sed -i "s:./configure:LEXLIB=/usr/lib/libfl.a &:" t/lex-{clean,depend}-
cxx.sh
make -j2 check
```

Install

```
make install
```

Installing xz

This package does not require the use of a build directory.

Amend the Source Code

This fixes warnings thrown by PERL 5.22 and above.

```
sed -e '/mf\.buffer = NULL/a next->coder->mf.size = 0;' -i src/liblzma/
lz/lz_encoder.c
```

Configuration

```
./configure --prefix=/usr --disable-static --docdir=/usr/share/doc/xz-
5.2.
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
mv -v   /usr/bin/{lzma,unlzma,lzcat,xz,unxz,xzcat} /bin
```

```
mv -v /usr/lib/liblzma.so.* /lib
ln -svf ../../lib/$(readlink /usr/lib/liblzma.so) /usr/lib/liblzma.so
```

Installing kmod

This package does not require the use of a build directory.

Configuration

```
./configure --prefix=/usr --bindir=/bin --sysconfdir=/etc --with-
rootlibdir=/lib --with-xz --with-zlib
```

Compile

```
make
```

Install

We create symlinks here, which provide backwards compatibility with the module-init tools, which previously provided loadable module support for the kernel.

```
make install
for target in depmod insmod lsmod modinfo modprobe rmmod; do
  ln -sfv ../bin/kmod /sbin/$target
done
ln -sfv kmod /bin/lsmod
```

Installing gettext

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Configuration

```
./configure --prefix=/usr --disable-static --docdir=/usr/share/doc/
gettext-0.19.8.1
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
chmod -v 0755 /usr/lib/preloadable_libintl.so
```

Installing procps-ng

This package does not require the use of a build directory.

Configuration

```
./configure --prefix=/usr --exec-prefix= --libdir=/usr/lib --docdir=/
usr/share/doc/procps-ng-3.3.12 --disable-static --disable-kill
```

Compile

```
make
```

Test

We need to modify the test code so that it does not rely on the presence of a tty device; otherwise, some tests will fail:

```
sed -i -r 's|(pmap_initname)\\\$|\1|' testsuite/pmap.test/pmap.exp
make check
```

Install

```
make install
mv -v /usr/lib/libprocps.so.* /lib
ln -sfv ../../lib/$(readlink /usr/lib/libprocps.so) /usr/lib/libprocps.
so
```

Installing e2fsprogs

This package does require the use of a `build` directory like binutils, GCC and glibc.

Amend the Source

We amend one of the test scripts here:

```
sed -i -e 's:\[\.-\]::' tests/filter.sed
```

Configure

The environment variables defined here instruct `configure` to look for libraries in the specified locations:

```
export LIBS=-L/tools/lib
export CFLAGS=-I/tools/include
export PKG_CONFIG_PATH=/tools/lib/pkgconfig
../configure --prefix=/usr --bindir=/bin --with-root-prefix="" --enable-
elf-shlibs --disable-libblkid --disable-libuuid --disable-uuidd
--disable-fsck
```

- **`bindir=/bin`** • Ensure that the compiled binaries are installed in `/lib` and `/sbin`.
- **`enable-elf-shlibs`** • Creates shared libraries used by the binaries installed as part of the package.
- **`disable`** • Prevents the installation of the specified libraries, as more recent versions are installed by the `util-linux` package.

Compile

```
make
```

Test

For the tests to run properly, we need to link some shared libraries to a place where the tests can find them:

```
ln -sfv /tools/lib/lib{blk,uu}id.so.1 lib
make LD_LIBRARY_PATH=/tools/lib check
```

Install

```
make install
```

```
make install-libs
chmod -v u+w /usr/lib/{libcom_err,libe2p,libext2fs,libss}.a
gunzip -v /usr/share/info/libext2fs.info.gz
install-info --dir-file=/usr/share/info/dir /usr/share/info/libext2fs.
info
makeinfo -o      doc/com_err.info ../lib/et/com_err.texinfo
install -v -m644 doc/com_err.info /usr/share/info
install-info --dir-file=/usr/share/info/dir /usr/share/info/com_err.info
```

Cleanup

DO NOT FORGET TO UNSET THESE ENVIRONMENT VARIABLES!

