Linux From Scratch

Version 5.1.1

Gerard Beekmans

Linux From Scratch: Version 5.1.1

by Gerard Beekmans

This book describes the process of creating a Linux system from scratch, using nothing but the sources of the required software.

Dedication

This book is dedicated to my loving and supportive wife *Beverly Beekmans*.

Table of Contents

Preface

Foreword

Having used a number of different Linux distributions, I was never fully satisfied with any of them. I didn't like the arrangement of the bootscripts. I didn't like the way certain programs were configured by default. Much more of that sort of thing bothered me. Finally I realized that if I wanted full satisfaction from my Linux system I would have to build my own system from scratch, using only the source code. I resolved not to use pre-compiled packages of any kind, nor CD-ROM or boot disk that would install some basic utilities. I would use my current Linux system to develop my own.

This wild idea seemed very difficult at the time and often seemed an impossible task. After sorting out all kinds of problems, such as dependencies and compile-time errors, a custom-built Linux system was created that was fully operational. I called this system a Linux From Scratch system, or LFS for short.

I hope you will have a great time working on your own LFS!

--
Gerard Beekmans
gerard@linuxfromscratch.org

Audience

Who would want to read this book

There are many reasons why somebody would want to read this book. The principal reason being to install a Linux system straight from the source code. A question many people raise is “Why go through all the hassle of manually building a Linux system from scratch when you can just download and install an existing one?”. That is a good question and is the impetus for this section of the book.

One important reason for LFS's existence is to help people learn how a Linux system works from the inside out. Building an LFS system helps demonstrate to you what makes Linux tick, how things work together and depend on each other. One of the best things that this learning experience provides is the ability to customize Linux to your own tastes and needs.

A key benefit of LFS is that you have more control of your system without relying on someone else's Linux implementation. With LFS, you are in the driver's seat and dictate every aspect of your system, such as the directory layout and bootscript setup. You also dictate where, why and how programs are installed.

Another benefit of LFS is the ability to create a very compact Linux system. When installing a regular distribution, you are usually forced to install several programs which you are likely never to use. They're just sitting there wasting precious disk space (or worse, CPU cycles). It isn't difficult to build an LFS system of less than 100 MB. Does that still sound like a lot? A few of us have been working on creating a very small embedded LFS system. We successfully built a system that was just enough to run the Apache web server with approximately 8MB of disk space used. Further stripping could bring that down to 5 MB or less. Try that with a regular distribution.

We could compare Linux distributions to a hamburger you buy at a fast-food restaurant -- you have no idea what you are eating. LFS, on the other hand, doesn't give you a hamburger, but the recipe to make a hamburger. This allows you to review it, to omit unwanted ingredients, and to add your own ingredients which enhance the flavor of your burger. When you are satisfied with the recipe, you go on to preparing it. You make it just the way you like it: broil it, bake it, deep-fry it, barbecue it, or eat it tar-tar (raw).

Another analogy that we can use is that of comparing LFS with a finished house. LFS will give you the skeletal plan of a house, but it's up to you to build it. You have the freedom to adjust your plans as you go.

One last advantage of a custom built Linux system is security. By compiling the entire system from source code, you are empowered to audit everything and apply all the security patches you feel are needed. You don't have to wait for somebody else to compile binary packages that fix a security hole. Unless you examine the patch and implement it yourself you have no guarantee that the new binary package was built correctly and actually fixes the problem (adequately).

There are too many good reasons to build your own LFS system for them all to be listed here. This section is only the tip of the iceberg. As you continue in your LFS experience, you will find on your own the power that information and knowledge truly bring.

Who would not want to read this book

There are probably some who, for whatever reason, would feel that they do not want to read this book. If you do not wish to build your own Linux system from scratch, then you probably don't want to read this book. Our goal is to help you build a complete and usable foundation-level system. If you only want to know what happens while your computer boots, then we recommend the “From Power Up To Bash Prompt” HOWTO. The HOWTO builds a bare system which is similar to that of this book, but it focuses strictly on creating a system capable of booting to a BASH prompt.

While you decide which to read, consider your objective. If you wish to build a Linux system while learning a bit along the way, then this book is probably your best choice. If your objective is strictly educational and you do not have any plans for your finished system, then the “From Power Up To Bash Prompt” HOWTO is probably a better choice.

The “From Power Up To Bash Prompt” HOWTO is located at http://axiom.anu.edu.au/~okeefe/p2b/ or on The Linux Documentation Project's website at http://www.tldp.org/HOWTO/From-PowerUp-To-Bash-Prompt-HOWTO.html.

Prerequisites

This book assumes that its reader has a good deal of knowledge about using and installing Linux software. Before you begin building your LFS system, you should read the following HOWTOs:

- Software-Building-HOWTO

  This is a comprehensive guide to building and installing “generic” Unix software distributions under Linux. This HOWTO is available at http://www.tldp.org/HOWTO/Software-Building-HOWTO.html.

- The Linux Users' Guide

  This guide covers the usage of assorted Linux software and is available at http://espc22.murdoch.edu.au/~stewart/guide/guide.html.

- The Essential Pre-Reading Hint

  This is an LFS Hint written specifically for new users of Linux. It is mostly a list of links to excellent sources of information on a wide range of topics. Any person attempting to install LFS, should at least have an understanding of many of the topics in this hint. It is available at http://www.linuxfromscratch.org/hints/downloads/files/essential_prereading.txt

Typography

To make things easier to follow, there are a few typographical conventions used throughout the book. Following are some examples:

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

This form of text is designed to be typed exactly as seen unless otherwise noted in the surrounding text. It is also used in the explanation sections to identify which of the commands is being referenced.

```
install-info: unknown option `--dir-file=/mnt/lfs/usr/info/dir'
```

This form of text (fixed width text) is showing screen output, probably as the result of commands issued, and is also used to show filenames, such as `/etc/ld.so.conf`.

Emphasis

This form of text is used for several purposes in the book, mainly to emphasize important points, and to give examples of what to type.

http://www.linuxfromscratch.org/

This form of text is used for hyperlinks, both within the book and to external pages such as HOWTOs, download locations and websites.

```
cat > $LFS/etc/group << "EOF"
root:x:0:
bin:x:1:
......
EOF
```

This type of section is used mainly when creating configuration files. The first command tells the system to create the file $LFS/etc/group from whatever is typed on the following lines until the sequence EOF is encountered. Therefore, this whole section is generally typed as seen.

Acknowledgments

We would like to thank the following people and organizations for their contributions to the Linux From Scratch Project.

Current Project Team Members

- Gerard Beekmans <gerard@linuxfromscratch.org> -- Linux-From-Scratch initiator, LFS Project organizer.
- Matthew Burgess <matthew@linuxfromscratch.org> -- LFS Project Co-Leader, LFS General Package maintainer, LFS Book editor.
- Craig Colton <meerkats@bellsouth.net> -- LFS, ALFS, BLFS and Hints Project logo creator.
- Nathan Coulson <nathan@linuxfromscratch.org> -- LFS-Bootscripts maintainer.
- Jeroen Coumans <jeroen@linuxfromscratch.org> -- Website developer, FAQ maintainer.
- Bruce Dubbs <bdubbs@linuxfromscratch.org> -- LFS Quality Assurance Team leader, BLFS Book editor.
- Manuel Canales Esparcia <manuel@linuxfromscratch.org> -- LFS Book Editor (XML).
- Alex Groenewoud <alex@linuxfromscratch.org> -- LFS Book editor.
- Mark Hymers <markh@linuxfromscratch.org> -- CVS maintainer, BLFS Book creator, former LFS Book editor.
- James Iwanek <iwanek@linuxfromscratch.org> -- System Administration Team member.
- Nicholas Leippe <nicholas@linuxfromscratch.org> -- Wiki maintainer.
- Anderson Lizardo <lizardo@linuxfromscratch.org> -- Website backend scripts creator and maintainer.
- Bill Maltby <bill@linuxfromscratch.org> -- LFS Project organizer.
- Alexander Patrakov <alexander@linuxfromscratch.org> -- LFS Book Editor (internationalization/localization).
- Scot Mc Pherson <scot@linuxfromscratch.org> -- LFS NNTP gateway maintainer.
- Ryan Oliver <ryan@linuxfromscratch.org> -- Testing Team leader, Toolchain maintainer, co-creator of PLFS.
- James Robertson <jwrober@linuxfromscratch.org> -- Bugzilla maintainer, Wiki developer, LFS Book editor.
- Greg Schafer <greg@linuxfromscratch.org> -- Toolchain maintainer, Former LFS Book editor, co-creator of PLFS.
- Tushar Teredesai <tushar@linuxfromscratch.org> -- BLFS Book editor, Hints and Patches Projects maintainer.
- Jeremy Utley <jeremy@linuxfromscratch.org> -- LFS Book editor, Bugzilla maintainer, LFS-Bootscripts Maintainer, LFS Server co-admin.
- Zack Winkles <winkie@linuxfromscratch.org> -- LFS Book editor (Emerging Technologies), LFS-Bootscripts co-maintainer.
- Countless other people on the various LFS and BLFS mailing lists who are making this book happen by giving their suggestions, testing the book and submitting bug reports, instructions and their experiences with installing various packages.

Translators

- Manuel Canales Esparcia <macana@lfs-es.org> -- Spanish LFS translation project.
- Johan Lenglet <johan@linuxfromscratch.org> -- French LFS translation project.
- Anderson Lizardo <lizardo@linuxfromscratch.org> -- Portuguese LFS translation project.
- Thomas Reitelbach <tr@erdfunkstelle.de> -- German LFS translation project.

Mirror Maintainers

North American Mirrors

- Scott Kveton <scott@osuosl.org> -- lfs.oregonstate.edu mirror
- Mikhail Pastukhov <miha@xuy.biz> -- lfs.130th.net mirror.
- Frank Mancuso <crash4o4@gameover.com> -- lfs.crash404.com mirror.
- William Astle <lost@l-w.net> -- ca.linuxfromscratch.org mirror.
- Jeremy Polen <jpolen@rackspace.com> -- us2.linuxfromscratch.org mirror.
- Tim Jackson <tim@idge.net> -- linuxfromscratch.idge.net mirror.
- Jeremy Utley <jeremy@linux-phreak.net> -- lfs.linux-phreak.net mirror.

South American Mirrors

- Manuel Canales Esparcia <manuel@linuxfromscratch.org> -- lfsmirror.lfs-es.org mirror.
- Andres Meggiotto <sysop@mesi.com.ar> -- lfs.mesi.com.ar mirror.
- Eduardo B. Fonseca <ebf@aedsolucoes.com.br> -- br.linuxfromscratch.org mirror.

European Mirrors

- Barna Koczka <barna@siker.hu> -- hu.linuxfromscratch.org mirror.
- UK Mirror Service -- linuxfromscratch.mirror.ac.uk mirror.
- Martin Voss <Martin.Voss@ada.de> -- lfs.linux-matrix.net mirror.
- Unknown -- mirror.vtx.ch mirror
- Guido Passet <guido@primerelay.net> -- nl.linuxfromscratch.org mirror.
- Bastiaan Jacques <baafie@planet.nl> -- lfs.pagefault.net mirror
- Roel Neefs <lfs-mirror@linuxfromscratch.rave.org> -- linuxfromscratch.rave.org mirror.
- Justin Knierim <justin@jrknierim.de> -- www.lfs-matrix.de mirror
- Stephan Brendel <stevie@stevie20.de> -- lfs.netservice-neuss.de mirror.
- Unknown -- linuxfromscratch.je-zi.de mirror
- Unknown -- linuxfromscratch.tuxcenter.net mirror
- Hagen Herrschaft <hrx@hrxnet.de> -- de.linuxfromscratch.org mirror.
- Antonin Sprinzl <Antonin.Sprinzl@tuwien.ac.at> -- at.linuxfromscratch.org mirror.
- Fredrik Danerklint <fredan-lfs@fredan.org> -- se.linuxfromscratch.org mirror.
- Parisian sysadmins <archive@doc.cs.univ-paris8.fr> -- www2.fr.linuxfromscratch.org mirror.
- Alexander Velin <velin@zadnik.org> -- bg.linuxfromscratch.org mirror.
- Dirk Webster <dirk@securewebservices.co.uk> -- lfs.securewebservices.co.uk mirror
- Thomas Skyt <thomas@sofagang.dk> -- dk.linuxfromscratch.org mirror.
- Simon Nicoll <sime@dot-sime.com> -- uk.linuxfromscratch.org mirror.

Asian Mirrors

- Pui Yong <pyng@spam.averse.net> -- sg.linuxfromscratch.org mirror.
- Stuart Harris <stuart@althalus.me.uk> -- lfs.mirror.intermedia.com.sg mirror
- Unknown -- lfs.mirror.if.itb.ac.id mirror

Australian Mirrors

- Jason Andrade <jason@dstc.edu.au> -- au.linuxfromscratch.org mirror.

Donators

- Dean Benson <dean@vipersoft.co.uk> for several monetary contributions.
- DREAMWVR.COM for their past sponsorship of donating various resources to the LFS and related sub projects.
- Hagen Herrschaft <hrx@hrxnet.de> for donating a 2.2 GHz P4 system, now running under the name of *lorien*.
- O'Reilly for donating books on SQL and PHP.
- VA Software who, on behalf of Linux.com, donated a VA Linux 420 (former StartX SP2) workstation.
- Mark Stone for donating *shadowfax*, the first linuxfromscratch.org server, a 750 MHz P3 with 512 MB RAM and two 9 GB SCSI drives. When the server moved it was renamed to *belgarath*.
- Jesse Tie-Ten-Quee <highos@linuxfromscratch.org> for donating a Yamaha CDRW 8824E CD-writer.
- Countless other people on the various LFS mailing lists who are making this book better by giving their suggestions, submitting bug reports, and throwing in their criticism.

Former Team Members and Contributors

- Timothy Bauscher <timothy@linuxfromscratch.org> -- LFS Book editor, Hints Project maintainer.
- Robert Briggs for originally donating the *linuxfromscratch.org* and *linuxfromscratch.com* domain names.
- Ian Chilton <ian@ichilton.co.uk> for maintaining the Hints project.
- Marc Heerdink <gimli@linuxfromscratch.org> -- LFS Book editor.
- Seth W. Klein <sklein@linuxfromscratch.org> -- LFS FAQ creator.
- Garrett LeSage <garrett@linuxart.com> -- Original LFS banner creator.
- Simon Perreault <nomis80@videotron.ca> -- Hints Project maintainer.
- Geert Poels <Geert.Poels@skynet.be> -- Original BLFS banner creator; based on the LFS banner by Garrett LeSage.
- Frank Skettino <bkenoah@oswd.org> for the initial design of the old website -- have a look at http://www.oswd.org/.
- Jesse Tie-Ten-Quee <highos@linuxfromscratch.org> for temporarily hosting the linuxfromscratch.org server, answering countless questions on IRC and having a great deal of patience.

Structure

This book is divided into the following parts:

Part I - Introduction

Part I explains a few important things on how to proceed with the installation, and gives meta-information about the book (version, changelog, acknowledgments, associated mailing lists, and so on).

Part II - Preparing for the build

Part II describes how to prepare for the building process: making a partition, downloading the packages, and compiling temporary tools.

Part III - Building the LFS system

Part III guides you through the building of the LFS system: compiling and installing all the packages one by one, setting up the boot scripts, and installing the kernel. The resulting basic Linux system is the foundation upon which you can build other software, to extend your system in the way you like. At the end of the book you'll find a list of all of the programs, libraries and important files that have been installed as an easy to use reference.

Part I. Introduction

Chapter 1. Introduction

How things are going to be done

You are going to build your LFS system by using a previously installed Linux distribution (such as Debian, Mandrake, Red Hat, or SuSE). This existing Linux system (the host) will be used as a starting point, because you will need programs like a compiler, linker and shell to build the new system. Normally all the required tools are available if you selected “development” as one of the options when you installed your distribution.

In Chapter 2[p.9] you will first create a new Linux native partition and file system, the place where your new LFS system will be compiled and installed. Then in Chapter 3[p.14] you download all the packages and patches needed to build an LFS system, and store them on the new file system. In Chapter 4[p.20] you set up a good environment to work in.

Chapter 5[p.26] then discusses the installation of a number of packages that will form the basic development suite (or *toolchain*) which is used to build the actual system in Chapter 6[p.66]. Some of these packages are needed to resolve circular dependencies -- for example, to compile a compiler you need a compiler.

The first thing to be done in Chapter 5[p.26] is build a first pass of the toolchain, made up of Binutils and GCC. The programs from these packages will be linked statically in order for them to be usable independently of the host system. The second thing to do is build Glibc, the C library. Glibc will be compiled by the toolchain programs just built in the first pass. The third thing to do is build a second pass of the toolchain. This time the toolchain will be dynamically linked against the newly built Glibc. The remaining Chapter 5[p.26] packages are all built using this second pass toolchain and dynamically linked against the new host-independent Glibc. When this is done, the LFS installation process will no longer depend on the host distribution, with the exception of the running kernel.

You may be thinking that “this seems like a lot of work, just to get away from my host distribution”. Well, a full technical explanation is provided at the start of Chapter 5[p.26], including some notes on the differences between statically and dynamically linked programs.

In Chapter 6[p.66] your real LFS system will be built. The chroot (change root) program is used to enter a virtual environment and start a new shell whose root directory will be set to the LFS partition. This is very similar to rebooting and instructing the kernel to mount the LFS partition as the root partition. The reason that you don't actually reboot, but instead chroot, is that creating a bootable system requires additional work which isn't necessary just yet. But the major advantage is that “chrooting” allows you to continue using the host while LFS is being built. While waiting for package compilation to complete, you can simply switch to a different VC (Virtual Console) or X desktop and continue using the computer as you normally would.

To finish the installation, the bootscripts are set up in Chapter 7[p.158], the kernel and bootloader are set up in Chapter 8[p.168], and Chapter 9[p.174] contains some pointers to help you after you finish the book. Then, finally, you're ready to reboot your computer into your new LFS system.

This is the process in a nutshell. Detailed information on the steps you will take are discussed in the chapters and package descriptions as you progress through them. If something isn't completely clear now, don't worry, everything will fall into place soon.

Please read Chapter 4[p.20] carefully as it explains a few important things you should be aware of before you begin to work through Chapter 5[p.26] and beyond.

Changelog

This is version 5.1.1 of the Linux From Scratch book, dated June 5th, 2004. If this book is more than two months old, a newer and better version is probably already available. To find out, please check one of the mirrors via http://www.linuxfromscratch.org/.

Below is a list of changes made since the previous release of the book, first a summary, then a detailed log.

- Upgraded to:
  - autoconf-2.59
  - automake-1.8.4
  - coreutils-5.2.1
  - e2fsprogs-1.35
  - expect-5.41.0
  - file-4.09
  - gcc-3.3.3
  - gettext-0.14.1
  - glibc-2.3.3-lfs-5.1
  - grub-0.94
  - kbd-1.12
  - less-382
  - lfs-bootscripts-2.0.5
  - libtool-2.5.6
  - linux-2.4.26
  - man-pages-1.66
  - modutils-2.4.27
  - ncurses-5.4
  - perl-5.8.4
  - procps-3.2.1
  - psmisc-21.4
  - sed-4.0.9
  - shadow-4.0.4.1
  - tar-1.13.94
  - tcl-8.4.6
  - texinfo-4.7
  - util-linux-2.12a
- Added:
  - iana-etc-1.00
  - inetutils-1.4.2-no_server_man_pages-1.patch
  - make_devices-1.2
  - mktemp-1.5 + mktemp-1.5-add-tempfile.patch

- Removed:
  - gcc-3.3.1-suppress-libiberty.patch
  - lfs-utils-0.5
  - MAKEDEV-1.7
  - man-1.5m2-manpath.patch
  - man-1.5m2-pager.patch
  - ncurses-5.3-etip-2.patch
  - ncurses-5.3-vsscanf.patch
  - perl-5.8.0-libc-3.patch
  - procps-3.1.11-locale-fix.patch
  - shadow-4.0.3-newgrp-fix.patch
  - zlib-1.1.4-vsnprintf.patch
- June 2, 2004 [matt]: Prologue - acknowledgments, Added Thomas Reitelbach as the German translator
- May 30, 2004 [matt]: Chapter 6 - vim, corrected the optional command for invoking the testsuite
- May 23, 2004 [matt]: Chapter 6 - kbd, removed the hardcoded path to the kernel source directory
- May 19, 2004 [matt]: Chapter 6 - mktemp, added instruction to install tempfile wrapper
- May 18, 2004 [manuel]: Chapter 3 - Updated the list of mirrors for Glibc package. Fixed several textual bugs.
- May 17th, 2004 [winkie]: Chapter 5 - Pass “AUTOCONF=no” to the Glibc build. This prevents autoconf from causing us problems.
- May 16th, 2004 [jeremy]: Chapter 9 - Added a brief paragraph to the rebooting system page to discuss packages which might be useful to add prior to rebooting to the new system
- May 15th, 2004 [matt]: Chapter 6 - Added a clearer warning that make_devices needs to be customised
- May 14th, 2004 [matt]: Chapter 3 - Added glibc's md5sum
- May 14th, 2004 [matt]: Chapters 5 & 6 - Upgraded to glibc-2.3.3-lfs-5.1
- May 11th, 2004 [jeremy]: Prologue - Updated the list of active staff in the project.
- May 9th, 2004 [winkie]: Chapter 6 - Removed unused and broken entries from `nsswitch.conf`.
- May 7th, 2004 [matt]: Merged Manuel's lfs-xsl-0.9 patches
- May 7th, 2004 [matt]: Fixed README error regarding invocation of `make`
- May 3rd, 2004: LFS 5.1-pre2 released
- May 2nd, 2004 [matt]: Quoted chroot commands in chapter 6 (bug #818).
- May 2nd, 2004 [matt]: Removed description of the now non-existent part IV from the structure section in the prologue.
- May 1st, 2004 [jeremy]: Added creation of the /media and /srv directories, as well as 2 directories under /media for floppy and cdrom, as per FHS - fixes bugzilla bug #785 and #819.
- April 14th, 2004 [jeremy]: Updated to lfs-bootscripts-2.0.3, no textual changes needed
- March 24th, 2004 [jeremy]: Chapter 7 - Updated to the new lfs-bootscripts-2.0.2, and all necessary changes to the bootscript configuration
- March 21st, 2004 [winkie]: Chapter 6 - Replaced Lfs-Utils with Iana-Etc and Mktemp.
- February 27th, 2004 [jeremy]: Upgraded to Procps-3.2.0.
- February 27th, 2004 [jeremy]: Upgraded to Lfs-utils-0.5 - fixes a possible symlink attack in iana-get.
- February 27th, 2004 [jeremy]: Chapter 6 - Altered the instructions for Findutils to be FHS-compliant.

- February 26th, 2004 [jeremy]: Removed the creation of the /usr/etc directory to conform with FHS - closes bug 775.
- February 26th, 2004 [jeremy]: Upgraded to Linux-2.4.25.
- February 23rd, 2004 [alex]: Chapters 6 + 9 - Cleaned up the Revision of chroot and Reboot sections.
- February 22nd, 2004 [alex]: Moved the stripping of the final system from chapter 9 to the end of chapter 6.
- February 22nd, 2004 [alex]: Chapter 6 - Coreutils and E2fsprogs: Clarified the prerequisites for running the tests.
- February 19th, 2004 [alex]: Chapter 5 - Stripping: Removed an unnecessary “{,share/}” from the documentation's **rm** command.
- February 14th, 2004 [jeremy]: Chapter 6 - Upgraded to Less-382.
- February 14th, 2004 [jeremy]: Chapters 5 + 6 - Upgraded to Ncurses-5.4, and removed references to the etip patch.
- February 12th, 2004 [jeremy]: Chapter 6 - Removed explicit paths from the pwconv and grpconv commands, since /usr/sbin is part of the default path.
- February 9th, 2004 [alex]: Chapter 6 - Moved the Bootscripts installation section to chapter 7.
- February 8th, 2004 [matt]: Chapter 6 - Updated to Man-pages-1.66.
- February 7th, 2004 [alex]: Chapter 1 - Moved the Conventions and Acknowledgments sections to the Preface.
- February 7th, 2004 [alex]: Chapter 6 - Creating devices: replaced the MAKEDEV script with the make_devices script. Contributed by Matthias Benkmann.
- February 5th, 2004 [alex]: Chapter 6 - Simplified the final install of the kernel headers to just copying them from the temporary tools directory.
- February 4th, 2004 [alex]: Chapters 5 + 6 - Moved the Mounting of proc and devpts to before Chrooting, dropped Util-linux from the tools, and added a little arch script for Perl.

Release of version 5.1-pre1 on February 1st, 2004.

Resources

FAQ

If during the building of your LFS system you encounter any errors, or have any questions, or think you found a typo in the book, then please first consult the FAQ (Frequently Asked Questions) at http://www.linuxfromscratch.org/faq/.

IRC

Several members of the LFS community offer assistance on our community IRC (Internet Relay Chat) network. Before you utilize this mode of support, we ask that you've at least checked the LFS FAQ (see above) and the mailing list archives (see below) for the answer to your question. You can find the IRC network at *irc.linuxfromscratch.org* or *irc.linux-phreak.net* port 6667. The support channel is named #LFS-support.