```
unset LIBS CFLAGS PKG_CONFIG_PATH
```

Installing coreutils

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Patch the Source

```
patch -Np1 -i ../coreutils-8.25-i18n-2.patch
```

Configure

The variable defined here allows `coreutils` to be built as root:

```
FORCE_UNSAFE_CONFIGURE=1 ./configure --prefix=/usr --enable-no-install-
program=kill,uptime
```

- **`enable-no-install-program`** - Prevents the installation of binaries which will be installed by other packages.

Compile

```
FORCE_UNSAFE_CONFIGURE=1 make
```

Test

For the tests to run properly, we need to link some shared libraries to a place where the tests can find them:

```
make NON_ROOT_USERNAME=nobody check-root
```

Some tests must be run as the *nobody* user, and require the user be part of more than one group:

```
echo "dummy:x:1000:nobody" >> /etc/group
```

Change the permissions so the tests can be run by the *nobody* user:

```
chown -Rv nobody .
```

And now we run the remainder of the tests:

```
su nobody -s /bin/bash  -c "PATH=$PATH make RUN_EXPENSIVE_TESTS=yes
check"
```

Install

```
make install
mv -v /usr/bin/{cat,chgrp,chmod,chown,cp,date,dd,df,echo} /bin
mv -v /usr/bin/{false,ln,ls,mkdir,mknod,mv,pwd,rm} /bin
mv -v /usr/bin/{rmdir,stty,sync,true,uname} /bin
mv -v /usr/bin/chroot /usr/sbin
mv -v /usr/share/man/man1/chroot.1 /usr/share/man/man8/chroot.8
sed -i s/\"1\"/\"8\"/1 /usr/share/man/man8/chroot.8
mv -v /usr/bin/{head,sleep,nice,test,[} /bin
```

Cleanup

Remove the *dummy* group we created for testing:

```
sed -i '/dummy/d' /etc/group
```

Installing Diffutils

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Amend the Source

We need to amend the makefiles so that the locale files are installed:

```
sed -i 's:= @mkdir_p@:= /bin/mkdir -p:' po/Makefile.in.in
```

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

The `test-update-copyright` failure is expected.

```
make test
```

Install

```
make install
```

Installing gawk

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Check

```
make check
```

Install

```
make install
```

Install Documentation

```
mkdir -v /usr/share/doc/gawk-4.1.3
cp    -v doc/{awkforai.txt,*.{eps,pdf,jpg}} /usr/share/doc/gawk-4.1.3
```

Installing findutils

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Configure

```
./configure --prefix=/usr --localstatedir=/var/lib/locate
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
mv -v /usr/bin/find /bin
sed -i 's|find:=${BINDIR}|find:=/bin|' /usr/bin/updatedb
```

Installing groff

Configuration

The `gruff` binary expects to find a "PAGE" variable in the environment which defines the default page size. This is normally stored in `/etc/papersize`; we provide a value on the command-line here.

```
export PAGE=letter
```

```
./configure --prefix=/usr
```

Build

```
make
```

Install

```
make install
```

Installing GRUB

Configure

```
./configure --prefix=/usr --sbindir=/sbin --sysconfdir=/etc --disable-
efiemu --disable-werror
```

- **`disable-efiemu`** • Disables EFI emulation and testing.

Compile

```
make
```

Install

```
make install
```

Installing less

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Configure

```
./configure --prefix=/usr --sysconfdir=/etc
```

Compile

```
make
```

Install

```
make install
```

Installing gzip

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
mv -v /usr/bin/gzip /bin
```

Installing iproute2

This package does not require the use of a build directory.

Amend the Source

We amend the source here to remove a dependency on Berkeley DB, which is not installed.

```
sed -i /ARPD/d Makefile
sed -i 's/arpd.8//' man/man8/Makefile
```

```
rm -v doc/arpd.sgml
```

This command removes a dependency on `iptables`:

```
sed -i 's/m_ipt.o//' tc/Makefile
```

Configure

There is no separate configure step.

Compile

```
make
```

Install

```
make DOCDIR=/usr/share/doc/iproute2-4.7.0 install
```

Installing kbd

This package does not require the use of a build directory.

Patch and Amend the Source

```
patch -Np1 -i ../kbd-2.0.3-backspace-1.patch
```

The following commands ensure the `resizecons` binary is not built:

```
sed -i 's/\(RESIZECONS_PROGS=\)yes/\1no/g' configure
sed -i 's/\(RESIZECONS_PROGS=\)yes/\1no/g' configure
```

Configure

```
PKG_CONFIG_PATH=/tools/lib/pkgconfig ./configure --prefix=/usr
--disable-vlock
```

- **`disable-block`** • Prevents the vlock utility from being built as it depends on the PAM library,

which we have not installed.

Compile

```
make
```

Test

```
make check
```

Install

```
make install
mkdir -v       /u [\w -.\/!:)(\&\$+{}\,]+ sr/share/doc/kbd-2.0.3
cp -R -v docs/doc/* /usr/share/doc/kbd-2.0.3
```

Installing libpipeline

This package does not require the use of a build directory.

Configure

```
PKG_CONFIG_PATH=/tools/lib/pkgconfig ./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing make

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing patch

This package does not require the use of a build directory. Be sure to start with a fresh source directory extracted from the tar file.

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Installing sysklogd

This package does not require the use of a build directory.

Amend the Source

The following commands address problems that cause `sysklogd` to segment fault, as well as remove a reference to an obsolete data structure:

```
sed -i '/Error loading kernel symbols/{n;n;d}' ksym_mod.c
sed -i 's/union wait/int/' syslogd.c
```

Configure

This package does not make use of a `configure` script.

Compile

```
make
```

Install

```
make BINDIR=/sbin install
```

Post-install Configuration

Create `/etc/syslog.conf` with the following contents:

```
auth,authpriv.* -/var/log/auth.log
*.*;auth,authpriv.none -/var/log/sys.log
daemon.* -/var/log/daemon.log
kern.* -/var/log/kern.log
mail.* -/var/log/mail.log
user.* -/var/log/user.log
*.emerg *
```

Install sysvinit

This package does not require the use of a build directory.

Patch

```
patch -Np1 -i ../sysvinit-2.88dsf-consolidated-1.patch
```

Configure

This program does not make use of a `configure` script.

Compile

```
make -C src
```

Install

```
make -C src install
```

Install eudev

This package does not require the use of a build directory.

Amend the Source

First, we need to amend one of the test scripts:

```
sed -r -i 's|/usr(/bin/test)|\1|' test/udev-test.pl
```

And now we prevent the `/tools` directory from being hard-coded as a library path into the binary; we do this by appending the following variables to the `config.cache` file:

```
cat > config.cache << "EOF"
HAVE_BLKID=1
BLKID_LIBS="-lblkid"
BLKID_CFLAGS="-I/tools/include"
EOF
```

Configure

```
./configure --prefix=/usr --bindir=/sbin --sbindir=/sbin --libdir=/
usr/lib --sysconfdir=/etc --libexecdir=/lib --with-rootprefix= --with-
rootlibdir=/lib --enable-manpages --disable-static --config-cache
```

Compile

```
LIBRARY_PATH=/tools/lib make
```

Test

First, we need to create some directories required by the testing scripts:

```
mkdir -pv /lib/udev/rules.d
mkdir -pv /etc/udev/rules.d
```

And run the tests:

```
make LD_LIBRARY_PATH=/tools/lib check
```

Install

```
make LD_LIBRARY_PATH=/tools/lib install
tar -xvf ../udev-lfs-20140408.tar.bz2
make -f udev-lfs-20140408/Makefile.lfs install
```

Post-install Configuration

```
LD_LIBRARY_PATH=/tools/lib udevadm hwdb --update
```

Installing util-linux

This package does not require the use of a build directory.

Configuration

As per FHS, we will use `/var/lib/hwclock` instead of `/etc` for our `daytime` file:

```
mkdir -pv /var/lib/hwclock
```

And now configure the package:

```
./configure ADJTIME_PATH=/var/lib/hwclock/adjtime --docdir=/usr/share/
doc/util-linux-2.28.1 --disable-chfn-chsh --disable-login --disable-
nologin --disable-su --disable-setpriv --disable-runuser --disable-
pylibmount --disable-static --without-python --without-systemd
--without-systemdsystemunitdir
```

Compile