Mailing lists

The *linuxfromscratch.org* server is hosting a number of mailing lists used for the development of the LFS project. These lists include, among others, the main development and support lists.

For information on which lists are available, how to subscribe to them, their archive locations, and so on, visit http://www.linuxfromscratch.org/mail.html.

News server

All the mailing lists hosted at *linuxfromscratch.org* are also accessible via the NNTP server. All messages posted to a mailing list are copied to the corresponding newsgroup, and vice versa.

The news server can be reached at *news.linuxfromscratch.org*.

Wiki

For more information on a package, updated versions, tweaks, personal experiences, and so on, see the LFS Wiki at http://wiki.linuxfromscratch.org/. You can add information there yourself too, to help others.

References

If you need still more detailed information on the packages, you will find useful pointers on this page: http://www.109bean.org.uk/LFS-references.html.

Mirror sites

The LFS project has a number of mirrors set up world-wide to make accessing the website and downloading the required packages more convenient. Please visit the website at http://www.linuxfromscratch.org/ for a list of current mirrors.

Contact information

Please direct all your questions and comments to one of the LFS mailing lists (see above).

How to ask for help

If you run into a problem while working through this book, you should first check the FAQ at http://www.linuxfromscratch.org/faq/ -- often your question is already answered there. If it is not, you should try to find the source of the problem. The following hint might give you some ideas for your troubleshooting: http://www.linuxfromscratch.org/hints/downloads/files/errors.txt.

If all that fails, you will find that most people on IRC and the mailing lists (see the section called “Resources”[p.6]) are willing to help you. But to assist them in diagnosing and solving your problem, please include all relevant information in your request for help.

Things to mention

Apart from a brief explanation of the problem you're having, the essential things to include in your request are:

- the version of the book you are using (being 5.1.1),
- the host distribution and version you are using to create LFS,
- the package or section giving you problems,
- the exact error message or symptom you are receiving,
- whether you have deviated from the book at all.

(Note that saying that you've deviated from the book doesn't mean that we won't help you. After all, LFS is about choice. It'll just help us to see other possible causes of your problem.)

Configure problems

When something goes wrong during the stage where the configure script is run, look through the `config.log` file. This file may contain errors encountered during configure which weren't printed to the screen. Include those relevant lines if you decide to ask for help.

Compile problems

To help us find the cause of the problem, both screen output and the contents of various files are useful. The screen output from both the ./configure script and the make run can be useful. Don't blindly include the whole thing but, on the other hand, don't include too little. As an example, here is some screen output from make:

```
gcc -DALIASPATH=\"/mnt/lfs/usr/share/locale:.\"
-DLOCALEDIR=\"/mnt/lfs/usr/share/locale\" -DLIBDIR=\"/mnt/lfs/usr/lib\"
-DINCLUDEDIR=\"/mnt/lfs/usr/include\" -DHAVE_CONFIG_H -I. -I.
-g -O2 -c getopt1.c
gcc -g -O2 -static -o make ar.o arscan.o commands.o dir.o expand.o file.o
function.o getopt.o implicit.o job.o main.o misc.o read.o remake.o rule.o
signame.o variable.o vpath.o default.o remote-stub.o version.o opt1.o
-lutil job.o: In function `load_too_high':
/lfs/tmp/make-3.79.1/job.c:1565: undefined reference to `getloadavg'
collect2: ld returned 1 exit status
make[2]: *** [make] Error 1
make[2]: Leaving directory `/lfs/tmp/make-3.79.1'
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory `/lfs/tmp/make-3.79.1'
make: *** [all-recursive-am] Error 2
```

In this case, many people just include the bottom section where it says:

```
make [2]: *** [make] Error 1
```

and onwards. This isn't enough for us to diagnose the problem because it only tells us that *something* went wrong, not *what* went wrong. The whole section, as in the example above, is what should be included to be helpful, because it includes the command that was executed and the command's error message(s).

An excellent article on asking for help on the Internet in general has been written by Eric S. Raymond. It is available online at http://catb.org/~esr/faqs/smart-questions.html. Read and follow the hints in that document and you are much more likely to get a response to start with and also to get the help you actually need.

Test suite problems

Many packages provide a test suite which, depending on the importance of the package, we may encourage you to run. Sometimes packages will generate false or expected failures. If you encounter these, you can check the LFS Wiki page at http://wiki.linuxfromscratch.org/ to see whether we have already noted and investigated them. If we already know about them, then usually there is no need to be concerned.

Chapter 2. Preparing a new partition

Introduction

In this chapter the partition which will host the LFS system is prepared. We will create the partition itself, make a file system on it, and mount it.

Creating a new partition

In order to build our new Linux system, we will need some space: an empty disk partition. If you don't have a free partition, and no room on any of your hard disks to make one, then you could build LFS on the same partition as the one on which your current distribution is installed. This procedure is not recommended for your first LFS install, but if you are short on disk space, and you feel brave, take a look at the hint at http://www.linuxfromscratch.org/hints/downloads/files/lfs_next_to_existing_systems.txt.

For a minimal system you will need a partition of around 1.3 GB. This is enough to store all the source tarballs and compile all the packages. But if you intend to use the LFS system as your primary Linux system, you will probably want to install additional software, and will need more space than this, probably around 2 or 3 GB.

As we almost never have enough RAM in our box, it is a good idea to use a small disk partition as swap space -- this space is used by the kernel to store seldom-used data to make room in memory for more urgent stuff. The swap partition for your LFS system can be the same one as for your host system, so you won't have to create another if your host system already uses a swap partition.

Start a disk partitioning program such as **`cfdisk`** or **`fdisk`** with an argument naming the hard disk upon which the new partition must be created -- for example `/dev/hda` for the primary IDE disk. Create a Linux native partition and a swap partition, if needed. Please refer to the man pages of **`cfdisk`** or **`fdisk`** if you don't yet know how to use the programs.

Remember the designation of your new partition -- something like `hda5`. This book will refer to it as the LFS partition. If you (now) also have a swap partition, remember its designation too. These names will later be needed for the `/etc/fstab` file.

Creating a file system on the new partition

Now that we have a blank partition, we can create a file system on it. Most widely used in the Linux world is the second extended file system (ext2), but with the high-capacity hard disks of today the so-called journaling file systems are becoming increasingly popular. Here we will create an ext2 file system, but build instructions for other file systems can be found at http://www.linuxfromscratch.org/blfs/view/stable/postlfs/filesystems.html.

To create an ext2 file system on the LFS partition run the following:

```
mke2fs /dev/xxx
```

Replace `xxx` with the name of the LFS partition (something like `hda5`).

If you created a (new) swap partition you need to initialize it as a swap partition too (also known as formatting, like you did above with **`mke2fs`**) by running:

```
mkswap /dev/yyy
```

Replace `yyy` with the name of the swap partition.

Mounting the new partition

Now that we've created a file system, we want to be able to access the partition. For that, we need to mount it, and have to choose a mount point. In this book we assume that the file system is mounted under `/mnt/lfs`, but it doesn't matter what directory you choose.

Choose a mount point and assign it to the LFS environment variable by running:

```
export LFS=/mnt/lfs
```

Now create the mount point and mount the LFS file system by running:

```
mkdir -p $LFS
mount /dev/xxx $LFS
```

Replace `xxx` with the designation of the LFS partition.

If you have decided to use multiple partitions for LFS (say one for `/` and another for `/usr`), mount them like this:

```
mkdir -p $LFS
mount /dev/xxx $LFS
mkdir $LFS/usr
mount /dev/yyy $LFS/usr
```

Of course, replace `xxx` and `yyy` with the appropriate partition names.

You should also ensure that this new partition is not mounted with permissions that are too restrictive (such as the nosuid, nodev or noatime options). You can run the **mount** command without any parameters to see with what options the LFS partition is mounted. If you see nosuid, nodev or noatime, you will need to remount it.

Now that we've made ourselves a place to work in, we're ready to download the packages.

Part II. Preparing for the build

Chapter 3. The materials: packages and patches

Introduction

Below is a list of packages you need to download for building a basic Linux system. The listed version numbers correspond to versions of the software that are *known* to work, and this book is based upon them. Unless you are an experienced LFS builder, we highly recommend not to try out newer versions, as the build commands for one version may not work with a newer version. Also, there is often a good reason for not using the latest version due to known problems that haven't been worked around yet.

All the URLs, when possible, refer to the project's page at http://www.freshmeat.net/. The Freshmeat pages will give you easy access to the official download sites as well as project websites, mailing lists, FAQs, changelogs and more.

We can't guarantee that these download locations are always available. In case a download location has changed since this book was published, please try to google for the package. Should you remain unsuccessful with this, you can consult the book's errata page at http://www.linuxfromscratch.org/lfs/print/ or, better yet, try one of the alternative means of downloading listed on http://www.linuxfromscratch.org/lfs/packages.html.

You'll need to store all the downloaded packages and patches somewhere that is conveniently available throughout the entire build. You'll also need a working directory in which to unpack the sources and build them. A scheme that works well is to use `$LFS/sources` as the place to store the tarballs and patches, *and* as a working directory. This way everything you need will be located on the LFS partition and available during all stages of the building process.

So you may want to execute, as *root*, the following command before starting your download session:

```
mkdir $LFS/sources
```

And make this directory writable (and sticky) for your normal user -- as you won't do the downloading as *root*, we guess:

```
chmod a+wt $LFS/sources
```

All the packages

Download or otherwise obtain the following packages:

Autoconf (2.59) - 903 KB:
http://freshmeat.net/projects/autoconf/

Automake (1.8.4) - 644 KB:
http://freshmeat.net/projects/automake/

Bash (2.05b) - 1,910 KB:
http://freshmeat.net/projects/gnubash/

Binutils (2.14) - 10,666 KB:
http://freshmeat.net/projects/binutils/

Bison (1.875) - 796 KB:
http://freshmeat.net/projects/bison/

Bzip2 (1.0.2) - 650 KB:
http://freshmeat.net/projects/bzip2/

Coreutils (5.2.1) - 3,860 KB:
http://freshmeat.net/projects/coreutils/

DejaGnu (1.4.4) - 1,055 KB:
http://freshmeat.net/projects/dejagnu/

Diffutils (2.8.1) - 762 KB:
http://freshmeat.net/projects/diffutils/

E2fsprogs (1.35) - 3,003 KB:
http://freshmeat.net/projects/e2fsprogs/

Ed (0.2) - 182 KB:
http://freshmeat.net/projects/ed/

Expect (5.41.0) - 510 KB:
http://freshmeat.net/projects/expect/

File (4.09) - 356 KB: -- *(see Note 1 below)*
http://freshmeat.net/projects/file/

Findutils (4.1.20) - 760 KB:
http://freshmeat.net/projects/findutils/

Flex (2.5.4a) - 372 KB:
ftp://ftp.gnu.org/gnu/non-gnu/flex/

Gawk (3.1.3) - 1,596 KB:
http://freshmeat.net/projects/gnuawk/

GCC (2.95.3) - 9,618 KB:
http://freshmeat.net/projects/gcc/

GCC-core (3.3.3) - 11,283KB:
http://freshmeat.net/projects/gcc/

GCC-g++ (3.3.3) - 2,026 KB:
http://freshmeat.net/projects/gcc/

GCC-testsuite (3.3.3) - 1,051 KB:
http://freshmeat.net/projects/gcc/

Gettext (0.14.1) - 6,397 KB:
http://freshmeat.net/projects/gettext/

Glibc (2.3.3-lfs-5.1) - 13,101 KB: -- *(see Note 2 below)*
http://freshmeat.net/projects/glibc/

Grep (2.5.1) - 545 KB:
http://freshmeat.net/projects/grep/

Groff (1.19) - 2,360 KB:
http://freshmeat.net/projects/groff/

Grub (0.94) - 902 KB:
ftp://alpha.gnu.org/pub/gnu/grub/

Gzip (1.3.5) - 324 KB:
ftp://alpha.gnu.org/gnu/gzip/

Iana-Etc (1.00) - 161 KB:
http://freshmeat.net/projects/iana-etc/

Inetutils (1.4.2) - 1,019 KB:
http://freshmeat.net/projects/inetutils/

Kbd (1.12) - 617 KB:
http://freshmeat.net/projects/kbd/

Less (382) - 259 KB:
http://freshmeat.net/projects/less/

LFS-Bootscripts (2.0.5) - 32 KB:
http://downloads.linuxfromscratch.org/

Libtool (1.5.6) - 2,602 KB:
http://freshmeat.net/projects/libtool/

Linux (2.4.26) - 30,051 KB:
http://freshmeat.net/projects/linux/

M4 (1.4) - 310 KB:
http://freshmeat.net/projects/gnum4/

Make (3.80) - 899 KB:
http://freshmeat.net/projects/gnumake/

Make_devices (1.2) - 20 KB:
http://downloads.linuxfromscratch.org/

Man (1.5m2) - 196 KB:
http://freshmeat.net/projects/man/

Man-pages (1.66) - 1,582 KB:
http://freshmeat.net/projects/man-pages/

Mktemp (1.5) - 69 KB:
http://freshmeat.net/projects/mktemp/

Modutils (2.4.27) - 229 KB:
http://freshmeat.net/projects/modutils/

Ncurses (5.4) - 2,019 KB:
http://freshmeat.net/projects/ncurses/

Net-tools (1.60) - 194 KB:
http://freshmeat.net/projects/net-tools/

Patch (2.5.4) - 182 KB:
http://freshmeat.net/projects/patch/

Perl (5.8.4) - 9,373 KB:
http://freshmeat.net/projects/perl/

Procinfo (18) - 24 KB:
http://freshmeat.net/projects/procinfo/

Procps (3.2.1) - 260 KB:
http://freshmeat.net/projects/procps/

Psmisc (21.4) - 375 KB:
http://freshmeat.net/projects/psmisc/

Sed (4.0.9) - 751 KB:
http://freshmeat.net/projects/sed/

Shadow (4.0.4.1) - 795 KB:
http://freshmeat.net/projects/shadow/

Sysklogd (1.4.1) - 80 KB:
http://freshmeat.net/projects/sysklogd/

Sysvinit (2.85) - 91 KB:
http://freshmeat.net/projects/sysvinit/

Tar (1.13.94) - 1,025 KB:
ftp://alpha.gnu.org/gnu/tar/

Tcl (8.4.6) - 3,363 KB:
http://freshmeat.net/projects/tcltk/

Texinfo (4.7) - 1,385 KB:
http://freshmeat.net/projects/texinfo/

Util-linux (2.12a) - 1,814 KB:
http://freshmeat.net/projects/util-linux/

Vim (6.2) - 3,193 KB:
http://freshmeat.net/projects/vim/

Zlib (1.2.1) - 277 KB:
http://freshmeat.net/projects/zlib/

Total size of these packages: 134 MB

Note

1) File (4.09) may not be available by the time you read this. The site administrators of the master download location are known to occasionally remove old versions when new ones are released. An alternative download location that may have older versions available is ftp://gaosu.rave.org/pub/linux/lfs/.

Note

2) As of this writing, the Glibc maintainers have decided in their wisdom not to make available new release tarballs for download. As such, the LFS toolchain team have provided a tarball of glibc sources pulled from Glibc CVS (Concurrent Versioning System) and generated a tarball from them, including patches where necessary.

We have made this tarball available courtesy of the generous LFS mirror sites:

ftp://gaosu.rave.org/pub/linux/lfs/packages/conglomeration/glibc-2.3.3-lfs-5.1.tar.bz2
ftp://lfs.mirror.intermedia.com.sg/pub/lfs/lfs-packages/conglomeration/glibc-2.3.3-lfs-5.1.tar.bz2
http://packages.lfs-es.org/glibc/glibc-2.3.3-lfs-5.1.tar.bz2
http://mirror.averse.net/lfs-packages/glibc-2.3.3-lfs-5.1.tar.bz2
ftp://mirror.averse.net/pub/lfs-packages/glibc-2.3.3-lfs-5.1.tar.bz2
ftp://ftp.lfs-matrix.de/lfs-packages/conglomeration/glibc-2.3.3-lfs-5.1.tar.bz2
ftp://ftp.sg.linuxfromscratch.org/pub/lfs-packages/glibc-2.3.3-lfs-5.1.tar.bz2
http://ftp.sg.linuxfromscratch.org/glibc-2.3.3-lfs-5.1.tar.bz2

If you wish to verify the integrity of the tarball, its MD5 digest is `cd11fabdf5162ad68329e7b28b308278`, which can be verified using **md5sum**.

Needed patches

Besides all those packages, you'll also need several patches. These correct tiny mistakes in the packages that should be fixed by the maintainer, or just make some small modifications to bend things our way. You'll need the following:

Bash Patch - 7 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/bash-2.05b-2.patch

Bison Attribute Patch - 2 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/bison-1.875-attribute.patch

Coreutils Hostname Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/coreutils-5.2.1-hostname-1.patch

Coreutils Uname Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/coreutils-5.2.1-uname-1.patch

Ed Mkstemp Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/ed-0.2-mkstemp.patch

Expect Spawn Patch - 6 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/expect-5.41.0-spawn-1.patch

GCC No-Fixincludes Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/gcc-3.3.3-no_fixincludes-1.patch

GCC Specs Patch - 11 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/gcc-3.3.3-specs-1.patch

GCC-2 Patch - 16 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/gcc-2.95.3-2.patch

GCC-2 No-Fixincludes Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/gcc-2.95.3-no-fixinc.patch

GCC-2 Return-Type Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/gcc-2.95.3-returntype-fix.patch

Inetutils No-Server-Man-Pages Patch - 4 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/inetutils-1.4.2-no_server_man_pages-1.patch

Kbd More-Programs Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/kbd-1.12-more-programs-1.patch

Man 80-Columns Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/man-1.5m2-80cols.patch

Mktemp Tempfile Patch - 3 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/mktemp-1.5-add-tempfile.patch

Net-tools Mii-Tool-Gcc33 Patch - 2 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/net-tools-1.60-miitool-gcc33-1.patch

Perl Libc Patch - 1 KB:
http://www.linuxfromscratch.org/patches/lfs/5.1.1/perl-5.8.4-libc-1.patch

In addition to the above required patches, there exist a number of optional ones created by the LFS community. Most of these solve slight problems, or enable some functionality that's not enabled by default. Feel free to examine the patches database, located at http://www.linuxfromscratch.org/patches/, and pick any additional patches you wish to use.

Chapter 4. Final Preparations

About $LFS

Throughout this book the environment variable LFS will be used several times. It is paramount that this variable is always defined. It should be set to the mount point you chose for your LFS partition. Check that your LFS variable is set up properly with:

```
echo $LFS
```

Make sure the output shows the path to your LFS partition's mount point, which is `/mnt/lfs` if you followed our example. If the output is wrong, you can always set the variable with:

```
export LFS=/mnt/lfs
```

Having this variable set means that if you are told to run a command like **`mkdir $LFS/tools`**, you can type it literally. Your shell will replace "$LFS" with "/mnt/lfs" (or whatever you set the variable to) when it processes the command line.

Don't forget to check that “$LFS” is set whenever you leave and reenter the environment (as when doing an “su” to root or another user).

Creating the $LFS/tools directory

All programs compiled in Chapter 5[p.26] will be installed under`$LFS/tools` to keep them separate from the programs compiled in Chapter 6[p.66]. The programs compiled here are only temporary tools and won't be a part of the final LFS system and by keeping them in a separate directory, we can later easily throw them away. This also helps prevent them from ending up in your host's production directories (easy to do in Chapter 5[p.26], which could be a very bad thing.

Later on you might wish to search through the binaries of your system to see what files they make use of or link against. To make this searching easier you may want to choose a unique name for the directory in which the temporary tools are stored. Instead of the simple “tools” you could use something like “tools-for-lfs”. However, you'll need to be careful to adjust all references to “tools” throughout the book -- including those in any patches, notably the GCC Specs Patch.

Create the required directory by running the following:

```
mkdir $LFS/tools
```

The next step is to create a `/tools` symlink on your *host* system. It will point to the directory we just created on the LFS partition:

```
ln -s $LFS/tools /
```

Note

The above command is correct. The **ln** command has a few syntactic variations, so be sure to check the info page before reporting what you may think is an error.

The created symlink enables us to compile our toolchain so that it always refers to `/tools`, meaning that the compiler, assembler and linker will work both in this chapter (when we are still using some tools from the host) *and* in the next (when we are “chrooted” to the LFS partition).

Adding the user lfs

When logged in as *root*, making a single mistake can damage or even wreck your system. Therefore we recommend that you build the packages in this chapter as an unprivileged user. You could of course use your own user name, but to make it easier to set up a clean work environment we'll create a new user *lfs* and use this one during the installation process. As *root*, issue the following command to add the new user:

```
useradd -s /bin/bash -m -k /dev/null lfs
```

The meaning of the switches:

- **`-s /bin/bash`**: This makes **`bash`** the default shell for user *lfs*.
- **`-m`**: This creates a home directory for *lfs*.
- **`-k /dev/null`**: This parameter prevents possible copying of files from a skeleton directory (default is `/etc/skel`) by changing the input location to the special null device.

If you want to be able to log in as *lfs*, then give *lfs* a password:

```
passwd lfs
```

and grant *lfs* full access to `$LFS/tools` by making *lfs* the directory owner:

```
chown lfs $LFS/tools
```

If you made a separate working directory as suggested, give user *lfs* ownership of this directory too:

```
chown lfs $LFS/sources
```

Next, login as user *lfs*. This can be done via a virtual console, through a display manager, or with the following substitute user command:

```
su - lfs
```

The "**-**" instructs **su** to start a *login* shell.

Setting up the environment

We're going to set up a good working environment by creating two new startup files for the **bash** shell. While logged in as user *lfs*, issue the following command to create a new `.bash_profile`:

```
cat > ~/.bash_profile << "EOF"
exec env -i HOME=$HOME TERM=$TERM PS1='\u:\w\$ ' /bin/bash
EOF
```

Normally, when you log on as user *lfs*, the initial shell is a *login* shell which reads the `/etc/profile` of your host (probably containing some settings of environment variables) and then `.bash_profile`. The **exec env -i ... /bin/bash** command in the latter file replaces the running shell with a new one with a completely empty environment, except for the HOME, TERM and PS1 variables. This ensures that no unwanted and potentially hazardous environment variables from the host system leak into our build environment. The technique used here is a little strange, but it achieves the goal of enforcing a clean environment.

The new instance of the shell is a *non-login* shell, which doesn't read the `/etc/profile` or `.bash_profile` files, but reads the `.bashrc` file instead. Create this latter file now:

```
cat > ~/.bashrc << "EOF"
set +h
umask 022
LFS=/mnt/lfs
LC_ALL=POSIX
PATH=/tools/bin:/bin:/usr/bin
export LFS LC_ALL PATH
EOF
```

The **set +h** command turns off **bash**'s hash function. Normally hashing is a useful feature: **bash** uses a hash table to remember the full pathnames of executable files to avoid searching the PATH time and time again to find the same executable. However, we'd like the new tools to be used as soon as they are installed. By switching off the hash function, our “interactive” commands (**make**, **patch**, **sed**, **cp** and so forth) will always use the newest available version during the build process.

Setting the user file-creation mask to 022 ensures that newly created files and directories are only writable for their owner, but readable and executable for anyone.

The LFS variable should of course be set to the mount point you chose.

The LC_ALL variable controls the localization of certain programs, making their messages follow the conventions of a specified country. If your host system uses a version of Glibc older than 2.2.4, having LC_ALL set to something other than “POSIX” or “C” during this chapter may cause trouble if you exit the chroot environment and wish to return later. By setting LC_ALL to “POSIX” (or “C”, the two are equivalent) we ensure that everything will work as expected in the chroot environment.

We prepend `/tools/bin` to the standard PATH so that, as we move along through this chapter, the tools we build will get used during the rest of the building process.

Finally, to have our environment fully prepared for building the temporary tools, source the just-created profile:

```
source ~/.bash_profile
```

About SBUs

Most people would like to know beforehand approximately how long it takes to compile and install each package. But "Linux from Scratch" is built on so many different systems, it is not possible to give actual times that are anywhere near accurate: the biggest package (Glibc) won't take more than twenty minutes on the fastest systems, but will take something like three days on the slowest -- no kidding. So instead of giving actual times, we've come up with the idea of using the *Static Binutils Unit* (abbreviated to *SBU*).

It works like this: the first package you compile in this book is the statically linked Binutils in Chapter 5[p.26], and the time it takes to compile this package is what we call the “Static Binutils Unit” or “SBU”. All other compile times will be expressed relative to this time.

For example, consider a particular package whose compilation time is 4.5 SBUs. This means that if on your system it took 10 minutes to compile and install the static Binutils, then you know it will take *approximately* 45 minutes to build this package. Fortunately, most build times are much shorter than the one of Binutils.

Note that if the system compiler on your host is GCC-2 based, the SBUs listed may end up being somewhat understated. This is because the SBU is based on the very first package, compiled with the old GCC, while the rest of the system is compiled with the newer GCC-3.3.3 which is known to be approximately 30% slower.

Also note that SBUs don't work well for SMP-based machines. But if you're so lucky as to have multiple processors, chances are that your system is so fast that you won't mind.