```
make
```

Test

We do not run the tests for this package, as they will not run without specific kernel options. As well, some of these tests have been known to damage hardware.

Install

```
make install
```

Installing man-DB

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --docdir=/usr/share/doc/man-db-2.7.5
--sysconfdir=/etc --disable-setuid --with-browser=/usr/bin/lynx --with-
vgrind=/usr/bin/vgrind --with-grap=/usr/bin/grap
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
```

Post-install Configuration:

Remove a reference to a non-existent user:

```
sed -i "s:man root:root root:g" /usr/lib/tmpfiles.d/man-db.conf
```

Installing tar

This package does not require the use of a build directory.

Configure

We set the FORCE_UNSAFE_CONFIGURE variable here to allow compilation using the *root* account.

```
FORCE_UNSAFE_CONFIGURE=1 ./configure --prefix=/usr --bindir=/bin
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
make -C doc install-html docdir=/usr/share/doc/tar-1.29
```

Installing texinfo

This package does not require the use of a build directory.

Configure

```
./configure --prefix=/usr --disable-static
```

Compile

```
make
```

Test

```
make check
```

Install

```
make install
make TEXMF=/usr/share/texmf install-tex
```

Installing vim

This package does not require the use of a build directory.

Amend the Source Code

We amend the source code so that the default location of the `vimrc` file is `/etc`:

```
echo '#define SYS_VIMRC_FILE "/etc/vimrc"' >> src/feature.h
```

Configure

```
./configure --prefix=/usr
```

Compile

```
make
```

Test

```
make -j1 test
```

Install

```
make install
```

Post-install Configuration

Here we install the symlinks for the `vi` binaries and man pages, so when people execute `vi`, they actually call the 'vim' binary.

```
ln -sv vim /usr/bin/vi
for L in  /usr/share/man/{,*/}man1/vim.1; do
    ln -sv vim.1 $(dirname $L)/vi.1
done
```

Here we symlink vim so that it appears versioned (in the interest of consistency):

```
ln -sv ../vim/vim74/doc /usr/share/doc/vim-7.4
```

Now we can create the vimrc file:

```
cat > /etc/vimrc << "EOF"
" Begin /etc/vimrc
set nocompatible
set backspace=2
syntax on
if (&term == "iterm") || (&term == "putty")
  set background=dark
endif
" End /etc/vimrc
EOF
```

Stripping Binaries and Libraries

We can decrease the size of the compiled binaries and libraries by stripping out the debug symbols. This step is optional, but unless you intend to perform debugging, the symbols serve only to consume space.

This is a good time to backup your destination drive yet again — a single typo here can destroy all of the work performed thus far.

Logout/Login

It is very important to note that none of the binaries we intend to strip are running. To accomplished this, we can exit the chroot jail and re-enter it with the following two commands:

```
logout
chroot $BYOL /tools/bin/env -i HOME=/root TERM=$TERM PS1='\u:\w\$ '
```

```
PATH=/bin:/usr/bin:/sbin:/usr/sbin /tools/bin/bash --login
```

Strip

A large number of 'file format' errors will be reported by these commands. These can be ignored; the files in question are usually script files.

```
/tools/bin/find /usr/lib -type f -name \*.a \ -exec /tools/bin/strip
--strip-debug {} ';'
/tools/bin/find /lib /usr/lib -type f -name \*.so* -exec /tools/bin/
strip --strip-unneeded {} ';'
/tools/bin/find /{bin,sbin} /usr/{bin,sbin,libexec} -type f -exec /
tools/bin/strip --strip-all {} ';'
```

Section 6

Cleanup

Now we can clean some of the extra files left behind by various testing processes:

```
rm -rf /tmp/*
```

Cleanup Unused Libraries

There are also some library files used only for testing we can cleanup:

```
rm -f /usr/lib/lib{bfd,opcodes}.a
rm -f /usr/lib/libbz2.a
rm -f /usr/lib/lib{com_err,e2p,ext2fs,ss}.a
rm -f /usr/lib/libltdl.a
rm -f /usr/lib/libfl.a
rm -f /usr/lib/libfl_pic.a
rm -f /usr/lib/libz.a
```

Entering the Jail

From this point forward, the jail can be re-entered after it has been exited by issuing the following command:

```
chroot "$LFS" /usr/bin/env -i                \
    HOME=/root TERM="$TERM" PS1='\u:\w\$ ' \
    PATH=/bin:/usr/bin:/sbin:/usr/sbin     \
    /bin/bash --login
```

Pseudo and Virtual Filesystems

These must be re-created and mounted each time you re-enter the jail.