If you wish to see actual timings for specific machines, have a look at http://www.linuxfromscratch.org/~bdubbs/.

About the test suites

Most packages provide a test suite. Running the test suite for a newly built package is generally a good idea, as it can provide a nice sanity check that everything compiled correctly. A test suite that passes its set of checks usually proves that the package is functioning as the developer intended. It does not, however, guarantee that the package is totally bug free.

Some test suites are more important than others. For example, the test suites for the core toolchain packages -- GCC, Binutils, and Glibc -- are of the utmost importance due to their central role in a properly functioning system. But be warned, the test suites for GCC and Glibc can take a very long time to complete, especially on slower hardware.

Note

Experience has shown us that there is little to be gained from running the test suites in Chapter 5[p.26]. There can be no escaping the fact that the host system always exerts some influence on the tests in that chapter, often causing weird and inexplicable failures. Not only that, the tools built in Chapter 5[p.26] are temporary and eventually discarded. For the average reader of this book we recommend *not* to run the test suites in Chapter 5[p.26]. The instructions for running those test suites are still provided for the benefit of testers and developers, but they are strictly optional for everyone else.

A common problem when running the test suites for Binutils and GCC is running out of pseudo terminals (PTYs for short). The symptom is a very high number of failing tests. This can happen for several reasons, but the most likely cause is that the host system doesn't have the *devpts* file system set up correctly. We'll discuss this in more detail later on in Chapter 5[p.26].

Sometimes package test suites will give false failures. You can consult the LFS Wiki at http://wiki.linuxfromscratch.org/ to verify that these failures are normal. This applies to all tests throughout the book.

Chapter 5. Constructing a temporary system

Introduction

In this chapter we will compile and install a minimal Linux system. This system will contain just enough tools to be able to start constructing the final LFS system in the next chapter and allow a working environment with a little more user convenience than a minimum environment.

The building of this minimal system is done in two steps: first we build a brand-new and host-independent toolchain (compiler, assembler, linker, libraries, and a few useful utilities), and then use this to build all the other essential tools.

The files compiled in this chapter will be installed under the `$LFS/tools` directory to keep them separate from the files installed in the next chapter and your host's production directories. Since the packages compiled here are merely temporary, we don't want them to pollute the soon-to-be LFS system.

Before issuing the build instructions for a package, you are expected to have already unpacked it (explained shortly) as user *lfs*, and to have performed a **`cd`** into the created directory. The build instructions assume that you are using the **bash** shell.

Several of the packages are patched before compilation, but only when the patch is needed to circumvent a problem. Often the patch is needed in both this and the next chapter, but sometimes in only one of them. Therefore, don't worry when instructions for a downloaded patch seem to be missing. Also, when applying a patch, you'll occasionally see warning messages about *offset* or *fuzz*. These warnings are nothing to worry about, as the patch was still successfully applied.

During the compilation of most packages you will see many warnings scroll by on your screen. These are normal and can safely be ignored. They are just what they say they are: warnings -- mostly about deprecated, but not invalid, use of the C or C++ syntax. It's just that C standards have changed rather often and some packages still use the older standard, which is not really a problem.

After installing each package you should delete its source and build directories, *unless* told otherwise. Deleting the sources saves space, but also prevents mis-configuration when the same package is reinstalled further on. Only for three packages you will need to keep the source and build directories around for a while, so their contents can be used by later commands. Do not miss the reminders.

Now first check that your LFS environment variable is set up properly:

```
echo $LFS
```

Make sure the output shows the path to your LFS partition's mount point, which is `/mnt/lfs` if you followed our example.

Toolchain technical notes

This section attempts to explain some of the rationale and technical details behind the overall build method. It's not essential that you understand everything here immediately. Most of it will make sense once you have performed an actual build. Feel free to refer back here at any time.

The overall goal of Chapter 5[p.26] is to provide a sane, temporary environment that we can chroot into, and from which we can produce a clean, trouble-free build of the target LFS system in Chapter 6[p.66]. Along the way, we attempt to divorce ourselves from the host system as much as possible, and in so doing build a self-contained and self-hosted toolchain. It should be noted that the build process has been designed to minimize the risks for new readers and provide maximum educational value at the same time. In other words, more advanced techniques could be used to build the system.

Important

Before continuing, you really should be aware of the name of your working platform, often also referred to as the *target triplet*. For many folks the target triplet will probably be *i686-pc-linux-gnu*. A simple way to determine your target triplet is to run the `config.guess` script that comes with the source for many packages. Unpack the Binutils sources and run the script: **`./config.guess`** and note the output.

You'll also need to be aware of the name of your platform's *dynamic linker*, often also referred to as the *dynamic loader*, not to be confused with the standard linker *ld* that is part of Binutils. The dynamic linker is provided by Glibc and has the job of finding and loading the shared libraries needed by a program, preparing the program to run and then running it. For most folks the name of the dynamic linker will be *ld-linux.so.2*. On platforms that are less prevalent, the name might be *ld.so.1* and newer 64 bit platforms might even have something completely different. You should be able to determine the name of your platform's dynamic linker by looking in the `/lib` directory on your host system. A sure-fire way is to inspect a random binary from your host system by running: **`readelf -l <name of binary> | grep interpreter`** and noting the output. The authoritative reference covering all platforms is in the `shlib-versions` file in the root of the Glibc source tree.

Some key technical points of how the Chapter 5[p.26] build method works:

- Similar in principle to cross compiling whereby tools installed into the same prefix work in cooperation and thus utilize a little GNU “magic”.
- Careful manipulation of the standard linker's library search path to ensure programs are linked only against libraries we choose.
- Careful manipulation of **gcc**'s *specs* file to tell the compiler which target dynamic linker will be used.

Binutils is installed first because the **./configure** runs of both GCC and Glibc perform various feature tests on the assembler and linker to determine which software features to enable or disable. This is more important than one might first realize. An incorrectly configured GCC or Glibc can result in a subtly broken toolchain where the impact of such breakage might not show up until near the end of the build of a whole distribution. Thankfully, a test suite failure will usually alert us before too much time is wasted.

Binutils installs its assembler and linker into two locations, `/tools/bin` and `/tools/$TARGET_TRIPLET/bin`. In reality, the tools in one location are hard linked to the other. An important facet of the linker is its library search order. Detailed information can be obtained from **ld** by passing it the *--verbose* flag. For example: **ld --verbose | grep SEARCH** will show you the current search paths and their order. You can see what files are actually linked by **ld** by compiling a dummy program and passing the *--verbose* switch to the linker. For example: **gcc dummy.c -Wl,--verbose 2>&1 | grep succeeded** will show you all the files successfully opened during the linking.

The next package installed is GCC and during its run of **./configure** you'll see, for example:

```
checking what assembler to use... /tools/i686-pc-linux-gnu/bin/as
checking what linker to use... /tools/i686-pc-linux-gnu/bin/ld
```

This is important for the reasons mentioned above. It also demonstrates that GCC's configure script does not search

the PATH directories to find which tools to use. However, during the actual operation of **gcc** itself, the same search paths are not necessarily used. You can find out which standard linker **gcc** will use by running: **gcc -print-prog-name=ld**. Detailed information can be obtained from **gcc** by passing it the *-v* flag while compiling a dummy program. For example: **gcc -v dummy.c** will show you detailed information about the preprocessor, compilation and assembly stages, including **gcc**'s include search paths and their order.

The next package installed is Glibc. The most important considerations for building Glibc are the compiler, binary tools and kernel headers. The compiler is generally no problem as Glibc will always use the **gcc** found in a PATH directory. The binary tools and kernel headers can be a little more troublesome. Therefore we take no risks and use the available configure switches to enforce the correct selections. After the run of **./configure** you can check the contents of the `config.make` file in the `glibc-build` directory for all the important details. You'll note some interesting items like the use of *CC="gcc -B/tools/bin/"* to control which binary tools are used, and also the use of the *-nostdinc* and *-isystem* flags to control the compiler's include search path. These items help to highlight an important aspect of the Glibc package: it is very self-sufficient in terms of its build machinery and generally does not rely on toolchain defaults.

After the Glibc installation, we make some adjustments to ensure that searching and linking take place only within our `/tools` prefix. We install an adjusted **ld**, which has a hard-wired search path limited to `/tools/lib`. Then we amend **gcc**'s specs file to point to our new dynamic linker in `/tools/lib`. This last step is *vital* to the whole process. As mentioned above, a hard-wired path to a dynamic linker is embedded into every ELF shared executable. You can inspect this by running: **readelf -l <name of binary> | grep interpreter**. By amending **gcc**'s specs file, we are ensuring that every program compiled from here through the end of this chapter will use our new dynamic linker in `/tools/lib`.

The need to use the new dynamic linker is also the reason why we apply the Specs patch for the second pass of GCC. Failure to do so will result in the GCC programs themselves having the name of the dynamic linker from the host system's `/lib` directory embedded into them, which would defeat our goal of getting away from the host.

During the second pass of Binutils, we are able to utilize the *--with-lib-path* configure switch to control **ld**'s library search path. From this point onwards, the core toolchain is self-contained and self-hosted. The remainder of the Chapter 5[p.26] packages all build against the new Glibc in `/tools` and all is well.

Upon entering the chroot environment in Chapter 6[p.66], the first major package we install is Glibc, due to its self-sufficient nature that we mentioned above. Once this Glibc is installed into `/usr`, we perform a quick changeover of the toolchain defaults, then proceed for real in building the rest of the target LFS system.

Notes on static linking

Most programs have to perform, beside their specific task, many rather common and sometimes trivial operations. These include allocating memory, searching directories, reading and writing files, string handling, pattern matching, arithmetic and many other tasks. Instead of obliging each program to reinvent the wheel, the GNU system provides all these basic functions in ready-made libraries. The major library on any Linux system is *Glibc*.

There are two primary ways of linking the functions from a library to a program that uses them: statically or dynamically. When a program is linked statically, the code of the used functions is included in the executable, resulting in a rather bulky program. When a program is dynamically linked, what is included is a reference to the dynamic linker, the name of the library, and the name of the function, resulting in a much smaller executable. (A third way is to use the programming interface of the dynamic linker. See the *dlopen* man page for more information.)

Dynamic linking is the default on Linux and has three major advantages over static linking. First, you need only one copy of the executable library code on your hard disk, instead of having many copies of the same code included into a whole bunch of programs -- thus saving disk space. Second, when several programs use the same library function at the same time, only one copy of the function's code is required in core -- thus saving memory space. Third, when a library function gets a bug fixed or is otherwise improved, you only need to recompile this one library, instead of having to recompile all the programs that make use of the improved function.

If dynamic linking has several advantages, why then do we statically link the first two packages in this chapter? The reasons are threefold: historical, educational, and technical. Historical, because earlier versions of LFS statically linked every program in this chapter. Educational, because knowing the difference is useful. Technical, because we gain an element of independence from the host in doing so, meaning that those programs can be used independently of the host system. However, it's worth noting that an overall successful LFS build can still be achieved when the first two packages are built dynamically.

Binutils-2.14 - Pass 1

The Binutils package contains a linker, an assembler, and other tools for handling object files.

```
Approximate build time:  1.0 SBU
Required disk space:     170 MB
```

Binutils installation depends on: Bash, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed, Texinfo.

Installation of Binutils

It is important that Binutils be the first package to get compiled, because both Glibc and GCC perform various tests on the available linker and assembler to determine which of their own features to enable.

This package is known to behave badly when you change its default optimization flags (including the *-march* and *-mcpu* options). Therefore, if you have defined any environment variables that override default optimizations, such as CFLAGS and CXXFLAGS, we recommend un-setting them when building Binutils.

The Binutils documentation recommends building Binutils outside of the source directory in a dedicated build directory:

```
mkdir ../binutils-build
cd ../binutils-build
```

Note

If you want the SBU values listed in the rest of the book to be of any use, you will have to measure the time it takes to build this package -- from the configuration up to and including the first install. To achieve this easily, you could wrap the four commands in a **time** command like this: **`time { ./configure ... && ... && ... && make install; }`**.

Now prepare Binutils for compilation:

```
../binutils-2.14/configure --prefix=/tools --disable-nls
```

The meaning of the configure options:

- **`--prefix=/tools`**: This tells the configure script to prepare to install the Binutils programs in the `/tools` directory.
- **`--disable-nls`**: This disables internationalization (a word often shortened to i18n). We don't need this for our static programs and *nls* often causes problems when linking statically.

Continue with compiling the package:

```
make configure-host
make LDFLAGS="-all-static"
```

The meaning of the make parameters:

- **`configure-host`**: This forces all the subdirectories to be configured immediately. A statically linked build will fail without it. We therefore use this option to work around the problem.
- **`LDFLAGS="-all-static"`**: This tells the linker that all the Binutils programs should be linked statically. However, strictly speaking, *"-all-static"* is passed to the **libtool** program, which then passes *"-static"* to the linker.

Compilation is complete. Normally we would now run the test suite, but at this early stage the test suite framework (Tcl, Expect and DejaGnu) is not yet in place. And there would be little point in running the tests anyhow, since the programs from this first pass will soon be replaced by those from the second.

Now install the package:

```
make install
```

Now prepare the linker for the “Adjusting” phase later on:

```
make -C ld clean
make -C ld LDFLAGS="-all-static" LIB_PATH=/tools/lib
```

The meaning of the make parameters:

- **`-C ld clean`**: This tells the make program to remove all the compiled files in the `ld` subdirectory.
- **`-C ld LDFLAGS="-all-static" LIB_PATH=/tools/lib`**: This option rebuilds everything in the `ld` subdirectory. Specifying the LIB_PATH makefile variable on the command line allows us to override the default value and have it point to our temporary tools location. The value of this variable specifies the linker's default library search path. You will see how this preparation is used later on in the chapter.

Warning

Do not yet remove the Binutils build and source directories. You will need them again in their current state a bit further on in this chapter.

The details on this package are found in the section called “Contents of Binutils”[p.85].

GCC-3.3.3 - Pass 1

The GCC package contains the GNU compiler collection, which includes the C and C++ compilers.

```
Approximate build time:  4.4 SBU
Required disk space:     411.7 MB
```

GCC installation depends on: Bash, Binutils, Coreutils, Diffutils, Findutils, Gawk, Gettext, Glibc, Grep, Make, Perl, Sed, Texinfo.

Installation of GCC

Unpack only the GCC-core tarball, as we won't be needing the C++ compiler nor the test suite here.

This package is known to behave badly when you change its default optimization flags (including the *-march* and *-mcpu* options). Therefore, if you have defined any environment variables that override default optimizations, such as CFLAGS and CXXFLAGS, we recommend un-setting them when building GCC.

The GCC documentation recommends building GCC outside of the source directory in a dedicated build directory:

```
mkdir ../gcc-build
cd ../gcc-build
```

Prepare GCC for compilation:

```
../gcc-3.3.3/configure --prefix=/tools \
    --with-local-prefix=/tools \
    --disable-nls --enable-shared \
    --enable-languages=c
```

The meaning of the configure options:

- **`--with-local-prefix=/tools`**: The purpose of this switch is to remove `/usr/local/include` from **gcc**'s include search path. This is not absolutely essential; however, we want to try to minimize the influence of the host system, so this a sensible thing to do.
- **`--enable-shared`**: This switch may seem counter-intuitive at first. But using it allows the building of `libgcc_s.so.1` and `libgcc_eh.a`, and having `libgcc_eh.a` available ensures that the configure script for Glibc (the next package we compile) produces the proper results. Note that the **gcc** binaries will still be linked statically, as this is controlled by the *-static* value of BOOT_LDFLAGS in the next step.
- **`--enable-languages=c`**: This option ensures that only the C compiler is built. The option is only needed when you have downloaded and unpacked the full GCC tarball.

Continue with compiling the package:

```
make BOOT_LDFLAGS="-static" bootstrap
```

The meaning of the make parameters:

- **`BOOT_LDFLAGS="-static"`**: This tells GCC to link its programs statically.
- **`bootstrap`**: This target doesn't just compile GCC, but compiles it several times. It uses the programs compiled in a first round to compile itself a second time, and then again a third time. It then compares these second and third compiles to make sure it can reproduce itself flawlessly, which most probably means that it was compiled correctly.

Compilation is now complete, and at this point we would normally run the test suite. But, as mentioned before, the test suite framework is not in place yet. And there would be little point in running the tests anyhow, since the programs from this first pass will soon be replaced.

Now install the package:

```
make install
```

As a finishing touch we'll create a symlink. Many programs and scripts run **cc** instead of **gcc**, a thing meant to keep programs generic and therefore usable on all kinds of Unix systems. Not everybody has the GNU C compiler installed. Simply running **cc** leaves the system administrator free to decide what C compiler to install, as long as there's a symlink pointing to it:

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

The details on this package are found in the section called “Contents of GCC”[p.87].

Linux-2.4.26 headers

```
Approximate build time:  0.1 SBU
Required disk space:     192.5 MB
```

Installation of the kernel headers

As some packages need to refer to the kernel header files, we're going to unpack the kernel archive now, set it up, and copy the required files to a place where **gcc** can later find them.

Prepare for the header installation with:

```
make mrproper
```

This ensures that the kernel tree is absolutely clean. The kernel team recommends that this command be issued prior to *each* kernel compilation. You shouldn't rely on the source tree being clean after un-tarring.

Create the `include/linux/version.h` file:

```
make include/linux/version.h
```

Create the platform-specific `include/asm` symlink:

```
make symlinks
```

Install the platform-specific header files:

```
mkdir /tools/include/asm
cp include/asm/* /tools/include/asm
cp -R include/asm-generic /tools/include
```

Finally, install the cross-platform kernel header files:

```
cp -R include/linux /tools/include
```

Glibc-2.3.3-lfs-5.1

The Glibc package contains the main C library. This library provides all the basic routines for allocating memory, searching directories, opening and closing files, reading and writing them, string handling, pattern matching, arithmetic, and so on.

```
Approximate build time:  11.8 SBU
Required disk space:     734.2 MB
```

Glibc installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Gettext, Grep, Make, Perl, Sed, Texinfo.

Installation of Glibc

This package is known to behave badly when you change its default optimization flags (including the *-march* and *-mcpu* options). Therefore, if you have defined any environment variables that override default optimizations, such as CFLAGS and CXXFLAGS, we recommend un-setting them when building Glibc.

Basically, compiling Glibc in any other way than the book suggests is putting the stability of your system at risk.

The Glibc documentation recommends building Glibc outside of the source directory in a dedicated build directory:

```
mkdir ../glibc-build
cd ../glibc-build
```

Next, prepare Glibc for compilation:

```
../glibc-2.3.3-lfs-5.1/configure --prefix=/tools \
    --disable-profile --enable-add-ons=linuxthreads \
    --with-binutils=/tools/bin --with-headers=/tools/include \
    --without-gd --without-cvs
```

The meaning of the configure options:

- **`--disable-profile`**: This builds the libraries without profiling information. Omit this option if you plan to do profiling on the temporary tools.
- **`--enable-add-ons=linuxthreads`**: This tells Glibc to use the Linuxthreads add-on as its threading library.
- **`--with-binutils=/tools/bin`** and **`--with-headers=/tools/include`**: Strictly speaking these switches are not required. But they ensure nothing can go wrong with regard to what kernel headers and Binutils programs get used during the Glibc build.
- **`--without-gd`**: This prevents the build of the **memusagestat** program, which strangely enough insists on linking against the host's libraries (libgd, libpng, libz, and so forth).
- **`--without-cvs`**: This is meant to prevent the Makefiles from attempting automatic CVS checkouts when using a CVS snapshot. But it's not actually needed these days. We use it because it suppresses an annoying but harmless warning about a missing **autoconf** program.

During this stage you might see the following warning:

```
configure: WARNING:
*** These auxiliary programs are missing or incompatible versions: msgfmt
*** some features will be disabled.
*** Check the INSTALL file for required versions.
```

The missing or incompatible **msgfmt** program is generally harmless, but it's believed it can sometimes cause problems when running the test suite.

Compile the package:

```
make AUTOCONF=no
```

Compilation is now complete. As mentioned earlier, we don't recommend running the test suites for the temporary system here in this chapter. If you still want to run the Glibc test suite anyway, the following command will do so:

```
make check
```

The Glibc test suite is highly dependent on certain functions of your host system, in particular the kernel. Additionally, here in this chapter some tests can be adversely affected by existing tools or environmental issues on the host system. Of course, these won't be a problem when we run the Glibc test suite inside the chroot environment of Chapter 6[p.66]. In general, the Glibc test suite is always expected to pass. However, as mentioned above, in certain circumstances some failures are unavoidable. Here is a list of the most common issues we are aware of:

- The *math* tests sometimes fail when running on systems where the CPU is not a relatively new genuine Intel or authentic AMD. Certain optimization settings are also known to be a factor here.
- The *gettext* test sometimes fails due to host system issues. The exact reasons are not yet clear.
- The *atime* test sometimes fails when the LFS partition is mounted with the *noatime* option, or due to other file system quirks.
- The *shm* test might fail when the host system is running the devfs file system but doesn't have the tmpfs file system mounted at `/dev/shm` due to lack of support for tmpfs in the kernel.
- When running on older and slower hardware, some tests might fail due to test timeouts being exceeded.

In summary, don't worry too much if you see Glibc test suite failures here in this chapter. The Glibc in Chapter 6[p.66] is the one we'll ultimately end up using, so that is the one we would really like to see pass the tests (but even there some failures could still occur -- the *math* tests, for example). When experiencing a failure, make a note of it, then continue by reissuing the **make check**. The test suite should pick up where it left off and continue. You can circumvent this stop-start sequence by issuing a **make -k check**. But if you do that, be sure to log the output so that you can later peruse the log file and examine the total number of failures.

Though it is a harmless message, the install stage of Glibc will at the end complain about the absence of `/tools/etc/ld.so.conf`. Prevent this confusing little warning with:

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

Now install the package:

```
make install
```

Different countries and cultures have varying conventions for how to communicate. These conventions range from very simple ones, such as the format for representing dates and times, to very complex ones, such as the language spoken. The “internationalization” of GNU programs works by means of *locales*.

Note

If you are not running the test suites here in this chapter as per our recommendation, there is little point in installing the locales now. We'll be installing the locales in the next chapter.

If you still want to install the Glibc locales anyway, the following command will do so:

```
make localedata/install-locales
```

An alternative to running the previous command is to install only those locales which you need or want. This can be achieved by using the **localedef** command. Information on this can be found in the `INSTALL` file in the Glibc source. However, there are a number of locales that are essential for the tests of future packages to pass, in particular, the *libstdc++* tests from GCC. The following instructions, instead of the install-locales target above, will install the minimum set of locales necessary for the tests to run successfully:

```
mkdir -p /tools/lib/locale
```

```
localedef -i de_DE -f ISO-8859-1 de_DE
localedef -i de_DE@euro -f ISO-8859-15 de_DE@euro
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 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 it_IT -f ISO-8859-1 it_IT
localedef -i ja_JP -f EUC-JP ja_JP
```

The details on this package are found in the section called “Contents of Glibc”[p.79].

Adjusting the toolchain

Now that the temporary C libraries have been installed, we want all the tools compiled in the rest of this chapter to be linked against these libraries. To accomplish this, we need to adjust the linker and the compiler's specs file. Some people would say that it is *"black magic juju below this line"*, but it is really very simple.

First install the adjusted linker (adjusted at the end of the first pass of Binutils) by running the following command from within the `binutils-build` directory:

```
make -C ld install
```

From this point onwards everything will link *only* against the libraries in `/tools/lib`.

Note

If you somehow missed the earlier warning to retain the Binutils source and build directories from the first pass or otherwise accidentally deleted them or just don't have access to them, don't worry, all is not lost. Just ignore the above command. The result is a small chance of the subsequent testing programs linking against libraries on the host. This is not ideal, but it's not a major problem. The situation is corrected when we install the second pass of Binutils a bit further on.

Now that the adjusted linker is installed, you have to *remove* the Binutils build and source directories.

The next thing to do is to amend our GCC specs file so that it points to the new dynamic linker. A simple sed will accomplish this:

```
SPECFILE=/tools/lib/gcc-lib/*/*/specs &&
sed -e 's@ /lib/ld-linux.so.2@ /tools/lib/ld-linux.so.2@g' \
    $SPECFILE > tempspecfile &&
mv -f tempspecfile $SPECFILE &&
unset SPECFILE
```

We recommend that you cut-and-paste the above rather than try and type it all in. Or you can edit the specs file by hand if you want to: just replace the occurrence of "/lib/ld-linux.so.2" with "/tools/lib/ld-linux.so.2". Be sure to visually inspect the specs file to verify the intended change was actually made.

Important

If you are working on a platform where the name of the dynamic linker is something other than `ld-linux.so.2`, you *must* replace `ld-linux.so.2` with the name of your platform's dynamic linker in the above commands. Refer back to the section called "Toolchain technical notes"[p.27] if necessary.