Installing the Bootscripts

These scripts are used to configure the system at boot and ensure an orderly shutdown.

LFS Bootscripts

We're going to make use of the "Linux From Scratch" bootscripts, which we downloaded along with the rest of our source code.

These install our rc scripts, as well as the scripts needed to bring the system up and down.

Find these in the `/sources` directory, un-tar them, and install them:

```
make install
```

Sysvinit Bootscripts

Once the kernel is loaded, the `init` program is usually executed. `init` needs to be given some configuration directives, which we place in `/etc/inittab`. Create this file now with the following content:

```
id:3:initdefault:
si::sysinit:/etc/rc.d/init.d/rc S
l0:0:wait:/etc/rc.d/init.d/rc 0
l1:S1:wait:/etc/rc.d/init.d/rc 1
l2:2:wait:/etc/rc.d/init.d/rc 2
l3:3:wait:/etc/rc.d/init.d/rc 3
l4:4:wait:/etc/rc.d/init.d/rc 4
l5:5:wait:/etc/rc.d/init.d/rc 5
l6:6:wait:/etc/rc.d/init.d/rc 6
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
su:S016:once:/sbin/sulogin
1:2345:respawn:/sbin/agetty --noclear tty1 9600
2:2345:respawn:/sbin/agetty tty2 9600
3:2345:respawn:/sbin/agetty tty3 9600
4:2345:respawn:/sbin/agetty tty4 9600
5:2345:respawn:/sbin/agetty tty5 9600
6:2345:respawn:/sbin/agetty tty6 9600
```

Installing Network Devices

Network devices are managed using `udev`, which provides a script that generates the initial rules for the device. These can be generated by running:

```
bash /lib/udev/init-net-rules.sh
```

The rules file which creates network devices can be examined if needed:

```
cat /etc/udev/rules.d/70-persistent-net.rules
```

Note that you must have the appropriate kernel module loaded (or compiled into the kernel) for your network device to be recognized and used.

Configuring Network Devices

The following content must be placed in `/etc/sysconfig/ifconfig.eth0` to configure the network device initially:

```
cd /etc/sysconfig/
cat > ifconfig.eth0 << "EOF"
ONBOOT=yes
IFACE=eth0
SERVICE=ipv4-static
IP=192.168.1.2
GATEWAY=192.168.1.1
PREFIX=24
BROADCAST=192.168.1.255
EOF
```

Configuring Domain Name Service (DNS) Resolution

By default, glibc looks in `/etc/resolv.conf` to determine where to send DNS queries. Create this file with the following contents:

```
domain localdomain
nameserver 8.8.8.8
nameserver 8.8.4.4
```

Configuring the Hostname

The hostname is stored in `/etc/hostname`. The hostname should not be a fully qualified domain name. Create this file with the hostname you prefer.

Configuring /etc/hosts

This file is where we provide IP address to hostname/FQDN mappings. Any number of hosts can be contained in this file, although it is usually better to rely on DNS.

We'll begin this file with the localhost mapping; create `/etc/hosts` with the following contents:

```
127.0.0.1 localhost localhost.localdomain
```