Lastly, there is a possibility that some include files from the host system have found their way into GCC's private include dir. This can happen because of GCC's "fixincludes" process which runs as part of the GCC build. We'll explain more about this further on in this chapter. For now, run the following commands to eliminate this possibility:

```
rm -f /tools/lib/gcc-lib/*/*/include/{pthread.h,bits/sigthread.h}
```

Caution

It is imperative at this point to stop and ensure that the basic functions (compiling and linking) of the new toolchain are working as expected. For this we are going to perform a simple sanity check:

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

If everything is working correctly, there should be no errors, and the output of the last command will be (allowing for platform specific differences in dynamic linker name):

```
[Requesting program interpreter: /tools/lib/ld-linux.so.2]
```

Note especially that `/tools/lib` appears as the prefix of our dynamic linker.

If you did not receive the output as shown above, or received no output at all, then something is seriously wrong. You will need to investigate and retrace your steps to find out where the problem is and correct it. There is no point in continuing until this is done. First, redo the sanity check using **gcc** instead of **cc**. If this works it means the `/tools/bin/cc` symlink is missing. Revisit the section called "GCC-3.3.3 - Pass 1"[p.31] and fix the symlink. Second, ensure your PATH is correct. You can check this by running **`echo $PATH`** and verifying that `/tools/bin` is at the head of the list. If the PATH is wrong it could mean you're not logged in as user *lfs* or something went wrong back in the section called "Setting up the environment"[p.23]. Third, something may have gone wrong with the specs file amendment above. In this case redo the specs file amendment ensuring to cut-and-paste the commands as was recommended.

Once you are satisfied that all is well, clean up the test files:

```
rm dummy.c a.out
```

Tcl-8.4.6

The Tcl package contains the Tool Command Language.

```
Approximate build time:  0.9 SBU
Required disk space:     22.7 MB
```

Tcl installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of Tcl

This package and the next two are only installed to support running the test suites for GCC and Binutils. Installing three packages just for testing purposes may seem like overkill, but it is very reassuring, if not essential, to know that our most important tools are working properly. Even if the the test suites are not run in this chapter (we recommend not running them), these packages are still required to run the test suites in the next chapter.

Prepare Tcl for compilation:

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

Build the package:

```
make
```

If you want to test the results, then issue: **`TZ=UTC make test`**. However, the Tcl test suite is known to experience failures under certain host conditions that are not fully understood. Therefore, test suite failures here are not surprising, and are not considered critical. The *TZ=UTC* parameter sets the time zone to Coordinated Universal Time (UTC) also known as Greenwich Mean Time (GMT), but only for the duration of the test suite run. This ensures the clock tests are exercised correctly. More information on the TZ environment variable will be given later on in Chapter 7[p.158].

Install the package:

```
make install
```

Warning

Do not remove the `tcl8.4.6` source directory yet, as the next package will need its internal headers.

Now make a necessary symbolic link:

```
ln -s tclsh8.4 /tools/bin/tclsh
```

Contents of Tcl

Installed programs: tclsh (link to tclsh8.4), tclsh8.4

Installed library: libtcl8.4.so

Short descriptions

tclsh8.4 is the Tcl command shell.

libtcl8.4.so is the Tcl library.

Expect-5.41.0

The Expect package contains a program for doing scripted dialogues with other interactive programs.

```
Approximate build time:  0.1 SBU
Required disk space:     3.9 MB
```

Expect installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed, Tcl.

Installation of Expect

First fix a bug that can result in bogus failures during the GCC test suite run:

```
patch -Np1 -i ../expect-5.41.0-spawn-1.patch
```

Now prepare Expect for compilation:

```
./configure --prefix=/tools --with-tcl=/tools/lib --with-x=no
```

The meaning of the configure options:

- **`--with-tcl=/tools/lib`**: This ensures that the configure script finds the Tcl installation in our temporary tools location. We don't want it to find an existing one that may possibly reside on the host system.
- **`--with-x=no`**: This tells the configure script not to search for Tk (the Tcl GUI component) or the X Window System libraries, both of which may possibly reside on the host system.

Build the package:

```
make
```

(If you insist on testing the results, then issue: **`make test`**. However, the Expect test suite is known to experience failures under certain host conditions that are not fully understood. Therefore, test suite failures here are not surprising, and are not considered critical.)

And install it:

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

The meaning of the make parameter:

- **`SCRIPTS=""`**: This prevents installation of the supplementary expect scripts which are not needed.

You can now remove the source directories of both Tcl and Expect.

Contents of Expect

Installed program: expect

Installed library: libexpect5.41.0.a

Short description

expect “talks” to other interactive programs according to a script.

DejaGnu-1.4.4

The DejaGnu package contains a framework for testing other programs.

```
Approximate build time:  0.1 SBU
Required disk space:     6.1 MB
```

For its installation Dejagnu depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of DejaGnu

Prepare DejaGnu for compilation:

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

Build and install the package:

```
make install
```

Contents of DejaGnu

Installed program: runtest

Short description

runtest is the wrapper script that finds the proper expect shell and then runs DejaGnu.

GCC-3.3.3 - Pass 2

```
Approximate build time:  11.0 SBU
Required disk space:     332.7 MB
```

Re-installation of GCC

The tools required to test GCC and Binutils are installed now: Tcl, Expect and DejaGnu. Therefore we can now rebuild GCC and Binutils, linking them against the new Glibc, and test them properly (if running the test suites in this chapter). One thing to note, however, is that these test suites are highly dependent on properly functioning pseudo terminals (PTYs) which are provided by your host. These days, PTYs are most commonly implemented via the *devpts* file system. You can quickly check if your host system is set up correctly in this regard by performing a simple test:

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

The response might be:

```
The system has no more ptys.  Ask your system administrator to create more.
```

If you receive the above message, your host doesn't have its PTYs set up properly. In this case there is no point in running the test suites for GCC and Binutils until you are able to resolve the issue. You can consult the LFS Wiki at http://wiki.linuxfromscratch.org/ for more information on how to get PTYs working.

This time we will build both the C and the C++ compilers, so you'll have to unpack both the core and the g++ tarballs (and testsuite too, if you want to run the tests). Unpacking them in your working directory, they will all unfold into a single `gcc-3.3.3/` subdirectory.

First correct a problem and make an essential adjustment:

```
patch -Np1 -i ../gcc-3.3.3-no_fixincludes-1.patch
patch -Np1 -i ../gcc-3.3.3-specs-1.patch
```

The first patch disables the GCC “fixincludes” script. We mentioned this briefly earlier, but a slightly more in-depth explanation of the fixincludes process is warranted here. Under normal circumstances, the GCC fixincludes script scans your system for header files that need to be fixed. It might find that some Glibc header files on your host system need to be fixed, fix them and put them in the GCC private include directory. Then, later on in Chapter 6[p.66], after we've installed the newer Glibc, this private include directory would be searched before the system include directory, resulting in GCC finding the fixed headers from the host system, which would most likely not match the Glibc version actually used for the LFS system.

The second patch changes GCC's default location of the dynamic linker (typically `ld-linux.so.2`). It also removes `/usr/include` from GCC's include search path. Patching now rather than adjusting the specs file after installation ensures that our new dynamic linker gets used during the actual build of GCC. That is, all the final (and temporary) binaries created during the build will link against the new Glibc.

Important

The above patches are *critical* in ensuring a successful overall build. Do not forget to apply them.

Create a separate build directory again:

```
mkdir ../gcc-build
cd ../gcc-build
```

Before starting to build GCC, remember to unset any environment variables that override the default optimization flags.

Now prepare GCC for compilation:

```
../gcc-3.3.3/configure --prefix=/tools \
    --with-local-prefix=/tools \
```

```
    --enable-clocale=gnu --enable-shared \
    --enable-threads=posix --enable-__cxa_atexit \
    --enable-languages=c,c++
```

The meaning of the new configure options:

- **`--enable-clocale=gnu`**: This option ensures the correct locale model is selected for the C++ libraries under all circumstances. If the configure script finds the *de_DE* locale installed, it will select the correct *gnu* locale model. However, people who don't install the *de_DE* locale would run the risk of building ABI incompatible C++ libraries due to the wrong *generic* locale model being selected.
- **`--enable-threads=posix`**: This enables C++ exception handling for multi-threaded code.
- **`--enable-__cxa_atexit`**: This option allows use of __cxa_atexit, rather than atexit, to register C++ destructors for local statics and global objects and is essential for fully standards-compliant handling of destructors. It also affects the C++ ABI and therefore results in C++ shared libraries and C++ programs that are interoperable with other Linux distributions.
- **`--enable-languages=c,c++`**: This option ensures that both the C and C++ compilers are built.

Compile the package:

```
make
```

There is no need to use the *bootstrap* target now, as the compiler we're using to compile this GCC was built from the exact same version of the GCC sources we used earlier.

Compilation is now complete. As mentioned earlier, we don't recommend running the test suites for the temporary tools here in this chapter. If you still want to run the GCC test suite anyway, the following command will do so:

```
make -k check
```

The *-k* flag is used to make the test suite run through to completion and not stop at the first failure. The GCC test suite is very comprehensive and is almost guaranteed to generate a few failures. To get a summary of the test suite results, run this:

```
../gcc-3.3.3/contrib/test_summary
```

(For just the summaries, pipe the output through **`grep -A7 Summ`**.)

You can compare your results to those posted to the gcc-testresults mailing list for similar configurations to your own. For an example of how current GCC-3.3.3 should look on i686-pc-linux-gnu, see http://gcc.gnu.org/ml/gcc-testresults/2004-01/msg00826.html.

Note that the results contain:

```
* 1 XPASS (unexpected pass) for g++
* 1 FAIL (unexpected failure) for gcc
* 24 XPASS's for libstdc++
```

The unexpected pass for g++ is due to the use of *--enable-__cxa_atexit*. Apparently not all platforms supported by GCC have support for “__cxa_atexit” in their C libraries, so this test is not always expected to pass.

The 24 unexpected passes for libstdc++ are due to the use of *--enable-clocale=gnu*. This option, which is the correct choice on Glibc-based systems of versions 2.2.5 and above, enables in the GNU C library a locale support that is superior to the otherwise selected *generic* model (which may be applicable if for instance you were using Newlibc, Sun-libc or whatever other libc). The libstdc++ test suite is apparently expecting the *generic* model, hence those tests are not always expected to pass.

Having a few unexpected failures often cannot be avoided. The GCC developers are usually aware of these, but haven't yet gotten around to fixing them. One particular case in point is the filebuf_members test in the C++ standard library testsuite. This test has been observed to fail in some situations, but succeeed in others. In short, unless your results are vastly different from those at the above URL, it is safe to continue.

And finally install the package:

```
make install
```

Note

At this point it is strongly recommended to repeat the sanity check we performed earlier in this chapter. Refer back to the section called “Adjusting the toolchain”[p.37] and repeat the little test compilation. If the result is wrong, then most likely you forgot to apply the above mentioned GCC Specs patch.

The details on this package are found in the section called “Contents of GCC”[p.87].

Binutils-2.14 - Pass 2

```
Approximate build time:  1.5 SBU
Required disk space:     35.6 MB
```

Re-installation of Binutils

Create a separate build directory again:

```
mkdir ../binutils-build
cd ../binutils-build
```

Now prepare Binutils for compilation:

```
../binutils-2.14/configure --prefix=/tools \
    --enable-shared --with-lib-path=/tools/lib
```

The meaning of the new configure option:

- **`--with-lib-path=/tools/lib`**: This tells the configure script to specify the library search path during the compilation of Binutils, resulting in */tools/lib* to be passed to the linker. This prevents the linker from searching through library directories on the host.

Before starting to build Binutils, remember to unset any environment variables that override the default optimization flags.

Compile the package:

```
make
```

Compilation is now complete. As discussed earlier, we don't recommend running the test suites for the temporary tools here in this chapter. If nevertheless you want to run the Binutils test suite, the following command will do so:

```
make check
```

There should be no unexpected failures here, expected failures are fine. Unfortunately, there is no easy way to view the test results summary like there was for the GCC package. However, if a failure occurs here, it should be easy to spot. The output shown will contain something like:

```
make[1]: *** [check-binutils] Error 2
```

And install the package:

```
make install
```

Now prepare the linker for the "Re-adjusting" phase in the next chapter:

```
make -C ld clean
make -C ld LIB_PATH=/usr/lib:/lib
```

Warning

Do not yet remove the Binutils source and build directories. You will need these directories again in the next chapter in the state they are in now.

The details on this package are found in the section called “Contents of Binutils”[p.85].

Gawk-3.1.3

The Gawk package contains programs for manipulating text files.

```
Approximate build time:  0.2 SBU
Required disk space:     16.9 MB
```

Gawk installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Gawk

Prepare Gawk for compilation:

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

Compile the package:

```
make
```

(If you insist on testing the results, then issue: **make check**.)

And install it:

```
make install
```

The details on this package are found in the section called “Contents of Gawk”[p.97].

Coreutils-5.2.1

The Coreutils package contains utilities for showing and setting the basic system characteristics.

```
Approximate build time:  0.9 SBU
Required disk space:     69 MB
```

Coreutils installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed.

Installation of Coreutils

Prepare Coreutils for compilation:

```
DEFAULT_POSIX2_VERSION=199209 ./configure --prefix=/tools
```

This package has an issue when compiled against versions of glibc later than 2.3.2. Some of the Coreutils utilities (such as **head**, **tail** and **sort**) will reject their traditional syntax, a syntax that has been in use for approximately 30 years. This old syntax is so pervasive that compatibility should be preserved until the many places where it is used can be updated. Backwards compatibility is achieved by setting the DEFAULT_POSIX2_VERSION environment variable to "199209" in the above command. If you don't want coreutils to be backwards compatible with the traditional syntax, then simply omit setting the DEFAULT_POSIX2_VERSION environment variable. Realise though, that doing so will mean you'll have to deal with the consequences yourself: patch the many packages that still use the old syntax. We therefore recommend using the instructions exactly as given above.

Compile the package:

```
make
```

(If you insist on testing the results, then issue: **`make RUN_EXPENSIVE_TESTS=yes check`**. The *RUN_EXPENSIVE_TESTS=yes* parameter tells the test suite to run several additional tests that are considered relatively expensive on some platforms but generally are not a problem on Linux.)

And install the package:

```
make install
```

The details on this package are found in the section called “Contents of Coreutils”[p.89].

Bzip2-1.0.2

The Bzip2 package contains programs for compressing and decompressing files. On text files they achieve a much better compression than the traditional **gzip**.

```
Approximate build time:  0.1 SBU
Required disk space:     2.5 MB
```

Bzip2 installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Make.

Installation of Bzip2

The Bzip2 package doesn't contain a **configure** script. Compile and install it with a straightforward:

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

The details on this package are found in the section called “Contents of Bzip2”[p.124].

Gzip-1.3.5

The Gzip package contains programs for compressing and decompressing files.

```
Approximate build time:  0.1 SBU
Required disk space:     2.6 MB
```

Gzip installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of Gzip

Prepare Gzip for compilation:

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

Compile the package:

```
make
```

And install it:

```
make install
```

The details on this package are found in the section called “Contents of Gzip”[p.134].

Diffutils-2.8.1

The Diffutils package contains programs that show the differences between files or directories.

```
Approximate build time:  0.1 SBU
Required disk space:     7.5 MB
```

Diffutils installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Diffutils

Prepare Diffutils for compilation:

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

Compile the package:

```
make
```

And install it:

```
make install
```

The details on this package are found in the section called “Contents of Diffutils”[p.126].

Findutils-4.1.20

The Findutils package contains programs to find files. Processes are provided to recursively search through a directory tree and to create, maintain and search a database (often faster than the recursive find, but unreliable if the database has not been recently updated).

```
Approximate build time:  0.2 SBU
Required disk space:     7.5 MB
```

Findutils installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Findutils

Prepare Findutils for compilation:

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

Compile the package:

```
make
```

(If you insist on testing the results, then issue: **make check**.)

And install the package:

```
make install
```

The details on this package are found in the section called “Contents of Findutils”[p.96].

Make-3.80

The Make package contains a program for compiling large packages.

```
Approximate build time:  0.2 SBU
Required disk space:     8.8 MB
```

Make installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Sed.

Installation of Make

Prepare Make for compilation:

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

Compile the program:

```
make
```

(If you insist on testing the results, then issue: **make check**.)

Then install it and its documentation:

```
make install
```

The details on this package are found in the section called “Contents of Make”[p.138].

Grep-2.5.1

The Grep package contains programs for searching through files.

```
Approximate build time:  0.1 SBU
Required disk space:     5.8 MB
```

Grep installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Make, Sed, Texinfo.

Installation of Grep

Prepare Grep for compilation:

```
./configure --prefix=/tools \
    --disable-perl-regexp --with-included-regex
```

The meaning of the configure options:

- **`--disable-perl-regexp`**: This makes sure that **grep** does not get linked against a PCRE library that may be present on the host and would not be available once we enter the chroot environment.
- **`--with-included-regex`**: This ensures that Grep uses its internal regular expression code. Without this switch, Grep will use the code from Glibc, which is known to be slightly buggy.

Compile the programs:

```
make
```

(If you insist on testing the results, then issue: **`make check`**.)

Then install them and their documentation:

```
make install
```

The details on this package are found in the section called “Contents of Grep”[p.132].

Sed-4.0.9

The Sed package contains a stream editor.

```
Approximate build time:  0.2 SBU
Required disk space:     5.9 MB
```

Sed installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Texinfo.

Installation of Sed

Prepare Sed for compilation:

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

Compile the program:

```
make
```

(If you insist on testing the results, then issue: **make check**.)

Then install it and its documentation:

```
make install
```

The details on this package are found in the section called “Contents of Sed”[p.107].

Gettext-0.14.1

The Gettext package contains utilities for internationalization and localization. These allow programs to be compiled with Native Language Support (NLS), enabling them to output messages in the user's native language.

```
Approximate build time:  0.5 SBU
Required disk space:     67.6 MB
```

Gettext installation depends on: Bash, Binutils, Bison, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Gettext

Prepare Gettext for compilation:

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

Compile the programs:

```
make
```

(If you insist on testing the results, then issue: **make check**. This takes a very long time, around 7 SBUs. Moreover, the Gettext test suite is known to experience failures under certain host conditions -- for example when it finds a Java compiler on the host (but an experimental patch to disable Java is available from the LFS Patches project).)

And install the package:

```
make install
```

The details on this package are found in the section called “Contents of Gettext”[p.109].

Ncurses-5.4

The Ncurses package contains libraries for terminal-independent handling of character screens.

```
Approximate build time:  0.7 SBU
Required disk space:     27.8 MB
```

Ncurses installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Ncurses

Prepare Ncurses for compilation:

```
./configure --prefix=/tools --with-shared \
    --without-debug --without-ada --enable-overwrite
```

The meaning of the configure options:

- **`--without-ada`**: This tells Ncurses not to build its Ada bindings, even if an Ada compiler is installed on the host. This must be done because once we enter the chroot environment, Ada will no longer be available.
- **`--enable-overwrite`**: This tells Ncurses to install its header files into `/tools/include` instead of `/tools/include/ncurses` to ensure that other packages can find the Ncurses headers successfully.

Compile the programs and libraries:

```
make
```

Then install them and their documentation:

```
make install
```

The details on this package are found in the section called “Contents of Ncurses”[p.98].

Patch-2.5.4

The Patch package contains a program for modifying files.

```
Approximate build time:  0.1 SBU
Required disk space:     1.9 MB
```

Patch installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of Patch

Prepare Patch for compilation (the preprocessor flag *-D_GNU_SOURCE* is only needed on the PowerPC platform, on other architectures you can leave it out):

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

Compile the program:

```
make
```

Then install it and its documentation:

```
make install
```

The details on this package are found in the section called “Contents of Patch”[p.140].

Tar-1.13.94

The Tar package contains an archiving program.

```
Approximate build time:  0.2 SBU
Required disk space:     10.3 MB
```

Tar installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Tar

Prepare Tar for compilation:

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

Compile the programs:

```
make
```

(If you insist on testing the results, then issue: **make check**.)

Then install them and their documentation:

```
make install
```

The details on this package are found in the section called “Contents of Tar”[p.150].

Texinfo-4.7

The Texinfo package contains programs for reading, writing, and converting Info documents.

```
Approximate build time:  0.2 SBU
Required disk space:     16.3 MB
```

Texinfo installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Ncurses, Sed.

Installation of Texinfo

Prepare Texinfo for compilation:

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

Compile the programs:

```
make
```

(If you insist on testing the results, then issue: **make check**.)

Then install them and their documentation:

```
make install
```

The details on this package are found in the section called “Contents of Texinfo”[p.117].

Bash-2.05b

The Bash package contains the Bourne-Again SHell.

```
Approximate build time:  1.2 SBU
Required disk space:     27 MB
```

Bash installation depends on: Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Ncurses, Sed.

Installation of Bash

Bash contains several known bugs. Fix these with the following patch:

```
patch -Np1 -i ../bash-2.05b-2.patch
```

Now prepare Bash for compilation:

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

Compile the program:

```
make
```

(If you insist on testing the results, then issue: **`make tests`**.)

Then install it and its documentation:

```
make install
```

And make a link for the programs that use **sh** for a shell:

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

The details on this package are found in the section called “Contents of Bash”[p.121].

Util-linux-2.12a

The Util-linux package contains miscellaneous utility programs. Among them are utilities for handling file systems, consoles, partitions, and messages.

```
Approximate build time:  0.2 SBU
Required disk space:     16 MB
```

Util-linux installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Ncurses, Sed, Zlib.

Installation of Util-linux

Util-linux doesn't use the freshly installed headers and libraries from the /tools directory. This is fixed by altering the configure script:

```
cp configure configure.backup
sed "s@/usr/include@/tools/include@g" configure.backup > configure
```

Prepare Util-linux for compilation:

```
./configure
```

Compile some support routines:

```
make -C lib
```

Since you'll only need a couple of the utilities contained in this package, build just those:

```
make -C mount mount umount
make -C text-utils more
```

Now copy these programs to the temporary tools directory:

```
cp mount/{,u}mount text-utils/more /tools/bin
```

The details on this package are found in the section called “Contents of Util-linux”[p.151].

Perl-5.8.4

The Perl package contains the Practical Extraction and Report Language.

```
Approximate build time:  0.8 SBU
Required disk space:     74 MB
```

Perl installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Perl

First adapt some hard-wired paths to the C library:

```
patch -Np1 -i ../perl-5.8.4-libc-1.patch
```

Perl insists on using the **arch** program to find out the machine type. Create a little script to mimic this command:

```
echo "uname -m" > /tools/bin/arch
chmod 755 /tools/bin/arch
```

Now prepare Perl for compilation (make sure you get the 'IO Fcntl POSIX' right, they are all letters):

```
./configure.gnu --prefix=/tools -Dstatic_ext='IO Fcntl POSIX'
```

The meaning of the configure option:

- **`-Dstatic_ext='IO Fcntl POSIX'`**: This tells Perl to build the minimum set of static extensions needed for installing and testing the Coreutils package in the next chapter.

Compile only the required tools:

```
make perl utilities
```

Then copy these tools and their libraries:

```
cp perl pod/pod2man /tools/bin
mkdir -p /tools/lib/perl5/5.8.4
cp -R lib/* /tools/lib/perl5/5.8.4
```

The details on this package are found in the section called “Contents of Perl”[p.115].

Stripping

The steps in this section are optional, but if your LFS partition is rather small, you will be glad to learn that you can remove some unnecessary things. The executables and libraries you have built so far contain about 130 MB of unneeded debugging symbols. Remove those symbols with:

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

The last of the above commands will skip some twenty files, reporting that it doesn't recognize their file format. Most of them are scripts instead of binaries.

Take care *not* to use *--strip-unneeded* on the libraries -- the static ones would be destroyed and you would have to build the three toolchain packages all over again.

To save another 30 MB, you can remove all the documentation:

```
rm -rf /tools/{doc,info,man}
```

You will now need to have at least 850 MB of free space on your LFS file system to be able to build and install Glibc in the next phase. If you can build and install Glibc, you can build and install the rest too.

Part III. Building the LFS system

Chapter 6. Installing basic system software

Introduction

In this chapter we enter the building site, and start constructing our LFS system in earnest. That is, we chroot into our temporary mini Linux system, create some auxiliary things, and then start installing all the packages, one by one.

The installation of all this software is pretty straightforward, and you will probably think it would be much shorter to give here the generic installation instructions and explain in full only the installation of those packages that require an alternate method. Although we agree with that, we nevertheless choose to give the full instructions for each and every package, simply to minimize the possibilities for mistakes.

The key to learning what makes a Linux system work is to know what each package is used for and why the user (or the system) needs it. For this purpose for every installed package a summary of its content is given followed by concise descriptions of each program and library it installed.

If you plan to use compiler optimizations in this chapter, take a look at the optimization hint at http://www.linuxfromscratch.org/hints/downloads/files/optimization.txt. Compiler optimizations can make a program run slightly faster, but they may also cause compilation difficulties and even problems when running the program. If a package refuses to compile when using optimization, try to compile it without optimization and see if the problem goes away. Even if the package does compile when using optimization, there is the risk it may have been compiled incorrectly due to complex interactions between the code and build tools. In short, the small potential gains achieved in using compiler optimization are generally outweighed by the risk. First time builders of LFS are encouraged to build without custom optimizations. Your system will still be very fast and very stable at the same time.