Miscellaneous Files

rc.site

The `/etc/rc.site` file provides settings that are used by the System V scripts. We're going to create this file with the following content:

```
DISTRO="BYOL" # The distro name
DISTRO_CONTACT="your-email.here" # Bug report address
DISTRO_MINI="BYOL" # Short name used in filenames for distro config
```

```
IPROMPT="yes" # Whether to display the interactive boot prompt
itime="3"    # The amount of time (in seconds) to display the prompt
wlen=$(echo "Welcome to ${DISTRO}" | wc -c )
welcome_message="Welcome to ${INFO}${DISTRO}${NORMAL}"
# Set scripts to skip the file system check on reboot
FASTBOOT=no
# Skip reading from the console
HEADLESS=no
# Write out fsck progress if yes
VERBOSE_FSCK=yes
# Speed up boot without waiting for settle in udev
OMIT_UDEV_SETTLE=n
# Speed up boot without waiting for settle in udev_retry
OMIT_UDEV_RETRY_SETTLE=no
# Skip cleaning /tmp if yes
SKIPTMPCLEAN=no
# For setclock
#UTC=1
#CLOCKPARAMS=
# For consolelog (Note that the default, 7=debug, is noisy)
#LOGLEVEL=7
# Delay between TERM and KILL signals at shutdown
KILLDELAY=3
```

/etc/profile

This file is read by various shells (notably bash) to set environment defaults. We're going to create this file with the following content:

```
if [ -x /usr/bin/id ]; then
    if [ -z "$EUID" ]; then
        # ksh workaround
        EUID=id -u
        UID=id -ru
    fi
    USER="id -un"
    LOGNAME=$USER
    MAIL="/var/spool/mail/$USER"
fi
HOSTNAME=/usr/bin/hostname 2>/dev/null
HISTSIZE=1000
PATH=/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/sbin:/bin
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE
```

This file can be amended a number of ways to meet your needs; take a look at https://tiswww.case.edu/php/chet/bash/bashref.html#Modifying-Shell-Behavior for more information.

/etc/inputrc

This file controls keyboard mappings, and the "readline" library (on which bash and other shells rely) uses this file to determine how keyboard input is handled.

We're going to create a somewhat generic file suitable for just about everyone:

```
# Allow the command prompt to wrap to the next line
set horizontal-scroll-mode Off
# Enable 8bit input
set meta-flag On
set input-meta On
# Turns off 8th bit stripping
set convert-meta Off
# Keep the 8th bit for display
set output-meta On
# none, visible or audible
set bell-style none
# All of the following map the escape sequence of the value
# contained in the 1st argument to the readline specific functions
"\eOd": backward-word
"\eOc": forward-word
# for linux console
"\e[1~": beginning-of-line
"\e[4~": end-of-line
"\e[5~": beginning-of-history
"\e[6~": end-of-history
"\e[3~": delete-char
"\e[2~": quoted-insert
# for xterm
"\eOH": beginning-of-line
"\eOF": end-of-line
# for Konsole
"\e[H": beginning-of-line
"\e[F": end-of-line
```

/etc/shells

This file contains a list of shells which can be used for the purposes of login. Each shell binary should be on its own line; we'll create this file with the following content:

```
/bin/sh
/bin/bash
```

/etc/fstab

We'll create this file with the following entries. Note that your device names may be different than those listed here; be sure you indicate the proper device name to be used during the boot process, or your system won't boot.

For example, we've built our distribution on `/dev/sdb` for most of this course. When we boot, however, it will be from a virtual machine with a single hard drive — so the device name will change to `sda`. Make sure your entries in `fstab` reflect the device names that will be present at boot.

```
/dev/sda3      /            ext4     defaults             1     1
/dev/sda4      swap         swap     pri=1                0     0
/dev/sda2      /boot        ext4     defaults             1     1
proc           /proc        proc     nosuid,noexec,nodev  0     0
sysfs          /sys         sysfs    nosuid,noexec,nodev  0     0
devpts         /dev/pts     devpts   gid=5,mode=620       0     0
tmpfs          /run         tmpfs    defaults             0     0
devtmpfs       /dev         devtmpfs mode=0755,nosuid     0     0
```

Note that we do not mount `/dev/sda1`. That partition does not need to be mounted, and in fact, is best left unmounted.

Building the Kernel

We downloaded the source for our kernel during stage 1. You may use a newer version of the kernel if it is available, but it is strongly recommended that you use source code from the stable branch.

For this course, we're using version 4.8.12 of the kernel source.

Configuring the Kernel

Kernel configuration is a subject worthy of its own course here at Linux Academy. We cannot cover all of the configuration options in this course but we do cover general configuration. Keep in mind that building the kernel is fairly straightforward once our distro is up and running, so re-building the kernel at a later time is not difficult.