The order in which packages are installed in this chapter has to be strictly followed, to ensure that no program gets a path referring to `/tools` hard-wired into it. For the same reason, *do not* compile packages in parallel. Compiling in parallel may save you some time (especially on dual-CPU machines), but it could result in a program containing a hard-wired path to `/tools`, which will cause the program to stop working when that directory is removed.

Before the installation instructions each installation page gives some information about the package: a concise description of what it contains, approximately how long it will take to build it, how much disk space it needs during this building process, the official download location of the package (in case you just want to update a few of them), and which other packages it needs in order to be built successfully. After the installation instructions follows a list of programs and libraries that the package installs, together with a series of short descriptions of these.

If you wish to keep track of which package installs what files, you may want to use a package manager. For a general overview of package managers have a look at http://www.linuxfromscratch.org/blfs/view/cvs/introduction/important.html. And for a package management method specifically geared towards LFS see http://www.linuxfromscratch.org/hints/downloads/files/more_control_and_pkg_man.txt.

Mounting the proc and devpts file systems

In order for certain programs to function properly, the *proc* and *devpts* file systems must be available within the chroot environment. The proc file system is the process information pseudo file system through which the kernel provides information about the status of the system. And the devpts file system is nowadays the most common way pseudo terminals (PTYs) are implemented. Since kernel version 2.4, a file system can be mounted as many times and in as many places as you like, thus it's not a problem that these file systems are already mounted on your host system, especially so because they are virtual file systems.

First become *root*, as only *root* can mount file systems in unusual places. Then check again that the LFS environment variable is set correctly by running **`echo $LFS`** and making sure it shows the path to your LFS partition's mount point, which is `/mnt/lfs` if you followed our example.

Now make the mount points for these filesystems:

```
mkdir -p $LFS/{proc,dev/pts}
```

Mount the *proc* file system with:

```
mount proc $LFS/proc -t proc
```

And mount the *devpts* file system with:

```
mount devpts $LFS/dev/pts -t devpts
```

This last command might fail with an error like:

```
filesystem devpts not supported by kernel
```

The most likely cause for this is that your host system's kernel was compiled without support for the devpts file system (you can check which file systems your kernel supports with **cat /proc/filesystems**, for example). A few PTYs are needed to be able to run the suites for Binutils and GCC later on. If your kernel does not support devpts, do not worry, there is another way to get them working inside the chroot environment. We'll cover this shortly in the Make_devices[p.73] section.

Remember that if for any reason you stop working on your LFS, and start again later, it's important to check that these file systems are mounted again before entering the chroot environment, otherwise problems could occur.

Entering the chroot environment

It is time to enter the chroot environment in order to begin building and installing your final LFS system. Still as *root* run the following command to enter the small world that is, at the moment, populated with only the temporary tools:

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

The *-i* option given to the **env** command will clear all variables of the chroot environment. After that, only the HOME, TERM, PS1 and PATH variables are set again. The TERM=$TERM construct will set the TERM variable inside chroot to the same value as outside chroot; this variable is needed for programs like **vim** and **less** to operate properly. If you need other variables present, such as CFLAGS or CXXFLAGS, this is a good place to set them again.

From this point on there's no need to use the LFS variable anymore, because everything you do will be restricted to the LFS file system -- since what the shell thinks is `/` is actually the value of `$LFS`, which was passed to the chroot command.

Notice that `/tools/bin` comes last in the PATH. This means that a temporary tool will not be used any more as soon as its final version is installed. Well, at least when the shell doesn't remember the locations of executed binaries -- for this reason hashing is switched off by passing the *+h* option to **bash**.

You have to make sure all the commands in the rest of this chapter and in the following chapters are run from within the chroot environment. If you ever leave this environment for any reason (rebooting for example), you must remember to first mount the proc and devpts file systems (discussed in the previous section) *and* enter chroot again before continuing with the installations.

Note that the bash prompt will say “I have no name!” This is normal, as the `/etc/passwd` file has not been created yet.

Changing ownership

Right now the `/tools` directory is owned by the user *lfs*, a user that exists only on your host system. Although you will probably want to delete the `/tools` directory once you have finished your LFS system, you may want to keep it around, for example to build more LFS systems. But if you keep the `/tools` directory as it is, you end up with files owned by a user ID without a corresponding account. This is dangerous because a user account created later on could get this same user ID and would suddenly own the `/tools` directory and all the files therein, thus exposing these files to possible malicious manipulation.

To avoid this issue, you could add the *lfs* user to your new LFS system later on when creating the `/etc/passwd` file, taking care to assign it the same user and group IDs as on your host system. Alternatively, you can (and the book assumes you do) assign the contents of the `/tools` directory to user *root* by running the following command:

```
chown -R 0:0 /tools
```

The command uses “0:0” instead of “root:root”, because **`chown`** is unable to resolve the name “root” until the password file has been created.

Creating directories

Let's now create some structure in our LFS file system. Let's create a directory tree. Issuing the following commands will create a more or less standard tree:

```
mkdir -p /{bin,boot,dev/{pts,shm},etc/opt,home,lib,mnt,proc}
mkdir -p /{root,sbin,srv,tmp,usr/local,var,opt}
mkdir -p /media/{floppy,cdrom}
mkdir /usr/{bin,include,lib,sbin,share,src}
ln -s share/{man,doc,info} /usr
mkdir /usr/share/{doc,info,locale,man}
mkdir /usr/share/{misc,terminfo,zoneinfo}
mkdir /usr/share/man/man{1,2,3,4,5,6,7,8}
mkdir /usr/local/{bin,etc,include,lib,sbin,share,src}
ln -s share/{man,doc,info} /usr/local
mkdir /usr/local/share/{doc,info,locale,man}
mkdir /usr/local/share/{misc,terminfo,zoneinfo}
mkdir /usr/local/share/man/man{1,2,3,4,5,6,7,8}
mkdir /var/{lock,log,mail,run,spool}
mkdir -p /var/{tmp,opt,cache,lib/misc,local}
mkdir /opt/{bin,doc,include,info}
mkdir -p /opt/{lib,man/man{1,2,3,4,5,6,7,8}}
```

Directories are, by default, created with permission mode 755, but this isn't desirable for all directories. We will make two changes: one to the home directory of *root*, and another to the directories for temporary files.

```
chmod 0750 /root
chmod 1777 /tmp /var/tmp
```

The first mode change ensures that not just anybody can enter the `/root` directory -- the same as a normal user would do with his or her home directory. The second mode change makes sure that any user can write to the `/tmp` and `/var/tmp` directories, but cannot remove other users' files from them. The latter is prohibited by the so-called “sticky bit” -- the highest bit in the 1777 bit mask.

FHS compliance note

We have based our directory tree on the FHS standard (available at http://www.pathname.com/fhs/). Besides the above created tree this standard stipulates the existence of `/usr/local/games` and `/usr/share/games`, but we don't much like these for a base system. However, feel free to make your system FHS-compliant. As to the structure of the `/usr/local/share` subdirectory, the FHS isn't precise, so we created here the directories that we think are needed.

Creating essential symlinks

Some programs hard-wire paths to programs which don't exist yet. In order to satisfy these programs, we create a number of symbolic links which will be replaced by real files throughout the course of this chapter when we're installing all the software.

```
ln -s /tools/bin/{bash,cat,pwd,stty} /bin
ln -s /tools/bin/perl /usr/bin
ln -s /tools/lib/libgcc_s.so.1 /usr/lib
ln -s bash /bin/sh
```

Creating the passwd, group and log files

In order for *root* to be able to login and for the name “root” to be recognized, there need to be relevant entries in the `/etc/passwd` and `/etc/group` files.

Create the `/etc/passwd` file by running the following command:

```
cat > /etc/passwd << "EOF"
root:x:0:0:root:/root:/bin/bash
EOF
```

The actual password for *root* (the “x” here is just a placeholder) will be set later.

Create the `/etc/group` file by running the following command:

```
cat > /etc/group << "EOF"
root:x:0:
bin:x:1:
sys:x:2:
kmem:x:3:
tty:x:4:
tape:x:5:
daemon:x:6:
floppy:x:7:
disk:x:8:
lp:x:9:
dialout:x:10:
audio:x:11:
EOF
```

The created groups aren't part of any standard -- they are some of the groups that the **make_devices** script in the next section uses. The LSB (Linux Standard Base) recommends only that, beside the group “root” with a GID of 0, a group “bin” with a GID of 1 be present. All other group names and GIDs can be chosen freely by the system administrator, since well-written packages don't depend on GID numbers but use the group's name.

To get rid of the “I have no name!” prompt, we will start a new shell. Since we installed a full Glibc in Chapter 5[p.26], and have just created the `/etc/passwd` and `/etc/group` files, user name and group name resolution will now work.

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

Note the use of the *+h* directive. This tells **bash** not to use its internal path hashing. Without this directive, **bash** would remember the paths to binaries it has executed. Since we want to use our newly compiled binaries as soon as they are installed, we turn off this function for the duration of this chapter.

The **login**, **agetty** and **init** programs (and some others) use a number of log files to record information such as who was logged into the system and when. These programs, however, won't write to the log files if they don't already exist. Initialize the log files and give them their proper permissions:

```
touch /var/run/utmp /var/log/{btmp,lastlog,wtmp}
chmod 644 /var/run/utmp /var/log/{btmp,lastlog,wtmp}
```

The `/var/run/utmp` file records the users that are currently logged in. The `/var/log/wtmp` file records all logins and logouts. The `/var/log/lastlog` file records for each user when he or she last logged in. The `/var/log/btmp` file records the bad login attempts.

Creating devices with Make_devices-1.2

The Make_devices package contains a script for creating device nodes.

```
Approximate build time:  1 SBU
Required disk space:     160 KB
```

For its installation Make_devices depends on: Bash, Bzip2, Coreutils.

Making devices

Note that unpacking the `make_devices-1.2.bz2` file doesn't create a directory for you to **cd** into, as the file contains only a shell script.

Install the `make_devices` script:

```
bzcat make_devices-1.2.bz2 > /dev/make_devices
chmod 754 /dev/make_devices
```

Device nodes are special files: things that can generate or receive data. They usually correspond to physical pieces of hardware. Device nodes can be created by issuing commands of the form: **mknod -m mode name type major minor**. In such a command, *mode* is the usual octal read/write/execute permissions triplet, and *name* is the name of the device file to be created. It may seem surprising, but the device name is actually arbitrary, except that most programs rely on devices such as `/dev/null` having their usual names. The remaining three parameters tell the kernel what device the node actually refers to. The *type* is a letter, either b or c, indicating whether the device is accessed in blocks (such as a hard disk) or character by character (such as the console). And *major* and *minor* are numbers, together forming a code that identifies the device to the kernel. A list of the currently assigned device numbers for Linux can be found in the file `devices.txt` in the `Documentation` subdirectory of the kernel sources.

Note that the same major/minor combination is usually assigned to both a block and a character device. These are, however, completely unrelated devices that cannot be interchanged. A device is identified by the type/major/minor triple, not just the major/minor pair, so when creating a device node it is important to choose the correct *type* of device.

Because looking up the type/major/minor triples and using **mknod** manually is tedious and error-prone, the `make_devices` script has been created. It contains a whole series of **mknod** commands, one for each device, complete with recommended name, permissions and group assignment. It has been set up so that only a minimal set of commonly used devices is enabled and the other lines are commented out. You should open `make_devices` in an editor and customize it to your needs. This takes some time, but is very simple. When you are satisfied, run the script to create the device files:

Warning

Failure to properly edit the **make_devices** to match your systems's setup (eg. number of partitions) can lead to boot errors.

```
cd /dev
./make_devices
```

If you had success with mounting the devpts file system earlier in the section called “Mounting the proc and devpts file systems”[p.67], you can continue with the next section. If you were unable to mount devpts, you will have to create a few static ptyXX and ttyXX device nodes instead. To do this, open `make_devices` in your editor, go to the section “Pseudo-TTY masters” and enable a few ptyXX devices -- a handful are enough to enable the test suites to run, but if you plan to run a kernel without devpts support you will probably need many more (every xterm, ssh connection, telnet connection, and the like, uses one of these pseudo terminals). In the immediately following section “Pseudo-TTY slaves”, enable the corresponding ttyXX devices. When you are done, rerun **./make_devices** from inside `/dev` to have it create the new devices.

Contents of Make_devices

Installed script: make_devices

Short description

make_devices is a script for creating a basic set of static device nodes, usually residing in the `/dev` directory.

Linux-2.4.26 headers

```
Approximate build time:  0.1 SBU
Required disk space:     186 MB
```

Installation of the kernel headers

We won't be compiling a new kernel yet -- we'll do that when we have finished the installation of all the packages. But the libraries installed in the next section need to refer to the kernel header files in order to know how to interface with the kernel. Instead of unpacking the kernel sources again, making the version file and the symlinks and so on, we will simply copy the headers from the temporary tools directory in one swoop:

```
cp -a /tools/include/{asm,asm-generic,linux} /usr/include
```

A few kernel header files refer to the `autoconf.h` header file. Since we have not yet configured the kernel, we need to create this file ourselves in order to avoid a compilation failure of Sysklogd. Create an empty `autoconf.h` file with:

```
touch /usr/include/linux/autoconf.h
```

Why we copy the kernel headers

In the past it was common practice to symlink the `/usr/include/{linux,asm}` directories to `/usr/src/linux/include/{linux,asm}`. This was a *bad* practice, as the following extract from a post by Linus Torvalds to the Linux Kernel Mailing List points out:

```
I would suggest that people who compile new kernels should:

 - not have a single symbolic link in sight (except the one that the
   kernel build itself sets up, namely the “linux/include/asm”
   symlink that is only used for the internal kernel compile itself)

And yes, this is what I do. My /usr/src/linux still has the old 2.2.13
header files, even though I haven't run a 2.2.13 kernel in a _loong_
time. But those headers were what Glibc was compiled against, so those
headers are what matches the library object files.

And this is actually what has been the suggested environment for at
least the last five years. I don't know why the symlink business keeps
on living on, like a bad zombie. Pretty much every distribution still
has that broken symlink, and people still remember that the linux
sources should go into “/usr/src/linux” even though that hasn't
been true in a _loong_ time.
```

The essential part is where Linus states that the header files should be *the ones which Glibc was compiled against*. These are the headers that should be used when you later compile other packages, as they are the ones that match the object-code library files. By copying the headers, we ensure that they remain available if later you upgrade your kernel.

Note, by the way, that it is perfectly all right to have the kernel sources in `/usr/src/linux`, as long as you don't have the `/usr/include/{linux,asm}` symlinks.

Man-pages-1.66

The Man-pages package contains over 1200 manual pages.

```
Approximate build time:  0.1 SBU
Required disk space:     15 MB
```

For its installation Man-pages depends on: Bash, Coreutils, Make.

Installation of Man-pages

Install Man-pages by running:

```
make install
```

Contents of Man-pages

Installed files: various manual pages

Short description

Examples of provided *manual pages* are the pages describing all the C and C++ functions, important device files, and important configuration files.

Glibc-2.3.3-lfs-5.1

The Glibc package contains the main C library. This library provides all the basic routines for allocating memory, searching directories, opening and closing files, reading and writing them, string handling, pattern matching, arithmetic, and so on.

```
Approximate build time:  12.3 SBU
Required disk space:     784 MB
```

Glibc installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Gettext, Grep, Make, Perl, Sed, Texinfo.

Installation of Glibc

The Glibc build system is very well self-contained and will install perfectly, even though our compiler specs file and linker are still pointing at `/tools`. We cannot adjust the specs and linker before the Glibc install, because the Glibc autoconf tests would then give bogus results and thus defeat our goal of achieving a clean build.

Before starting to build Glibc, remember to unset any environment variables that override the default optimization flags.

The Glibc documentation recommends building Glibc outside of the source directory in a dedicated build directory:

```
mkdir ../glibc-build
cd ../glibc-build
```

Now prepare Glibc for compilation:

```
../glibc-2.3.3-lfs-5.1/configure --prefix=/usr \
    --disable-profile --enable-add-ons=linuxthreads \
    --libexecdir=/usr/lib --with-headers=/usr/include \
    --without-cvs
```

The meaning of the new configure options:

- **`--libexecdir=/usr/lib`**: This changes the location of the `pt_chown` program from its default of `/usr/libexec` to `/usr/lib`. The use of *libexec* is considered not to be FHS-compliant because the FHS doesn't even mention it.
- **`--with-headers=/usr/include`**: This ensures that the kernel headers in `/usr/include` are used for this build. If you don't pass this switch then the headers from `/tools/include` are used which of course is not ideal (although they should be identical). Using this switch has the advantage that you will be informed immediately should you have forgotten to install the kernel headers into `/usr/include`.

Compile the package:

```
make
```

Important

The test suite for Glibc in this section is considered *critical*. Our advice is to not skip it under any circumstance.

Test the results:

```
make check
```

The test suite notes from the section called “Glibc-2.3.3-lfs-5.1”[p.34] are still very much appropriate here. Be sure to refer back there should you have any doubts.

Though it is a harmless message, the install stage of Glibc will complain about the absence of `/etc/ld.so.conf`. Fix this annoying little warning with:

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

And install the package:

```
make install
```

The locales that can make your system respond in a different language weren't installed by the above command. Do it with this:

```
make localedata/install-locales
```

An alternative to running the previous command is to install only those locales which you need or want. This can be achieved by using the **localedef** command. Information on this can be found in the `INSTALL` file in the Glibc source. However, there are a number of locales that are essential for the tests of future packages to pass, in particular, the *libstdc++* tests from GCC. The following instructions, instead of the install-locales target above, will install the minimum set of locales necessary for the tests to run successfully:

```
mkdir -p /usr/lib/locale
localedef -i de_DE -f ISO-8859-1 de_DE
localedef -i de_DE@euro -f ISO-8859-15 de_DE@euro
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 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 it_IT -f ISO-8859-1 it_IT
localedef -i ja_JP -f EUC-JP ja_JP
```

Finally, build the linuxthreads man pages:

```
make -C ../glibc-2.3.3-lfs-5.1/linuxthreads/man
```

And install these pages:

```
make -C ../glibc-2.3.3-lfs-5.1/linuxthreads/man install
```

Configuring Glibc

We need to create the `/etc/nsswitch.conf` file, because, although Glibc provides defaults when this file is missing or corrupt, the Glibc defaults don't work well with networking. Also, our time zone needs to be set up.

Create a new file `/etc/nsswitch.conf` by running the following:

```
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
```

To find out what time zone you're in, run the following script:

```
tzselect
```

When you've answered a few questions about your location, the script will output the name of your time zone, something like *EST5EDT* or *Canada/Eastern*. Then create the `/etc/localtime` file by running:

```
cp --remove-destination /usr/share/zoneinfo/Canada/Eastern /etc/localtime
```

The meaning of the option:

- **`--remove-destination`**: This is needed to force removal of the already existing symbolic link. The reason why we copy instead of symlink is to cover the situation where `/usr` is on a separate partition. This could matter, for example, when booted into single user mode.

Of course, instead of *Canada/Eastern*, fill in the name of the time zone that the **tzselect** script gave you.

Configuring Dynamic Loader

By default, the dynamic loader (`/lib/ld-linux.so.2`) searches through `/lib` and `/usr/lib` for dynamic libraries that are needed by programs when you run them. However, if there are libraries in directories other than `/lib` and `/usr/lib`, you need to add them to the `/etc/ld.so.conf` file for the dynamic loader to find them. Two directories that are commonly known to contain additional libraries are `/usr/local/lib` and `/opt/lib`, so we add those directories to the dynamic loader's search path.

Create a new file `/etc/ld.so.conf` by running the following:

```
cat > /etc/ld.so.conf << "EOF"
# Begin /etc/ld.so.conf

/usr/local/lib
/opt/lib

# End /etc/ld.so.conf
EOF
```

Contents of Glibc

Installed programs: catchsegv, gencat, getconf, getent, glibcbug, iconv, iconvconfig, ldconfig, ldd, lddlibc4, locale, localedef, mtrace, nscd, nscd_nischeck, pcprofiledump, pt_chown, rpcgen, rpcinfo, sln, sprof, tzselect, xtrace, zdump and zic

Installed libraries: ld.so, libBrokenLocale.[a,so], libSegFault.so, libanl.[a,so], libbsd-compat.a, libc.[a,so], libc_nonshared.a, libcrypt.[a,so], libdl.[a,so], libg.a, libieee.a, libm.[a,so], libmcheck.a, libmemusage.so, libnsl.a, libnss_compat.so, libnss_dns.so, libnss_files.so, libnss_hesiod.so, libnss_nis.so, libnss_nisplus.so, libpcprofile.so, libpthread.[a,so], libresolv.[a,so], librpcsvc.a, librt.[a,so], libthread_db.so and libutil.[a,so]

Short descriptions

catchsegv can be used to create a stack trace when a program terminates with a segmentation fault.

gencat generates message catalogues.

getconf displays the system configuration values for file system specific variables.

getent gets entries from an administrative database.

glibcbug creates a bug report and mails it to the bug email address.

iconv performs character set conversion.

iconvconfig creates fastloading iconv module configuration file.

ldconfig configures the dynamic linker runtime bindings.

ldd reports which shared libraries are required by each given program or shared library.

lddlibc4 assists ldd with object files.

locale is a Perl program that tells the compiler to enable or disable the use of POSIX locales for built-in operations.

localedef compiles locale specifications.

mtrace...

nscd is a name service cache daemon providing a cache for the most common name service requests.

nscd_nischeck checks whether or not secure mode is necessary for NIS+ lookup.

pcprofiledump dumps information generated by PC profiling.

pt_chown is a helper program for grantpt to set the owner, group and access permissions of a slave pseudo terminal.

rpcgen generates C code to implement the RPC protocol.

rpcinfo makes an RPC call to an RPC server.

sln is used to make symbolic links. The program is statically linked, so it is useful for making symbolic links to dynamic libraries if the dynamic linking system for some reason is nonfunctional.

sprof reads and displays shared object profiling data.

tzselect asks the user about the location of the system and reports the corresponding time zone description.

xtrace traces the execution of a program by printing the currently executed function.

zdump is the time zone dumper.

zic is the time zone compiler.

ld.so is the helper program for shared library executables.

libBrokenLocale is used by programs, such as Mozilla, to solve broken locales.

libSegFault is a segmentation fault signal handler. It tries to catch segfaults.

libanl is an asynchronous name lookup library.

libbsd-compat provides the portability needed in order to run certain BSD programs under Linux.

libc is the main C library -- a collection of commonly used functions.

libcrypt is the cryptography library.

libdl is the dynamic linking interface library.

libg is a runtime library for g++.

libieee is the IEEE floating point library.

libm is the mathematical library.

libmcheck contains code run at boot.

libmemusage is used by memusage to help collect information about the memory usage of a program.

libnsl is the network services library.

libnss* are the Name Service Switch libraries, containing functions for resolving host names, user names, group names, aliases, services, protocols,and the like.

libpcprofile contains profiling functions used to track the amount of CPU time spent in which source code lines.

libpthread is the POSIX threads library.

libresolv contains functions for creating, sending, and interpreting packets to the Internet domain name servers.

librpcsvccontains functions providing miscellaneous RPC services.

librt contains functions providing most of the interfaces specified by the POSIX.1b Realtime Extension.

libthread_db contains functions useful for building debuggers for multi-threaded programs.

libutil contains code for "standard" functions used in many different Unix utilities.

Re-adjusting the toolchain

Now that the new and final C libraries have been installed, it's time to adjust our toolchain again. We'll adjust it so that it will link any newly compiled program against these new libraries. This is in fact the same thing we did in the "Adjusting" phase in the beginning of the previous chapter, even though it looks like the reverse: then we guided the chain from the host's `/{,usr/}lib` to the new `/tools/lib`, now we guide it from that same `/tools/lib` to the LFS's `/{,usr/}lib`.

First we adjust the linker. For this we retained the source and build directories from the second pass over Binutils. Install the adjusted linker by running the following from within the `binutils-build` directory:

```
make -C ld INSTALL=/tools/bin/install install
```

Note

If you somehow missed the earlier warning to retain the Binutils source and build directories from the second pass in Chapter 5[p.26], or otherwise accidentally deleted them or just don't have access to them, don't worry, all is not lost. Just ignore the above command. The result will be that the next package, Binutils, will link against the C libraries in `/tools` rather than in `/{,usr/}lib`. This is not ideal, however, our testing has shown that the resulting Binutils program binaries should be identical.

From now on every compiled program will link *only* against the libraries in `/usr/lib` and `/lib`. The extra *INSTALL=/tools/bin/install* is needed because the Makefile created during the second pass still contains the reference to `/usr/bin/install`, which we obviously haven't installed yet. Some host distributions contain a `ginstall` symbolic link which takes precedence in the Makefile and thus can cause a problem here. The above command takes care of this also.

You can now remove the Binutils source and build directories.

The next thing to do is to amend our GCC specs file so that it points to the new dynamic linker. Just like earlier on, we use a sed to accomplish this:

```
SPECFILE=/tools/lib/gcc-lib/*/*/specs &&
sed -e 's@ /tools/lib/ld-linux.so.2@ /lib/ld-linux.so.2@g' \
    $SPECFILE > newspecfile &&
mv -f newspecfile $SPECFILE &&
unset SPECFILE
```

Again, cutting and pasting the above is recommended. And just like before, it is a good idea to visually inspect the specs file to verify the intended change was actually made.

Important

If you are working on a platform where the name of the dynamic linker is something other than `ld-linux.so.2`, you *must* substitute `ld-linux.so.2` with the name of your platform's dynamic linker in the above commands. Refer back to the section called "Toolchain technical notes"[p.27] if necessary.

Caution

It is imperative at this point to stop and ensure that the basic functions (compiling and linking) of the adjusted toolchain are working as expected. For this we are going to perform a simple sanity check:

```
echo 'main(){}' > dummy.c
cc dummy.c
readelf -l a.out | grep ': /lib'
```

If everything is working correctly, there should be no errors, and the output of the last command will be (allowing for platform specific differences in dynamic linker name):

```
[Requesting program interpreter: /lib/ld-linux.so.2]
```

Note especially that `/lib` is now the prefix of our dynamic linker.

If you did not receive the output as shown above, or received no output at all, then something is seriously wrong. You will need to investigate and retrace your steps to find out where the problem is and correct it. There is no point in continuing until this is done. Most likely something went wrong with the specs file amendment above.

Once you are satisfied that all is well, clean up the test files:

```
rm dummy.c a.out
```

Binutils-2.14

The Binutils package contains a linker, an assembler, and other tools for handling object files.

```
Approximate build time:  1.4 SBU
Required disk space:     167 MB
```

Binutils installation depends on: Bash, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed, Texinfo.

Installation of Binutils

Now is an appropriate time to verify that your pseudo terminals (PTYs) are working properly inside the chroot environment. We will again quickly check that everything is set up correctly by performing a simple test:

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

If you receive the message:

```
The system has no more ptys.  Ask your system administrator to create more.
```

Your chroot environment is not set up for proper PTY operation. In this case there is no point in running the test suites for Binutils and GCC until you are able to resolve the issue. Please refer back to the section called “Mounting the proc and devpts file systems”[p.67] and the Make_devices[p.73] section and perform the recommended steps to fix the problem.

This package is known to behave badly when you have changed its default optimization flags (including the -march and -mcpu options). Therefore, if you have defined any environment variables that override default optimizations, such as CFLAGS and CXXFLAGS, we recommend un-setting or modifying them when building Binutils.

The Binutils documentation recommends building Binutils outside of the source directory in a dedicated build directory:

```
mkdir ../binutils-build
cd ../binutils-build
```

Now prepare Binutils for compilation:

```
../binutils-2.14/configure --prefix=/usr --enable-shared
```

Compile the package:

```
make tooldir=/usr
```

Normally, the *tooldir* (the directory where the executables end up) is set to $(exec_prefix)/$(target_alias), which expands into, for example, `/usr/i686-pc-linux-gnu`. Since we only build for our own system, we don't need this target specific directory in `/usr`. That setup would be used if the system was used to cross-compile (for example compiling a package on an Intel machine that generates code that can be executed on PowerPC machines).

Important

The test suite for Binutils in this section is considered *critical*. Our advice is to not skip it under any circumstances.

Test the results:

```
make check
```

The test suite notes from the section called “Binutils-2.14 - Pass 2”[p.45] are still very much appropriate here. Be sure to refer back there should you have any doubts.

Install the package:

```
make tooldir=/usr install
```

Install the *libiberty* header file that is needed by some packages:

```
cp ../binutils-2.14/include/libiberty.h /usr/include
```

Contents of Binutils

Installed programs: addr2line, ar, as, c++filt, gprof, ld, nm, objcopy, objdump, ranlib, readelf, size, strings and strip

Installed libraries: libiberty.a, libbfd.[a,so] and libopcodes.[a,so]

Short descriptions

addr2line translates program addresses to file names and line numbers. Given an address and the name of an executable, it uses the debugging information in the executable to figure out which source file and line number are associated with the address.

ar creates, modifies, and extracts from archives. An archive is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called members of the archive).

as is an assembler. It assembles the output of gcc into object files.

c++filt is used by the linker to de-mangle C++ and Java symbols, to keep overloaded functions from clashing.

gprof displays call graph profile data.

ld is a linker. It combines a number of object and archive files into a single file, relocating their data and tying up symbol references.

nm lists the symbols occurring in a given object file.

objcopy is used to translate one type of object file into another.

objdump displays information about the given object file, with options controlling what particular information to display. The information shown is mostly only useful to programmers who are working on the compilation tools.

ranlib generates an index of the contents of an archive, and stores it in the archive. The index lists all the symbols defined by archive members that are relocatable object files.

readelf displays information about elf type binaries.

size lists the section sizes -- and the grand total -- for the given object files.

strings outputs, for each given file, the sequences of printable characters that are of at least the specified length (defaulting to 4). For object files it prints, by default, only the strings from the initializing and loading sections. For other types of files it scans the whole file.

strip discards symbols from object files.

libiberty contains routines used by various GNU programs, including getopt, obstack, strerror, strtol and strtoul.

libbfd is the Binary File Descriptor library.

libopcodes is a library for dealing with opcodes. It is used for building utilities like objdump. Opcodes are the “readable text” versions of instructions for the processor.

GCC-3.3.3

The GCC package contains the GNU compiler collection, which includes the C and C++ compilers.

```
Approximate build time:  11.7 SBU
Required disk space:     294 MB
```

GCC installation depends on: Bash, Binutils, Coreutils, Diffutils, Findutils, Gawk, Gettext, Glibc, Grep, Make, Perl, Sed, Texinfo.

Installation of GCC

This package is known to behave badly when you have changed its default optimization flags (including the -march and -mcpu options). Therefore, if you have defined any environment variables that override default optimizations, such as CFLAGS and CXXFLAGS, we recommend un-setting or modifying them when building GCC.

Unpack the GCC-core *and* the GCC-g++ tarball -- they will unfold into the same directory. You should likewise extract the GCC-testsuite package. The full GCC package contains even more compilers. Instructions for building these can be found at http://www.linuxfromscratch.org/blfs/view/stable/general/gcc.html.

First apply only the No-Fixincludes patch (and *not* the Specs patch) also used in the previous chapter:

```
patch -Np1 -i ../gcc-3.3.3-no_fixincludes-1.patch
```

Now apply a sed substitution that will suppress the installation of `libiberty.a`. We want to use the version of `libiberty.a` provided by Binutils:

```
sed -i 's/install_to_$(INSTALL_DEST) //' libiberty/Makefile.in
```

The GCC documentation recommends building GCC outside of the source directory in a dedicated build directory:

```
mkdir ../gcc-build
cd ../gcc-build
```

Now prepare GCC for compilation:

```
../gcc-3.3.3/configure --prefix=/usr \
    --enable-shared --enable-threads=posix \
    --enable-__cxa_atexit --enable-clocale=gnu \
    --enable-languages=c,c++
```

Compile the package:

```
make
```

Important

The test suite for GCC in this section is considered *critical*. Our advice is to not skip it under any circumstance.

Test the results, but don't stop at errors (you'll remember the few known ones):

```
make -k check
```

The test suite notes from the section called “GCC-3.3.3 - Pass 2”[p.42] are still very much appropriate here. Be sure to refer back there should you have any doubts.

Now install the package:

```
make install
```

Some packages expect the C PreProcessor to be installed in the `/lib` directory. To support those packages, create this

symlink:

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

Many packages use the name **cc** to call the C compiler. To satisfy those packages, create a symlink:

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

Note

At this point it is strongly recommended to repeat the sanity check we performed earlier in this chapter. Refer back to the section called “Re-adjusting the toolchain”[p.82] and repeat the check. If the results are wrong, then most likely you erroneously applied the GCC Specs patch from Chapter 5[p.26].

Contents of GCC

Installed programs: c++, cc (link to gcc), cc1, cc1plus, collect2, cpp, g++, gcc, gccbug, and gcov

Installed libraries: libgcc.a, libgcc_eh.a, libgcc_s.so, libstdc++.[a,so] and libsupc++.a

Short descriptions

cpp is the C preprocessor. It is used by the compiler to have the #include and #define and such statements expanded in the source files.

g++ is the C++ compiler.

gcc is the C compiler. It is used to translate the source code of a program into assembly code.

gccbug is a shell script used to help create good bug reports.

gcov is a coverage testing tool. It is used to analyze programs to find out where optimizations will have the most effect.

libgcc* contains run-time support for gcc.

libstdc++ is the standard C++ library. It contains many frequently-used functions.

libsupc++ provides supporting routines for the c++ programming language.

Coreutils-5.2.1

The Coreutils package contains utilities for showing and setting the basic system characteristics.

```
Approximate build time:  0.9 SBU
Required disk space:     69 MB
```

Coreutils installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed.

Installation of Coreutils

Normally the functionality of **uname** is somewhat broken, in that the *-p* switch always returns “unknown”. The following patch fixes this behavior for Intel architectures:

```
patch -Np1 -i ../coreutils-5.2.1-uname-1.patch
```

We do not want Coreutils to install its version of the **hostname** program, because it is inferior to the version provided by Net-tools. Prevent its installation by applying a patch:

```
patch -Np1 -i ../coreutils-5.2.1-hostname-1.patch
```

Now prepare Coreutils for compilation:

```
DEFAULT_POSIX2_VERSION=199209 ./configure --prefix=/usr
```

Compile the package:

```
make
```

The test suite of Coreutils makes several assumptions about the presence of files and users that aren't valid this early in the LFS build. We will therefore have to set up a few things before being able to run the tests. If you choose not to run these tests, skip down to “Install the package”.

To be able to run the full test suite, the **su** program needs to be installed. We didn't bother to install this little program in Chapter 5[p.26] because it requires root privileges, so do it now:

```
make install-root
```

Create a 'table of mounted filesystems' file with:

```
touch /etc/mtab
```

And create two dummy groups and a dummy user name:

```
echo "dummy1:x:1000" >> /etc/group
echo "dummy2:x:1001:dummy" >> /etc/group
echo "dummy:x:1000:1000:::/bin/bash" >> /etc/passwd
```

Now you're all set to run the test suite. First run the few tests that are meant to be run as *root*:

```
export NON_ROOT_USERNAME=dummy; make check-root
```

Then run the remainder of the tests as the *dummy* user:

```
su dummy -c "make RUN_EXPENSIVE_TESTS=yes check"
```

When you're done testing, remove the dummy user and groups:

```
sed -i.bak '/dummy/d' /etc/passwd /etc/group
```

Install the package:

```
make install
```

And move some programs to their proper locations:

```
mv /usr/bin/{basename,cat,chgrp,chmod,chown,cp,dd,df} /bin
mv /usr/bin/{date,echo,false,head,install,ln,ls} /bin
mv /usr/bin/{mkdir,mknod,mv,pwd,rm,rmdir,sync} /bin
mv /usr/bin/{sleep,stty,su,test,touch,true,uname} /bin
mv /usr/bin/chroot /usr/sbin
```

We'll be using the `kill` program from the Procps package (installed as `/bin/kill` later in the chapter). Remove the one installed by Coreutils:

```
rm /usr/bin/kill
```

Finally, create two symlinks to be FHS-compliant:

```
ln -s test /bin/[
ln -s ../../bin/install /usr/bin
```

Contents of Coreutils

Installed programs: basename, cat, chgrp, chmod, chown, chroot, cksum, comm, cp, csplit, cut, date, dd, df, dir, dircolors, dirname, du, echo, env, expand, expr, factor, false, fmt, fold, groups, head, hostid, hostname, id, install, join, link, ln, logname, ls, md5sum, mkdir, mkfifo, mknod, mv, nice, nl, nohup, od, paste, pathchk, pinky, pr, printenv, printf, ptx, pwd, readlink, rm, rmdir, seq, sha1sum, shred, sleep, sort, split, stat, stty, su, sum, sync, tac, tail, tee, test, touch, tr, true, tsort, tty, uname, unexpand, uniq, unlink, uptime, users, vdir, wc, who, whoami and yes

Short descriptions

basename strips any path and a given suffix from the given file name.

cat concatenates files to standard output.

chgrp changes the group ownership of each given file to the given group. The group can be either given a a name or a numeric ID.

chmod changes the permissions of each given file to the given mode. The mode can be either a symbolic representation of the changes to make, or an octal number representing the new permissions.

chown changes the user and/or group ownership of each given file to the given user:group pair.

chroot runs a given command with the specified directory as the `/` directory. The given command can be an interactive shell. On most systems only *root* can do this.

cksum prints the CRC (Cyclic Redundancy Check) checksum and the byte counts of each specified file.

comm compares two sorted files, outputting in three columns the lines that are unique, and the lines that are common.

cp copies files.

csplit splits a given file into several new files, separating them according to given patterns or line numbers, and outputting the byte count of each new file.

cut prints parts of lines, selecting the parts according to given fields or positions.

date displays the current time in the given format, or sets the system date.

dd copies a file using the given block size and count, while optionally performing conversions on it.

df reports the amount of disk space available (and used) on all mounted file systems, or only on the file systems holding the given files.

dir is the same as ls.

dircolors outputs commands to set the LS_COLOR environment variable, to change the color scheme used by ls.

dirname strips the non-directory suffix from a given file name.

du reports the amount of disk space used by the current directory, or by each of the given directories including all their subdirectories, or by each of the given files.

echo displays the given strings.

env runs a command in a modified environment.

expand converts tabs to spaces.

expr evaluates expressions.

factor prints the prime factors of all specified integer numbers.

false does nothing, unsuccessfully. It always exits with a status code indicating failure.

fmt reformats the paragraphs in the given files.

fold wraps the lines in the given files.

groups reports a user's group memberships.

head prints the first ten lines (or the given number of lines) of each given file.

hostid reports the numeric identifier (in hexadecimal) of the host.

hostname reports or sets the name of the host.

id reports the effective user ID, group ID, and group memberships of the current user, or of a given user.

install copies files while setting their permission modes and, if possible, their owner and group.

join joins from two files the lines that have identical join fields.

link creates a hard link with the given name to the given file.

ln makes hard links or soft links between files.

logname reports the current user's login name.

ls lists the contents of each given directory. By default it orders the files and subdirectories alphabetically.

md5sum reports or checks MD5 (Message Digest 5) checksums.

mkdir creates directories with the given names.

mkfifo creates FIFOs (First-In, First-Out, a "named pipe" in UNIX parlance) with the given names.

mknod creates device nodes with the given names. A device node is a character special file, or a block special file, or a FIFO.

mv moves or renames files or directories.

nice runs a program with modified scheduling priority.

nl numbers the lines from the given files.

nohup runs a command immune to hangups, with output redirected to a log file.

od dumps files in octal and other formats.

paste merges the given files, joining sequentially corresponding lines side by side, separated by tab characters..

pathchk checks whether file names are valid or portable.

pinky is a lightweight finger. It reports some information about the given users.

pr paginates and columnates files for printing.

printenv prints the environment.

printf prints the given arguments according to the given format -- much like the C printf function.

ptx produces from the contents of the given files a permuted index, with each keyword in its context.

pwd reports the name of the current directory.

readlink reports the value of the given symbolic link.

rm removes files or directories.

rmdir removes directories, if they are empty.

seq prints a sequence of numbers, within a given range and with a given increment.

sha1sum prints or checks 160-bit SHA1 checksums.

shred overwrites the given files repeatedly with strange patterns, to make it real hard to recover the data.

sleep pauses for the given amount of time.

sort sorts the lines from the given files.

split splits the given file into pieces, by size or by numbspliter of lines.

stty sets or reports terminal line settings.

su runs a shell with substitute user and group IDs.

sum prints checksum and block counts for each given file.

sync flushes file system buffers. It forces changed blocks to disk and updates the super block.

tac concatenates the given files in reverse.

tail prints the last ten lines (or the given number of lines) of each given file.

tee reads from standard input while writing both to standard output and to the given files.

test compares values and checks file types.

touch changes file timestamps, setting the access and modification times of the given files to the current time. Files that do not exist are created with zero length.

tr translates, squeezes, and deletes the given characters from standard input.

true does nothing, successfully. It always exits with a status code indicating success.

tsort performs a topological sort. It writes a totally ordered list according to the partial ordering in a given file.

tty reports the file name of the terminal connected to standard input.

uname reports system information.

unexpand converts spaces to tabs.

uniq discards all but one of successive identical lines.

unlink removes the given file.

uptime reports how long the system has been running, how many users are logged on, and the system load averages.

users reports the names of the users currently logged on.

vdir is the same as ls -l.

wc reports the number of lines, words, and bytes for each given file, and a total line when more than one file is given.

who reports who is logged on.

whoami reports the user name associated with the current effective user ID.

yes outputs 'y' or a given string repeatedly, until killed.

Zlib-1.2.1

The Zlib package contains compression and un-compression routines used by some programs.

```
Approximate build time:  0.1 SBU
Required disk space:     1.5 MB
```

Zlib installation depends on: Binutils, Coreutils, GCC, Glibc, Make, Sed.

Installation of Zlib

Note

Zlib is known to build its shared library incorrectly if CFLAGS is specified in the environment. If you are using your own CFLAGS variable, be sure to add the *-fPIC* directive to your CFLAGS for the duration of the below **configure** command, then remove it afterwards.

Prepare Zlib for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**.

Install the shared library:

```
make install
```

Now also build the non-shared (static) library:

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

To again test the results, issue: **make check**.

Install the static library:

```
make install
```

And fix the permissions on the static library:

```
chmod 644 /usr/lib/libz.a
```

It is good policy and common practice to place important libraries into the `/lib` directory. This matters most in scenarios where `/usr` is on a separate partition. Essentially, the run-time components of any libraries that are used by programs in `/bin` or `/sbin` should reside in `/lib` so that they are on the root partition and available in the event of `/usr` being inaccessible.

For the above reason we move the run-time components of the shared Zlib into `/lib`:

```
mv /usr/lib/libz.so.* /lib
```

Now we need to fix the `/usr/lib/libz.so` symlink because we just moved the file it points to:

```
ln -sf ../../lib/libz.so.1 /usr/lib/libz.so
```

Contents of Zlib

Installed libraries: libz[a,so]

Short description

libz* contains compression and un-compression functions used by some programs.

Mktemp-1.5

The Mktemp package contains programs used to create secure temporary files in shell scripts.

```
Approximate build time:  0.1 SBU
Required disk space:     317 KB
```

The installation dependencies for Mktemp haven't been checked yet.

Installation of Mktemp

Many scripts still use the deprecated **tempfile** program, which has functionality much the same as **mktemp**. Patch mktemp to include a **tempfile** wrapper:

```
patch -Np1 -i ../mktemp-1.5-add-tempfile.patch
```

Now prepare Mktemp for compilation:

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

The meaning of the configure option:

- **`--with-libc`**: This causes the **mktemp** program to use the *mkstemp* and *mkdtemp* functions from the system C library.

Compile the package:

```
make
```

Now install it:

```
make install
make install-tempfile
```

Contents of Mktemp

Installed programs: mktemp, tempfile

Short descriptions

mktemp creates temporary files in a secure manner. It is used in scripts.

tempfile creates temporary files in a less secure manner than **mktemp**. It is installed for backwards-compatibility.

Iana-Etc-1.00

The Iana-Etc package provides data for network services and protocols.

```
Approximate build time:  0.1 SBU
Required disk space:     641 KB
```

The installation dependencies for Iana-Etc haven't been checked yet.

Installation of Iana-Etc

Parse the data:

```
make
```

Now install it:

```
make install
```

Contents of Iana-Etc

Installed files: protocols, services

Findutils-4.1.20

The Findutils package contains programs to find files. Processes are provided to recursively search through a directory tree and to create, maintain and search a database (often faster than the recursive find, but unreliable if the database has not been recently updated).

```
Approximate build time:  0.2 SBU
Required disk space:     7.5 MB
```

Findutils installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Findutils

Prepare Findutils for compilation:

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

The localstatedir directive above changes the location of the locate database to be in /var/lib/misc, which is FHS-compliant.

Compile the package:

```
make
```

To test the results, issue: **`make check`**.

Now install the package:

```
make install
```

Contents of Findutils

Installed programs: bigram, code, find, frcode, locate, updatedb and xargs

Short descriptions

bigram was formerly used to produce locate databases.

code was formerly used to produce locate databases. It is the ancestor of frcode.

find searches given directory trees for files matching the specified criteria.

frcode is called by updatedb to compress the list of file names. It uses front-compression, reducing the database size by a factor of 4 to 5.

locate searches through a database of file names, and reports the names that contain a given string or match a given pattern.

updatedb updates the locate database. It scans the entire file system (including other file systems that are currently mounted, unless told not to) and puts every file name it finds in the database.

xargs can be used to apply a given command to a list of files.

Gawk-3.1.3

The Gawk package contains programs for manipulating text files.

```
Approximate build time:  0.2 SBU
Required disk space:     17 MB
```

Gawk installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Gawk

Prepare Gawk for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **`make check`**.

Now install the package:

```
make install
```

Contents of Gawk

Installed programs: awk (link to gawk), gawk, gawk-3.1.3, grcat, igawk, pgawk, pgawk-3.1.3 and pwcat

Short descriptions

gawk is a program for manipulating text files. It is the GNU implementation of awk.

grcat dumps the group database `/etc/group`.

igawk gives gawk the ability to include files.

pgawk is the profiling version of gawk.

pwcat dumps the password database `/etc/passwd`.

Ncurses-5.4

The Ncurses package contains libraries for terminal-independent handling of character screens.

```
Approximate build time:  0.6 SBU
Required disk space:     27 MB
```

Ncurses installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Ncurses

Prepare Ncurses for compilation:

```
./configure --prefix=/usr --with-shared --without-debug
```

Compile the package:

```
make
```

Install the package:

```
make install
```

Give the Ncurses libraries execute permissions:

```
chmod 755 /usr/lib/*.5.4
```

Now fix a library that shouldn't be executable:

```
chmod 644 /usr/lib/libncurses++.a
```

Move the libraries to the `/lib` directory, where they're expected to reside:

```
mv /usr/lib/libncurses.so.5* /lib
```

Because the libraries have been moved, a few symlinks are pointing to non-existent files. Recreate those symlinks:

```
ln -sf ../../lib/libncurses.so.5 /usr/lib/libncurses.so
ln -sf libncurses.so /usr/lib/libcurses.so
```

Contents of Ncurses

Installed programs: captoinfo (link to tic), clear, infocmp, infotocap (link to tic), reset (link to tset), tack, tic, toe, tput and tset

Installed libraries: libcurses.[a,so] (link to libncurses.[a,so]), libform.[a,so], libmenu.[a,so], libncurses++.a, libncurses.[a,so], libpanel.[a,so]

Short descriptions

captoinfo converts a termcap description into a terminfo description.

clear clears the screen, if this is possible.

infocmp compares or prints out terminfo descriptions.

infotocap converts a terminfo description into a termcap description.

reset reinitializes a terminal to its default values.

tack is the terminfo action checker. It is mainly used to test the correctness of an entry in the terminfo database.

tic is the terminfo entry-description compiler. It translates a terminfo file from source format into the binary format

needed for the ncurses library routines. A terminfo file contains information on the capabilities of a certain terminal.

toe lists all available terminal types, for each giving its primary name and its description.

tput makes the values of terminal-dependent capabilities available to the shell. It can also be used to reset or initialize a terminal, or report its long name.

tset can be used to initialize terminals.

libncurses* contains functions to display text in many complicated ways on a terminal screen. A good example of the use of these functions is the menu displayed during the kernel's make menuconfig.

libform* contains functions to implement forms.

libmenu* contains functions to implement menus.

libpanel* contains functions to implement panels.

Vim-6.2

The Vim package contains a powerful text editor.

```
Approximate build time:  0.4 SBU
Required disk space:     34 MB
```

Vim installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Ncurses, Sed.

Alternatives to Vim

If you prefer another editor -- like Emacs, Joe, or Nano -- to Vim, have a look at http://www.linuxfromscratch.org/blfs/view/stable/postlfs/editors.html for suggested installation instructions.

Installation of Vim

First change the default locations of the `vimrc` and `gvimrc` configuration files to `/etc`.

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

Now prepare Vim for compilation:

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

Compile the package:

```
make
```

To test the results, you can issue: **`make test`**. However, this test suite outputs a lot of seemingly garbage characters to the screen, and this can wreak havoc with the settings of the current terminal. Therefore the running of the test suite here is strictly optional.

Now install the package:

```
make install
```

Many users are used to using **vi**, instead of **vim**. To let them execute **vim** when they habitually enter **vi**, create a symlink:

```
ln -s vim /usr/bin/vi
```

If you are going to install the X Window system on your LFS system, you may want to re-compile Vim after having installed X. Vim comes with a nice GUI version of the editor that requires X and a few other libraries to be installed. For more information read the Vim documentation.

Configuring Vim

By default, **vim** runs in vi-incompatible mode. Some people might not like this, but we prefer to run **vim** in its own mode (else we wouldn't have included it in this book, but the original **vi**). We've included the setting of "nocompatible" below to high-light the fact that the new behavior is being used. It also reminds those who would change to "compatible" mode that it should appear first because it changes other settings and overrides must come after this setting. Create a default vim configuration file by running the following:

```
cat > /etc/vimrc << "EOF"
" Begin /etc/vimrc

set nocompatible
set backspace=2
syntax on

" End /etc/vimrc
```

```
EOF
```

The *set nocompatible* makes **vim** behave in a more useful way (the default) than the vi-compatible manner. Remove the "no" if you want the old **vi** behavior. The *set backspace=2* allows backspacing over line breaks, autoindents and the start of insert. The *syntax on* enables **vim**'s semantic coloring.