Cleaning the Kernel Source Tree

Change into the kernel source directory and issue:

```
make mrproper
```

Entering the Kernel Configuration Utility

A text-based menu-driven configuration program is used to select the kernel options; we enter this system by executing:

```
make menuconfig
```

Setting Specific Kernel Options

Be sure to set the following options:

devtmpfs

```
Device Drivers ->
Generic Driver Options ->
* Maintain a devtmpfs filesystem to mount at /dev
```

DHCP Support

```
Networking Support ->
Networking Options ->
<*> The IPv6 Protocol
```

Building and Installing the Kernel

Once you've configured the kernel, save your configuration. Execute `make` to build the kernel:

```
make
```

Install the kernel modules:

```
make modules_install
```

NOTE: The /boot partition must be mounted before the following steps are undertaken. Be sure of this.

Now we need to copy the kernel image to the `/boot` directory:

```
cp -v arch/x86/boot/bzImage /boot/vmlinuz-4.8.12
```

The same goes for the kernel symbols, which are stored in `System.map`:

```
cp -v System.map /boot/System.map-4.8.12
```

It's a good idea, as well, to copy the kernel configuration file itself:

```
cp -v .config /boot/config-4.8.12
```

Install the documentation:

```
install -d /usr/share/doc/linux-4.8.12
cp -r Documentation/* /usr/share/doc/linux-4.8.12
```

Configuring Linux Modules

Now that we've got the kernel image installed, we need to tell the kernel how and in what order to load modules. For USB devices, specifically, we need to load modules in a specific order.

Create the `/etc/modprobe.d` directory:

```
install -v -m755 -d /etc/modprobe.d
```

And now, create a `usb.conf` file in that directory with the following contents:

```
install ohci_hcd /sbin/modprobe ehci_hcd ; /sbin/modprobe -i ohci_hcd ;
true
install uhci_hcd /sbin/modprobe ehci_hcd ; /sbin/modprobe -i uhci_hcd ;
true
```

Installing GRUB

For this video, we employ GRUB as a bootloader. We strongly encourage you to look at the "Bootloading with GRUB" video on the LinuxAcademy.com website, even if you're familiar with GRUB. It is very easy to misconfigure GRUB and end up with an unbootable system.

Run the following command to install the GRUB bootloader:

```
grub-install /dev/sdb
```

Creating the GRUB Configuration File

We'll create this file with the following content. Note that the value of the `set root` command must be the `/boot` partition; the `root=` entry must specify the location of the `/` partition. Again, see the "Bootloading With GRUB" video if you do not understand what these things mean.

```
set default=0
set timeout=10
insmod ext2 biosdisk
set root=(hd0,2)
menuentry "Linux 4.8.12" {
        linux   /vmlinuz-4.8.12 root=/dev/sda2 ro
}
```

Adding an Entry to an Existing GRUB Configuration

If you want to add the new distribution to an existing grub configuration, simply add the "menu entry" from

the configuration file above. You may have to run `grub-update` or a similar command to actually update the bootloader.

Logout and Reboot

We now want to reboot our system. First, let's logout:

```
logout
```

Unmount Virtual Filesystems

```
umount -v $LFS/dev/pts
umount -v $LFS/dev
umount -v $LFS/run
umount -v $LFS/proc
umount -v $LFS/sys
```

Umount Our Destination Drive

```
umount $BROOT/boot
umount $BROOT
```

Now Reboot

```
shutdown -rf now
```

The system should go down for reboot. If necessary, detach the destination drive and attach it to a new VM or re-configure your virtual machine (or hardware) as needed to boot.

Trouble

If your system doesn't boot, it's likely you have issues with GRUB. Remember to double-check your device names. Again, the "Bootloading With GRUB" video can be of great help when troubleshooting problems with the bootloader.

Final Thoughts

Updates

Updates to both the kernel and installed packages are a frequent occurrence. You will need to check monthly, if not weekly, for updates to the source. Updating the packages makes use of the same process by which the packages were built (`configure`, `make`, `make install`) most of the time.

Adding Additional Software

Feel free to add additional software as needed. How you do this is entirely up to you — building from source is one option, but keep in mind that constantly rebuilding software from source to stay abreast of security updates is a time-consuming activity.