Contents of Vim

Installed programs: efm_filter.pl, efm_perl.pl, ex (link to vim), less.sh, mve.awk, pltags.pl, ref, rview (link to vim), rvim (link to vim), shtags.pl, tcltags, vi (link to vim), view (link to vim), vim, vim132, vim2html.pl, vimdiff (link to vim), vimm, vimspell.sh, vimtutor and xxd

Short descriptions

efm_filter.pl is a filter for creating an error file that can be read by vim.

efm_perl.pl reformats the error messages of the Perl interpreter for use with the “quickfix” mode of vim.

ex starts vim in ex mode.

less.sh is a script that starts vim with less.vim.

mve.awk processes vim errors.

pltags.pl creates a tags file for perl code, for use by vim.

ref checks the spelling of arguments.

rview is a restricted version of view: no shell commands can be started and view can't be suspended.

rvim is a restricted version of vim: no shell commands can be started and vim can't be suspended.

shtags.pl generates a tag file for perl scripts.

tcltags generates a tag file for TCL code.

view starts vim in read-only mode.

vim is the editor.

vim132 starts vim with the terminal in 132-column mode.

vim2html.pl converts vim documentation to HTML.

vimdiff edits two or three versions of a file with vim and show differences.

vimm enables the DEC locator input model on a remote terminal.

vimspell.sh is a script which spells a file and generates the syntax statements necessary to highlight in vim.

vimtutor teaches you the basic keys and commands of vim.

xxd makes a hex dump of the given file. It can also do the reverse, so it can be used for binary patching.

M4-1.4

The M4 package contains a macro processor.

```
Approximate build time:  0.1 SBU
Required disk space:     3.0 MB
```

M4 installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed.

Installation of M4

Prepare M4 for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**.

And install the package:

```
make install
```

Contents of M4

Installed program: m4

Short description

m4 copies the given files while expanding the macros that they contain. These macros are either built-in or user-defined and can take any number of arguments. Besides just doing macro expansion, m4 has built-in functions for including named files, running Unix commands, doing integer arithmetic, manipulating text in various ways, recursion, and so on. The m4 program can be used either as a front-end to a compiler or as a macro processor in its own right.

Bison-1.875

The Bison package contains a parser generator.

```
Approximate build time:  0.6 SBU
Required disk space:     10.6 MB
```

Bison installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, M4, Make, Sed.

Installation of Bison

First fix a minor compilation problem that Bison has with some packages, the patch is back-ported from CVS:

```
patch -Np1 -i ../bison-1.875-attribute.patch
```

Now prepare Bison for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **`make check`**.

Now install the package:

```
make install
```

Contents of Bison

Installed programs: bison and yacc

Installed library: liby.a

Short descriptions

bison generates, from a series of rules, a program for analyzing the structure of text files. Bison is a replacement for yacc (Yet Another Compiler Compiler).

yacc is a wrapper for bison, meant for programs that still call yacc instead of bison. It calls bison with the -y option.

liby.a is the Yacc library containing implementations of Yacc-compatible yyerror and main functions. This library is normally not very useful, but POSIX requires it.

Less-382

The Less package contains a text file viewer.

```
Approximate build time:  0.1 SBU
Required disk space:     3.4 MB
```

Less installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Ncurses, Sed.

Installation of Less

Prepare Less for compilation:

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

The meaning of the configure option:

- **`--sysconfdir=/etc`**: This option tells the programs created by the package to look in `/etc` for their configuration files.

Compile the package:

```
make
```

Now install it:

```
make install
```

Contents of Less

Installed programs: less, lessecho and lesskey

Short descriptions

less is a file viewer or pager. It displays the contents of the given file, letting you scroll around, find strings, and jump to marks.

lessecho is needed to expand meta-characters, such as * and ?, in filenames on Unix systems.

lesskey is used to specify the key bindings for less.

Groff-1.19

The Groff package contains programs for processing and formatting text.

```
Approximate build time:  0.5 SBU
Required disk space:     43 MB
```

Groff installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Groff

Groff expects the environment variable PAGE to contain the default paper size. For those in the United States, the command below is appropriate. If you live elsewhere, you may want to change *PAGE=letter* to *PAGE=A4*.

Prepare Groff for compilation:

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

Compile the package:

```
make
```

Now install it:

```
make install
```

Some documentation programs, such as **xman**, will not work properly without the following symlinks:

```
ln -s soelim /usr/bin/zsoelim
ln -s eqn /usr/bin/geqn
ln -s tbl /usr/bin/gtbl
```

Contents of Groff

Installed programs: addftinfo, afmtodit, eqn, eqn2graph, geqn (link to eqn), grn, grodvi, groff, groffer, grog, grolbp, grolj4, grops, grotty, gtbl (link to tbl), hpftodit, indxbib, lkbib, lookbib, mmroff, neqn, nroff, pfbtops, pic, pic2graph, post-grohtml, pre-grohtml, refer, soelim, tbl, tfmtodit, troff and zsoelim (link to soelim)

Short descriptions

addftinfo reads a troff font file and adds some additional font-metric information that is used by the groff system.

afmtodit creates a font file for use with groff and grops.

eqn compiles descriptions of equations embedded within troff input files into commands that are understood by troff.

eqn2graph converts an EQN equation into a cropped image.

grn is a groff preprocessor for gremlin files.

grodvi is a driver for groff that produces TeX dvi format.

groff is a front-end to the groff document formatting system. Normally it runs the troff program and a post-processor appropriate for the selected device.

groffer displays groff files and man pages on X and tty terminals.

grog reads files and guesses which of the groff options -e, -man, -me, -mm, -ms, -p, -s, and -t are required for printing files, and reports the groff command including those options.

grolbp is a groff driver for Canon CAPSL printers (LBP-4 and LBP-8 series laser printers).

grolj4 is a driver for groff that produces output in PCL5 format suitable for an HP Laserjet 4 printer.

grops translates the output of GNU troff to Postscript.

grotty translates the output of GNU troff into a form suitable for typewriter-like devices.

gtbl is the GNU implementation of tbl.

hpftodit creates a font file for use with groff -Tlj4 from an HP-tagged font metric file.

indxbib makes an inverted index for the bibliographic databases a specified file for use with refer, lookbib, and lkbib.

lkbib searches bibliographic databases for references that contain specified keys and reports any references found.

lookbib prints a prompt on the standard error (unless the standard input is not a terminal), reads from the standard input a line containing a set of keywords, searches the bibliographic databases in a specified file for references containing those keywords, prints any references found on the standard output and repeats this process until the end of input.

mmroff is a simple preprocessor for groff.

neqn formats equations for ASCII (American Standard Code for Information Interchange) output.

nroff is a script that emulates the nroff command using groff.

pfbtops translates a Postscript font in .pfb format to ASCII.

pic compiles descriptions of pictures embedded within troff or TeX input files into commands understood by TeX or troff.

pic2graph converts a PIC diagram into a cropped image.

pre-grohtml translates the output of GNU troff to html.

post-grohtml translates the output of GNU troff to html.

refer copies the contents of a file to the standard output, except that lines between .[and .] are interpreted as citations, and lines between .R1 and .R2 are interpreted as commands about how citations are to be processed.

soelim reads files and replaces lines of the form *.so file* by the contents of the mentioned *file*.

tbl compiles descriptions of tables embedded within troff input files into commands that are understood by troff.

tfmtodit creates a font file for use with groff -Tdvi.

troff is highly compatible with Unix troff. Usually it should be invoked using the groff command, which will also run preprocessors and post-processors in the appropriate order and with the appropriate options.

zsoelim is the GNU implementation of soelim.

Sed-4.0.9

The Sed package contains a stream editor.

```
Approximate build time:  0.2 SBU
Required disk space:     5.2 MB
```

Sed installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Texinfo.

Installation of Sed

Prepare Sed for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**.

Now install the package:

```
make install
```

Contents of Sed

Installed program: sed

Short description

sed is used to filter and transform text files in a single pass.

Flex-2.5.4a

The Flex package contains a utility for generating programs that recognize patterns in text.

```
Approximate build time:  0.1 SBU
Required disk space:     3.4 MB
```

Flex installation depends on: Bash, Binutils, Bison, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, M4, Make, Sed.

Installation of Flex

Prepare Flex for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make bigcheck**.

Now install the package:

```
make install
```

There are some packages that expect to find the *lex* library in `/usr/lib`. Create a symlink to account for this:

```
ln -s libfl.a /usr/lib/libl.a
```

A few programs don't know about **flex** yet and try to run its predecessor **lex**. To support those programs, create a wrapper script named `lex` that calls **flex** in *lex* emulation mode:

```
cat > /usr/bin/lex << "EOF"
#!/bin/sh
# Begin /usr/bin/lex

exec /usr/bin/flex -l "$@"

# End /usr/bin/lex
EOF
chmod 755 /usr/bin/lex
```

Contents of Flex

Installed programs: flex, flex++ (link to flex) and lex

Installed library: libfl.a

Short descriptions

flex is a tool for generating programs that recognize patterns in text. Pattern recognition is useful in many applications. From a set of rules on what to look for, flex makes a program that looks for those patterns. The reason to use flex is that it is much easier to specify the rules for a pattern-finding program than to write the program.

flex++ invokes a version of flex that is used exclusively for C++ scanners.

libfl.a is the flex library.

Gettext-0.14.1

The Gettext package contains utilities for internationalization and localization. These allow programs to be compiled with Native Language Support (NLS), enabling them to output messages in the user's native language.

```
Approximate build time:  0.5 SBU
Required disk space:     55 MB
```

Gettext installation depends on: Bash, Binutils, Bison, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Gettext

Prepare Gettext for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **`make check`**. This takes a very long time, around 7 SBUs.

Now install the package:

```
make install
```

Contents of Gettext

Installed programs: autopoint, config.charset, config.rpath, envsubst, gettext, gettextize, hostname, msgattrib, msgcat, msgcmp, msgcomm, msgconv, msgen, msgexec, msgfilter, msgfmt, msggrep, msginit, msgmerge, msgunfmt, msguniq, ngettext, project-id, team-address, trigger, urlget, user-email and xgettext

Installed libraries: libasprintf[a,so], libgettextlib[a,so], libgettextpo[a,so] and libgettextsrc[a,so]

Short descriptions

autopoint copies standard gettext infrastructure files into a source package.

config.charset outputs a system-dependent table of character encoding aliases.

config.rpath outputs a system-dependent set of variables, describing how to set the runtime search path of shared libraries in an executable.

envsubst substitutes environment variables in shell format strings.

gettext translates a natural language message into the user's language, by looking up the translation in a message catalog.

gettextize copies all standard Gettext files into the given top-level directory of a package, to begin inter-nationalizing it.

hostname displays a network hostname in various forms.

msgattrib filters the messages of a translation catalog according to their attributes and manipulates the attributes.

msgcat concatenates and merges the given `.po` files.

msgcmp compares two `.po` files to check that both contain the same set of msgid strings.

msgcomm finds the messages that are common to to the given `.po` files.

msgconv converts a translation catalog to a different character encoding.

msgen creates an English translation catalog.

msgexec applies a command to all translations of a translation catalog.

msgfilter applies a filter to all translations of a translation catalog.

msgfmt generates a binary message catalog from from a translation catalog.

msggrep extracts all messages of a translation catalog that match a given pattern or belong to some given source files.

msginit creates a new `.po` file, initializing the meta information with values from the user's environment.

msgmerge combines two raw translations into a single file.

msgunfmt decompiles a binary message catalog into raw translation text.

msguniq unifies duplicate translations in a translation catalog.

ngettext displays native language translations of a textual message whose grammatical form depends on a number.

xgettext extracts the translatable message lines from the given source files, to make the first translation template.

libasprintf defines the autosprintf class, which makes C formatted output routines usable in C++ programs, for use with the <string> strings and the <iostream> streams.

libgettextlib is a private library containing common routines used by the various gettext programs. They're not meant for general use.

libgettextpo is used to write specialized programs that process PO files. This library is used when the standard applications shipped with gettext won't suffice (such as msgcomm, msgcmp, msgattrib and msgen).

libgettextsrc is a private library containing common routines used by the various gettext programs. They're not meant for general use.

Net-tools-1.60

The Net-tools package contains programs for basic networking.

```
Approximate build time:  0.1 SBU
Required disk space:     9.4 MB
```

Net-tools installation depends on: Bash, Binutils, Coreutils, GCC, Glibc, Make.

Installation of Net-tools

If you don't know what to answer to all the questions asked during the **make config** phase below, then just accept the defaults. This will be just fine in the majority of cases. What you're asked here is a bunch of questions about which network protocols you've enabled in your kernel. The default answers will enable the tools from this package to work with the most common protocols: TCP, PPP, and several others. You still need to actually enable these protocols in the kernel -- what you do here is merely telling the package to include support for those protocols in its programs, but it's up to the kernel to make the protocols available.

First fix a small syntax problem in the sources of the **mii-tool** program:

```
patch -Np1 -i ../net-tools-1.60-miitool-gcc33-1.patch
```

Now prepare Net-tools for compilation (if you intend to accept the defaults, you can skip all the questions by running **`yes "" | make config`** instead):

```
make config
```

Compile the package:

```
make
```

Now install it:

```
make update
```

Contents of Net-tools

Installed programs: arp, dnsdomainname (link to hostname), domainname (link to hostname), hostname, ifconfig, nameif, netstat, nisdomainname (link to hostname), plipconfig, rarp, route, slattach and ypdomainname (link to hostname)

Short descriptions

arp is used to manipulate the kernel's ARP cache, usually to add or delete an entry, or to dump the entire cache.

dnsdomainname reports the system's DNS (Domain Name Server) domain name.

domainname reports or sets the system's NIS/YP domain name.

hostname reports or sets the name of the current host system.

ifconfig is the main utility for configuring network interfaces.

nameif names network interfaces based on MAC addresses.

netstat is used to report network connections, routing tables, and interface statistics..

nisdomainname does the same as domainname.

plipconfig is used to fine tune the PLIP device parameters, to improve its performance.

rarp is used to manipulate the kernel's RARP table.

route is used to manipulate the IP routing table.

slattach attaches a network interface to a serial line. This allows you to use normal terminal lines for point-to-point links to other computers.

ypdomainname does the same as domainname.

Inetutils-1.4.2

The Inetutils package contains programs for basic networking.

```
Approximate build time:  0.2 SBU
Required disk space:     11 MB
```

Inetutils installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Ncurses, Sed.

Installation of Inetutils

We are not going to install all the programs that come with Inetutils. However, the Inetutils build system will insist on installing all the man pages anyway. The following patch will correct this situation:

```
patch -Np1 -i ../inetutils-1.4.2-no_server_man_pages-1.patch
```

Now prepare Inetutils for compilation:

```
./configure --prefix=/usr --libexecdir=/usr/sbin \
    --sysconfdir=/etc --localstatedir=/var \
    --disable-logger --disable-syslogd \
    --disable-whois --disable-servers
```

The meaning of the configure options:

- **`--disable-logger`**: This option prevents Inetutils from installing the logger program, which is used by scripts to pass messages to the System Log Daemon. We do not install it because Util-linux installs a better version later.
- **`--disable-syslogd`**: This option prevents Inetutils from installing the System Log Daemon, which is installed with the Sysklogd package.
- **`--disable-whois`**: This option disables the building of the Inetutils whois client, which is woefully out of date. Instructions for a better whois client are in the BLFS book.
- **`--disable-servers`**: This disables the installation of the various network servers included as part of the Inetutils package. These servers are deemed not appropriate in a basic LFS system. Some are insecure by nature and are only considered safe on trusted networks. More information can be found at http://www.linuxfromscratch.org/blfs/view/stable/basicnet/inetutils.html. Note that better replacements are available for many of these servers.

Compile the package:

```
make
```

Install it:

```
make install
```

Move the **ping** program to its FHS-compliant place:

```
mv /usr/bin/ping /bin
```

Contents of Inetutils

Installed programs: ftp, ping, rcp, rlogin, rsh, talk, telnet and tftp

Short descriptions

ftp is the ARPANET file transfer program.

ping sends echo-request packets and reports how long the replies take.

rcp does remote file copy.

rlogin does remote login.

rsh runs a remote shell.

talk is used to chat up another user.

telnet is an interface to the TELNET protocol.

tftp is a trivial file transfer program.

Perl-5.8.4

The Perl package contains the Practical Extraction and Report Language.

```
Approximate build time:  2.9 SBU
Required disk space:     143 MB
```

Perl installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Perl

If you want full control over the way Perl is set up, you can run the interactive **Configure** script and hand-pick the way this package is built. If you think you can live with the (sensible) defaults it auto-detects, then prepare Perl for compilation with:

```
./configure.gnu --prefix=/usr -Dpager="/bin/less -isR"
```

The meaning of the configure option:

- **`-Dpager="/bin/less -isR"`**: This corrects an error in the perldoc code with the invocation of the less program.

Compile the package:

```
make
```

If you wish to run the test suite, you first have to create a basic `/etc/hosts` file, which is needed by a couple of tests to resolve the name *localhost*:

```
echo "127.0.0.1 localhost $(hostname)" > /etc/hosts
```

Now run the tests, if you wish:

```
make test
```

Finally, install the package:

```
make install
```

Contents of Perl

Installed programs: a2p, c2ph, dprofpp, enc2xs, find2perl, h2ph, h2xs, libnetcfg, perl, perl5.8.4 (link to perl), perlbug, perlcc, perldoc, perlivp, piconv, pl2pm, pod2html, pod2latex, pod2man, pod2text, pod2usage, podchecker, podselect, psed (link to s2p), pstruct (link to c2ph), s2p, splain and xsubpp

Installed libraries: (too many to name)

Short descriptions

a2p translates awk to perl.

c2ph dumps C structures as generated from "cc -g -S" stabs.

dprofpp displays perl profile data.

en2cxs builds a Perl extension for the Encode module, from either Unicode Character Mappings or Tcl Encoding Files.

find2perl translates find commands to perl.

h2ph converts .h C header files to .ph Perl header files.

h2xs converts .h C header files to Perl extensions.

libnetcfg can be used to configure the libnet.

perl combines some of the best features of C, sed, awk and sh into a single swiss-army language.

perlbug is used to generate bug reports about Perl or the modules that come with it, and mail them.

perlcc generates executables from Perl programs.

perldoc displays a piece of documentation in pod format that is embedded in the perl installation tree or in a perl script.

perlivp is the Perl Installation Verification Procedure. It can be used to verify that Perl and its libraries have been installed correctly.

piconv is a Perl version of the character encoding converter **iconv**.

pl2pm is a rough tool for converting Perl4 .pl files to Perl5 .pm modules.

pod2html converts files from pod format to HTML format.

pod2latex converts files from pod format to LaTeX format.

pod2man converts pod data to formatted *roff input.

pod2text converts pod data to formatted ASCII text.

pod2usage prints usage messages from embedded pod docs in files.

podchecker checks the syntax of pod format documentation files.

podselect displays selected sections of pod documentation.

psed is a Perl version of the stream editor **sed**.

pstruct dumps C structures as generated from "cc -g -S" stabs.

s2p translates sed to perl.

splain is used to force verbose warning diagnostics in perl.

xsubpp converts Perl XS code into C code.

Texinfo-4.7

The Texinfo package contains programs for reading, writing, and converting Info documents.

```
Approximate build time:  0.2 SBU
Required disk space:     17 MB
```

Texinfo installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Ncurses, Sed.

Installation of Texinfo

Prepare Texinfo for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**.

Install the package:

```
make install
```

Optionally install the components belonging in a TeX installation:

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

The meaning of the make parameter:

- **TEXMF=/usr/share/texmf**: The TEXMF makefile variable holds the location of the root of your TeX tree if, for example, you plan to install a TeX package later on.

The Info documentation system uses a plain text file to hold its list of menu entries. The file is located at `/usr/share/info/dir`. Unfortunately, due to occasional problems in the Makefiles of various packages, it can sometimes get out of step with the Info manuals actually installed on the system. If ever you need to recreate the `/usr/share/info/dir` file, the following optional commands will accomplish the task:

```
cd /usr/share/info
rm dir
for f in *
do install-info $f dir 2>/dev/null
done
```

Contents of Texinfo

Installed programs: info, infokey, install-info, makeinfo, texi2dvi and texindex

Short descriptions

info is used to read Info documents. Info documents are a bit like man pages, but often go much deeper than just explaining all the flags. Compare for example man tar and info tar.

infokey compiles a source file containing Info customizations into a binary format.

install-info is used to install Info files. It updates entries in the Info index file.

makeinfo translates the given Texinfo source documents into various other formats: Info files, plain text, or HTML.

texi2dvi is used to format the given Texinfo document into a device-independent file that can be printed.

texindex is used to sort Texinfo index files.

Autoconf-2.59

The Autoconf package contains programs for producing shell scripts that can automatically configure source code.

```
Approximate build time:  0.5 SBU
Required disk space:     7.7 MB
```

Autoconf installation depends on: Bash, Coreutils, Diffutils, Grep, M4, Make, Perl, Sed.

Installation of Autoconf

Prepare Autoconf for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**. This takes a long time, about 2 SBUs.

Install the package:

```
make install
```

Contents of Autoconf

Installed programs: autoconf, autoheader, autom4te, autoreconf, autoscan, autoupdate and ifnames

Short descriptions

autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of Unix-like systems. The configuration scripts it produces are independent -- running them does not require the autoconf program.

autoheader is a tool for creating template files of C #define statements for configure to use.

autom4te is a wrapper for the M4 macro processor.

autoreconf comes in handy when there are a lot of autoconf-generated configure scripts around. The program runs autoconf and autoheader repeatedly (where appropriate) to remake the autoconf configure scripts and configuration header templates in a given directory tree.

autoscan can help to create a `configure.in` file for a software package. It examines the source files in a directory tree, searching them for common portability problems and creates a `configure.scan` file that serves as as a preliminary `configure.in` for the package.

autoupdate modifies a `configure.in` file that still calls autoconf macros by their old names to use the current macro names.

ifnames can be helpful when writing a `configure.in` for a software package. It prints the identifiers that the package uses in C preprocessor conditionals. If a package has already been set up to have some portability, this program can help to determine what **configure** needs to check. It can fill in some gaps in a `configure.in` file generated by autoscan.

Automake-1.8.4

The Automake package contains programs for generating Makefiles for use with Autoconf.

```
Approximate build time:  0.2 SBU
Required disk space:     6.8 MB
```

Automake installation depends on: Autoconf, Bash, Coreutils, Diffutils, Grep, M4, Make, Perl, Sed.

Installation of Automake

Prepare Automake for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **`make check`**. This takes a long time, about 5 SBUs.

Install the package:

```
make install
```

Contents of Automake

Installed programs: acinstall, aclocal, aclocal-1.8, automake, automake-1.8, compile, config.guess, config.sub, depcomp, elisp-comp, install-sh, mdate-sh, missing, mkinstalldirs, py-compile, symlink-tree, ylwrap

Short descriptions

acinstall is a script that installs aclocal-style M4 files.

aclocal generates `aclocal.m4` files based on the contents of `configure.in` files.

automake is a tool for automatically generating `Makefile.in`'s from files called `Makefile.am`. To create all the `Makefile.in` files for a package, run this program in the top-level directory. By scanning the `configure.in`s it automatically finds each appropriate `Makefile.am` and generate the corresponding `Makefile.in`.

compile is a wrapper for compilers.

config.guess is a script that attempts to guess the canonical triplet for the given build, host, or target architecture.

config.sub is a configuration validation subroutine script.

depcomp is a script for compiling a program so that not only the desired output is generated, but also dependency information.

elisp-comp byte-compiles Emacs Lisp code.

install-sh is a script that installs a program, a script, or a datafile.

mdate-sh is a script that prints the modification time of a file or directory.

missing is a script acting as a common stub for missing GNU programs during an installation.

mkinstalldirs is a script that creates a directory tree.

py-compile compiles a Python program.

symlink-tree is a script to create a symlink tree of a directory tree.

ylwrap is a wrapper for lex and yacc.

Bash-2.05b

The Bash package contains the Bourne-Again SHell.

```
Approximate build time:  1.2 SBU
Required disk space:     27 MB
```

Bash installation depends on: Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Ncurses, Sed.

Installation of Bash

Bash has a number of bugs in it that cause it to not behave the way it is expected at times. Fix this behavior with the following patch:

```
patch -Np1 -i ../bash-2.05b-2.patch
```

Now prepare Bash for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **`make tests`**.

Install the package:

```
make install
```

Now run the newly compiled **bash** program (replacing the one you are currently executing):

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

Note that the parameters used make it an interactive login instance (so /etc/profile is read, if it exists, and the first found ~/.bash_profile, ~/.bash_login or and ~/.profile) and continue to disable hashing so that new programs are found as they become available.

Contents of Bash

Installed programs: bash, sh (link to bash) and bashbug

Short descriptions

bash is a widely-used command interpreter. It performs many kinds of expansions and substitutions on a given command line before executing it, thus making this interpreter a powerful tool.

bashbug is a shell script to help the user compose and mail bug reports concerning bash in a standard format.

sh is a symlink to the bash program. When invoked as sh, bash tries to mimic the startup behavior of historical versions of sh as closely as possible, while conforming to the POSIX standard as well.

File-4.09

The File package contains a utility for determining the type of files.

```
Approximate build time:  0.1 SBU
Required disk space:     6.3 MB
```

File installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed, Zlib.

Installation of File

Prepare File for compilation:

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

Compile the package:

```
make
```

Now install it:

```
make install
```

Contents of File

Installed program: file

Installed library: libmagic.[a,so]

Short descriptions

file tries to classify each given file. It does this by performing several tests: file system tests, magic number tests, and language tests. The first test that succeeds determines the result.

libmagic contains routines for magic number recognition, used by the file program.

Libtool-1.5.6

The Libtool package contains the GNU generic library support script. It wraps the complexity of using shared libraries in a consistent, portable interface.

```
Approximate build time:  1.5 SBU
Required disk space:     20 MB
```

Libtool installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of Libtool

Prepare Libtool for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**.

Install the package:

```
make install
```

Contents of Libtool

Installed programs: libtool and libtoolize

Installed libraries: libltdl.[a,so].

Short descriptions

libtool provides generalized library-building support services.

libtoolize provides a standard way to add libtool support to a package.

libltdl hides the various difficulties of dlopening libraries.

Bzip2-1.0.2

The Bzip2 package contains programs for compressing and decompressing files. On text files they achieve a much better compression than the traditional **gzip**.

```
Approximate build time:  0.1 SBU
Required disk space:     3.0 MB
```

Bzip2 installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Make.

Installation of Bzip2

Prepare Bzip2 for compilation with:

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

The *-f* flag will cause Bzip2 to be built using a different `Makefile` file, in this case the `Makefile-libbz2_so` file, which creates a dynamic `libbz2.so` library and links the Bzip2 utilities against it.

Compile the package:

```
make
```

If you are reinstalling Bzip2, you need to do **`rm -f /usr/bin/bz*`** first, otherwise the following **make install** will fail.

Install the programs:

```
make install
```

Now install the shared **bzip2** binary into the `/bin` directory, then make some necessary symbolic links, and clean up:

```
cp bzip2-shared /bin/bzip2
cp -a libbz2.so* /lib
ln -s ../../lib/libbz2.so.1.0 /usr/lib/libbz2.so
rm /usr/bin/{bunzip2,bzcat,bzip2}
mv /usr/bin/{bzip2recover,bzless,bzmore} /bin
ln -s bzip2 /bin/bunzip2
ln -s bzip2 /bin/bzcat
```

Contents of Bzip2

Installed programs: bunzip2 (link to bzip2), bzcat (link to bzip2), bzcmp, bzdiff, bzegrep, bzfgrep, bzgrep, bzip2, bzip2recover, bzless and bzmore

Installed libraries: libbz2.a, libbz2.so (link to libbz2.so.1.0), libbz2.so.1.0 (link to libbz2.so.1.0.2) and libbz2.so.1.0.2

Short descriptions

bunzip2 decompresses bzipped files.

bzcat decompresses to standard output.

bzcmp runs cmp on bzipped files.

bzdiff runs diff on bzipped files.

bzgrep and friends run grep on bzipped files.

bzip2 compresses files using the Burrows-Wheeler block sorting text compression algorithm with Huffman coding. The compression rate is generally considerably better than that achieved by more conventional compressors using LZ77/LZ78, like **gzip**.

bzip2recover tries to recover data from damaged bzip2 files.

bzless runs less on bzipped files.

bzmore runs more on bzipped files.

libbz2* is the library implementing lossless, block-sorting data compression, using the Burrows-Wheeler algorithm.

Diffutils-2.8.1

The Diffutils package contains programs that show the differences between files or directories.

```
Approximate build time:  0.1 SBU
Required disk space:     7.5 MB
```

Diffutils installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Diffutils

Prepare Diffutils for compilation:

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

Compile the package:

```
make
```

Install it:

```
make install
```

Contents of Diffutils

Installed programs: cmp, diff, diff3 and sdiff

Short descriptions

cmp compares two files and reports whether or in which bytes they differ.

diff compares two files or directories and reports which lines in the files differ.

diff3 compares three files line by line.

sdiff merges two files and interactively outputs the results.

Ed-0.2

The Ed package contains a spartan line editor.

```
Approximate build time:  0.1 SBU
Required disk space:     3.1 MB
```

Ed installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of Ed

Note

Ed isn't something which many people use. It's installed here because it can be used by the patch program if you encounter an ed-based patch file. This happens rarely because diff-based patches are preferred these days.

Ed normally uses the *mktemp* function to create temporary files in `/tmp`, but this function contains a vulnerability (see the section on Temporary Files in http://en.tldp.org/HOWTO/Secure-Programs-HOWTO/avoid-race.html). Apply the following patch to make Ed use *mkstemp* instead, a secure way to create temporary files:

```
patch -Np1 -i ../ed-0.2-mkstemp.patch
```

Now prepare Ed for compilation:

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

The meaning of the configure option:

- **`--exec-prefix=""`**: This forces the programs to be installed into the `/bin` directory. Having the programs available there is useful in the event of the `/usr` partition being unavailable.

Compile the package:

```
make
```

To test the results, issue: **`make check`**.

Install the package:

```
make install
```

Contents of Ed

Installed programs: ed and red (link to ed)

Short descriptions

ed is a line-oriented text editor. It can be used to create, display, modify and otherwise manipulate text files.

red is a restricted ed -- it can only edit files in the current directory and cannot execute shell commands.

Kbd-1.12

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

```
Approximate build time:  0.1 SBU
Required disk space:     12 MB
```

Kbd installation depends on: Bash, Binutils, Bison, Coreutils, Diffutils, Flex, GCC, Gettext, Glibc, Grep, Gzip, M4, Make, Sed.

Installation of Kbd

By default some of Kbd's utilities (**setlogcons**, **setvesablank** and **getunimap**) are not installed. First enable the compilation of these utilities:

```
patch -Np1 -i ../kbd-1.12-more-programs-1.patch
```

Now prepare Kbd for compilation:

```
./configure
```

Compile the package:

```
make
```

Now install it:

```
make install
```

Configuring your keyboard

Few things are more annoying than using Linux while a wrong keymap for your keyboard is loaded. If you have a standard US keyboard, however, you can skip this section, as the US keymap is the default as long as you don't change it.

To change the default keymap, create the `/usr/share/kbd/keymaps/defkeymap.map.gz` symlink by running the following command:

```
ln -s path/to/keymap /usr/share/kbd/keymaps/defkeymap.map.gz
```

Of course, replace `path/to/keymap` with the path and name of your keyboard's map file. For example, if you have a Dutch keyboard, you would use `/usr/share/kbd/keymaps/i386/qwerty/nl.map.gz`.

Another way to set your keyboard's layout is to compile the keymap into the kernel. This ensures that your keyboard will always work as expected, even when you boot into maintenance mode (by passing `init=/bin/sh' to the kernel), as then the bootscript that normally sets up your keymap isn't run.

When in Chapter 8[p.168] you're ready to compile the kernel, run the following command to patch the current default keymap into the source (you will have to repeat this command whenever you unpack a new kernel):

```
loadkeys -m /usr/share/kbd/keymaps/defkeymap.map.gz > \
    [unpacked sources dir]/linux-2.4.26/drivers/char/defkeymap.c
```

Contents of Kbd

Installed programs: chvt, deallocvt, dumpkeys, fgconsole, getkeycodes, getunimap, kbd_mode, kbdrate, loadkeys, loadunimap, mapscrn, openvt, psfaddtable (link to psfxtable), psfgettable (link to psfxtable), psfstriptable (link to psfxtable), psfxtable, resizecons, setfont, setkeycodes, setleds, setlogcons, setmetamode, setvesablank, showconsolefont, showkey, unicode_start and unicode_stop

Short descriptions

chvt changes the foreground virtual terminal.

deallocvt deallocates unused virtual terminals.

dumpkeys dumps the keyboard translation tables.

fgconsole prints the number of the active virtual terminal.

getkeycodes prints the kernel scancode-to-keycode mapping table.

getunimap prints the currently used unimap.

kbd_mode reports or sets the keyboard mode.

kbdrate sets the keyboard repeat and delay rates.

loadkeys loads the keyboard translation tables.

loadunimap loads the kernel unicode-to-font mapping table.

mapscrn is an obsolete program that used to load a user-defined output character mapping table into the console driver. This is now done by setfont.

openvt starts a program on a new virtual terminal (VT).

psf* are a set of tools for handling Unicode character tables for console fonts.

resizecons changes the kernel idea of the console size.

setfont lets you change the EGA/VGA fonts on the console.

setkeycodes loads kernel scancode-to-keycode mapping table entries, useful if you have some unusual keys on your keyboard.

setleds sets the keyboard flags and LEDs. Many people find it useful to have "Num Lock" on by default, setleds +num achieves this.

setlogcons sends kernel messages to the console.

setmetamode defines the keyboard meta-key handling.

setvesablank lets you fiddle with the built-in hardware screensaver (no toasters, just a blank screen).

showconsolefont shows the current EGA/VGA console screen font.

showkey reports the scancodes and keycodes and ASCII codes of the keys pressed on the keyboard.

unicode_start puts the keyboard and console in unicode mode.

unicode_stop reverts keyboard and console from unicode mode.

E2fsprogs-1.35

The E2fsprogs package contains the utilities for handling the ext2 file system. It also supports the ext3 journaling file system.

```
Approximate build time:  0.6 SBU
Required disk space:     48.4 MB
```

E2fsprogs installation depends on: Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Gettext, Glibc, Grep, Make, Sed, Texinfo.

Installation of E2fsprogs

It is recommended to build E2fsprogs outside of the source tree:

```
mkdir ../e2fsprogs-build
cd ../e2fsprogs-build
```

Prepare E2fsprogs for compilation:

```
../e2fsprogs-1.35/configure --prefix=/usr --with-root-prefix="" \
    --enable-elf-shlibs
```

The meaning of the configure options:

- **`--with-root-prefix=""`**: Certain programs (such as the e2fsck program) are considered essential programs. When, for example, `/usr` isn't mounted, these essential programs have to be available. They belong in directories like `/lib` and `/sbin`. If this option isn't passed to E2fsprogs's configure, the programs are placed in the `/usr` directory, which is not what we want.
- **`--enable-elf-shlibs`**: This creates the shared libraries which some programs in this package use.

Compile the package:

```
make
```

If you to test the results, first make sure an mtab file exists with **`touch /etc/mtab`** to prevent some sixty tests from failing, and (if it doesn't already exist) fake the presence of an old pager with **`ln -s /tools/bin/cat /bin/more`** to prevent one test from failing, then issue: **`make check`**.

Install most of the package:

```
make install
```

Also install the shared libraries:

```
make install-libs
```

Contents of E2fsprogs

Installed programs: badblocks, blkid, chattr, compile_et, debugfs, dumpe2fs, e2fsck, e2image, e2label, findfs, fsck, fsck.ext2, fsck.ext3, logsave, lsattr, mk_cmds, mke2fs, mkfs.ext2, mkfs.ext3, mklost+found, resize2fs, tune2fs and uuidgen.

Installed libraries: libblkid.[a,so], libcom_err.[a,so], libe2p.[a,so], libext2fs.[a,so], libss.[a,so] and libuuid.[a,so]

Short descriptions

badblocks searches a device (usually a disk partition) for bad blocks.

blkid is a command line utility to locate and print block device attributes.

chattr changes the attributes of files on a second extended (ext2) file system, and also ext3 file systems, the journaling version of ext2 file systems.

compile_et is an error table compiler. It converts a table of error-code names and messages into a C source file suitable for use with the com_err library.

debugfs is a file system debugger. It can be used to examine and change the state of an ext2 file system.

dumpe2fs prints the super block and blocks group information for the file system present on a given device.

e2fsck is used to check, and optionally repair, second extended (ext2) file systems, and also ext3 file systems.

e2image is used to save critical ext2 file system data to a file.

e2label will display or change the file system label on the ext2 file system present on a given device.

findfs finds a file system by label or UUID (Universally Unique Identifier).

fsck is used to check, and optionally repair, file systems. By default it checks the file systems listed in `/etc/fstab`

logsave saves the output of a command in a log file.

lsattr lists the attributes of files on a second extended file system.

mk_cmds converts a table of command names and help messages into a C source file suitable for use with the `libss` subsystem library.

mke2fs is used to create a second extended file system on the given device.

mklost+found is used to create a `lost+found` directory on a second extended file system. It pre-allocates disk blocks to this directory to lighten the task of e2fsck.

resize2fs can be used to enlarge or shrink an ext2 file system.

tune2fs is used adjust tunable file system parameters on a second extended file system.

uuidgen creates new UUID. Each new UUID can reasonably be considered unique among all UUIDs created, on the local system and on other systems, in the past and in the future.

libblkid contains routines for device identification and token extraction.

libcom_err is the common error display routine.

libe2p is used by dumpe2fs, chattr, and lsattr.

libext2fs contains routines to enable user-level programs to manipulate an ext2 file system.

libss is used by debugfs.

libuuid contains routines for generating unique identifiers for objects that may be accessible beyond the local system.

Grep-2.5.1

The Grep package contains programs for searching through files.

```
Approximate build time:  0.1 SBU
Required disk space:     5.8 MB
```

Grep installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Make, Sed, Texinfo.

Installation of Grep

Prepare Grep for compilation:

```
./configure --prefix=/usr --bindir=/bin --with-included-regex
```

Compile the package:

```
make
```

To test the results, issue: **make check**.

Now install the package:

```
make install
```

Contents of Grep

Installed programs: egrep (link to grep), fgrep (link to grep) and grep

Short descriptions

egrep prints lines matching an extended regular expression.

fgrep prints lines matching a list of fixed strings.

grep prints lines matching a basic regular expression.

Grub-0.94

The Grub package contains the GRand Unified Bootloader.

```
Approximate build time:  0.2 SBU
Required disk space:     10 MB
```

Grub installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Ncurses, Sed.

Installation of Grub

This package is known to behave badly when you have changed its default optimization flags (including the -march and -mcpu options). Therefore, if you have defined any environment variables that override default optimizations, such as CFLAGS and CXXFLAGS, we recommend un-setting them when building Grub.

First prepare Grub for compilation:

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

Compile the package:

```
make
```

Now install it:

```
make install
mkdir /boot/grub
cp /usr/share/grub/i386-pc/stage{1,2} /boot/grub
```

Replace `i386-pc` with whatever directory is appropriate for your hardware.

The `i386-pc` directory also contains a number of `*stage1_5` files, different ones for different file systems. Have a look at the ones available and copy the appropriate ones to the `/boot/grub` directory. Most people will copy the `e2fs_stage1_5` and/or `reiserfs_stage1_5` files.

Contents of Grub

Installed programs: grub, grub-install, grub-md5-crypt, grub-terminfo and mbchk

Short descriptions

grub is the GRand Unified Bootloader's command shell.

grub-install installs GRUB on the given device.

grub-md5-crypt encrypts a password in MD5 format.

grub-terminfo generates a terminfo command from a terminfo name. It can be used if you have an uncommon terminal.

mbchk checks the format of a multi-boot kernel.

Gzip-1.3.5

The Gzip package contains programs for compressing and decompressing files.

```
Approximate build time:  0.1 SBU
Required disk space:     2.6 MB
```

Gzip installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of Gzip

Prepare Gzip for compilation:

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

The **gzexe** script has the location of the **gzip** binary hard-wired into it. Because we later change the location of the binary, the following command ensures that the new location gets placed into the script:

```
cp gzexe.in{,.backup}
sed 's%"BINDIR"%/bin%' gzexe.in.backup > gzexe.in
```

Compile the package:

```
make
```

Install the package:

```
make install
```

Move the programs to the `/bin` directory:

```
mv /usr/bin/gzip /bin
rm /usr/bin/{gunzip,zcat}
ln -s gzip /bin/gunzip
ln -s gzip /bin/zcat
ln -s gunzip /bin/uncompress
```

Contents of Gzip

Installed programs: gunzip (link to gzip), gzexe, gzip, uncompress (link to gunzip), zcat (link to gzip), zcmp, zdiff, zegrep, zfgrep, zforce, zgrep, zless, zmore and znew

Short descriptions

gunzip decompresses gzipped files.

gzexe is used to create self-uncompressing executable files.

gzip compresses the given files, using Lempel-Ziv (LZ77) coding.

zcat uncompresses the given gzipped files to standard output.

zcmp runs cmp on gzipped files.

zdiff runs diff on gzipped files.

zegrep runs egrep on gzipped files.

zfgrep runs fgrep on gzipped files.

zforce forces a .gz extension on all given files that are gzipped files, so that gzip will not compress them again. This can be useful when file names were truncated during a file transfer.

zgrep runs grep on gzipped files.

zless runs less on gzipped files.

zmore runs more on gzipped files.

znew re-compresses files from compress format to gzip format -- .Z to .gz.

Man-1.5m2

The Man package contains programs for finding and viewing manual pages.

```
Approximate build time:  0.1 SBU
Required disk space:     1.9MB
```

Man installation depends on: Bash, Binutils, Coreutils, Gawk, GCC, Glibc, Grep, Make, Sed.

Installation of Man

We'll make three adjustments to the sources of Man.

The first is a patch which allows Man to work better with recent releases of Groff. In particular, man pages will now display using the full terminal width instead of being limited to 80 characters:

```
patch -Np1 -i ../man-1.5m2-80cols.patch
```

The second is a sed substitution to add the *-R* switch to the *PAGER* variable so that escape sequences are properly handled by Less:

```
sed -i 's/-is/&R/' configure
```

The third is also a sed substitution to comment out the “MANPATH /usr/man” line in the `man.conf` file to prevent redundant results when using programs such as **whatis**:

```
sed -i 's%MANPATH./usr/man%#&%' src/man.conf.in
```

Now prepare Man for compilation:

```
./configure -default -confdir=/etc
```

The meaning of the configure options:

- **`-default`**: This tells the configure script to select a sensible set of default options. For example: only English man pages, no message catalogs, man not suid, handle compressed man pages, compress cat pages, create cat pages whenever the appropriate directory exists, follow FHS by putting cat pages under /var/cache/man (provided that directory exists).
- **`-confdir=/etc`**: This tells the **man** program to look for the `man.conf` configuration file in the `/etc` directory.

Compile the package:

```
make
```

Lastly, install it:

```
make install
```

Note

If you wish to disable SGR (Select Graphic Rendition) escape sequences, you should edit the `man.conf` file and add the *-c* switch to NROFF.

You may want to also take a look at the BLFS page at http://www.linuxfromscratch.org/blfs/view/cvs/postlfs/compressdoc.html which deals with formatting and compression issues for man pages.

Contents of Man

Installed programs: apropos, makewhatis, man, man2dvi, man2html and whatis

Short descriptions

apropos searches the whatis database and displays the short descriptions of system commands that contain a given string.

makewhatis builds the whatis database. It reads all the manual pages in the manpath and for each page writes the name and a short description in the whatis database.

man formats and displays the requested on-line manual page.

man2dvi converts a manual page into dvi format.

man2html converts a manual page into html.

whatis searches the whatis database and displays the short descriptions of system commands that contain the given keyword as a separate word.

Make-3.80

The Make package contains a program for compiling large packages.

```
Approximate build time:  0.2 SBU
Required disk space:     8.8 MB
```

Make installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Sed.

Installation of Make

Prepare Make for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**.

Now install the package:

```
make install
```

Contents of Make

Installed program: make

Short description

make automatically determines which pieces of a large package need to be recompiled, and then issues the relevant commands.

Modutils-2.4.27

The Modutils package contains programs for handling kernel modules.

```
Approximate build time:  0.1 SBU
Approximate build time:  2.9 MB
```

Modutils installation depends on: Bash, Binutils, Bison, Coreutils, Diffutils, Flex, GCC, Glibc, Grep, M4, Make, Sed.

Installation of Modutils

Prepare Modutils for compilation:

```
./configure
```

Compile the package:

```
make
```

Install it:

```
make install
```

Contents of Modutils

Installed programs: depmod, genksyms, insmod, insmod_ksymoops_clean, kallsyms (link to insmod), kernelversion, ksyms (link to insmod), lsmod (link to insmod), modinfo, modprobe (link to insmod) and rmmod (link to insmod)

Short descriptions

depmod creates a dependency file, based on the symbols it finds in the existing set of modules. This dependency file is used by modprobe to automatically load the required modules.

genksyms generates symbol version information.

insmod installs a loadable module in the running kernel.

insmod_ksymoops_clean deletes saved ksyms and modules not accessed for two days.

kallsyms extracts all kernel symbols for debugging.

kernelversion reports the major version of the running kernel.

ksyms displays exported kernel symbols.

lsmod shows which modules are loaded.

modinfo examines an object file associated with a kernel module and displays any information that it can glean.

modprobe uses a dependency file, created by depmod, to automatically load the relevant modules.

rmmod unloads modules from the running kernel.

Patch-2.5.4

The Patch package contains a program for modifying files.

```
Approximate build time:  0.1 SBU
Required disk space:     1.9 MB
```

Patch installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

Installation of Patch

Prepare Patch for compilation (the preprocessor flag *-D_GNU_SOURCE* is only needed on PowerPCs, on other machines you can leave it out):

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

Compile the package:

```
make
```

Now install it:

```
make install
```

Contents of Patch

Installed program: patch

Short description

patch modifies files according to a patch file. A patch file normally is a difference listing created with the diff program. By applying these differences to the original files, patch creates the patched versions. Using patches instead of entirely new tarballs to keep your sources up-to-date can save you a lot of download time.

Procinfo-18

The Procinfo package contains programs for displaying system information.

```
Approximate build time:  0.1 SBU
Required disk space:     0.2 MB
```

Procinfo installation depends on: Binutils, GCC, Glibc, Make, Ncurses.

Installation of Procinfo

Compile Procinfo:

```
make LDLIBS=-lncurses
```

The meaning of the make parameter:

- **`LDLIBS=-lncurses`**: This tells Procinfo to use the `libncurses` library instead of the long-obsolete `libtermcap`.

Install the package:

```
make install
```

Contents of Procinfo

Installed programs: lsdev, procinfo and socklist

Short descriptions

lsdev lists the devices present in your system, and which IRQs (Interrupt ReQuest) and IO ports they use.

procinfo displays an overview of some of the information present in the virtual proc file system.

socklist lists the open sockets, reporting their type, port number, and other specifics.

Procps-3.2.1

The Procps package contains programs for monitoring processes.

```
Approximate build time:  0.1 SBU
Required disk space:     6.2 MB
```

Procps installation depends on: Bash, Binutils, Coreutils, GCC, Glibc, Make, Ncurses.

Installation of Procps

Compile Procps:

```
make
```

Install it:

```
make install
```

Remove a spurious library link:

```
rm /lib/libproc.so
```

Contents of Procps

Installed programs: free, kill, pgrep, pkill, pmap, ps, skill, snice, sysctl, tload, top, uptime, vmstat, w and watch

Installed library: libproc.so

Short descriptions

free reports the amount of free and used memory in the system, both physical and swap memory.

kill is used to send signals to processes.

pgrep looks up processes based on their name and other attributes.

pkill signals processes based on their name and other attributes.

pmap reports the memory map of the given process.

ps gives a snapshot of the current processes.

skill sends signals to processes matching the given criteria.

snice changes the scheduling priority of processes matching the given criteria.

sysctl modifies kernel parameters at run time.

tload prints a graph of the current system load average.

top displays the top CPU processes. It provides an ongoing look at processor activity in real time.

uptime reports how long the system has been running, how many users are logged on, and the system load averages.

vmstat reports virtual memory statistics, giving information about processes, memory, paging, block IO, traps, and CPU activity.

w shows which users are currently logged on, where and since when.

watch runs a given command repeatedly, displaying the first screen-full of its output. This allows you to watch the output change over time.

libproc contains the functions used by most programs in this package.

Psmisc-21.4

The Psmisc package contains programs for displaying information on processes.

```
Approximate build time:  0.1 SBU
Required disk space:     2.2 MB
```

Psmisc installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Ncurses, Sed.

Installation of Psmisc

Prepare Psmisc for compilation:

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

The meaning of the configure option:

- **`--exec-prefix=/`**: This causes the binaries to be installed in `/bin` instead of `/usr/bin`. As the Psmisc programs are often used in bootscripts, they should be available also when the `/usr` file system isn't mounted.

Compile the package:

```
make
```

Now install it:

```
make install
```

There is no reason for the `pstree` and `pstree.x11` programs to reside in `/bin`. We therefore move them to `/usr/bin`. Also, there is no need for `pstree.x11` to exist as a separate program. We therefore make it a symbolic link to `pstree`:

```
mv /bin/pstree* /usr/bin
ln -sf pstree /usr/bin/pstree.x11
```

By default Psmisc's **pidof** program isn't installed. Generally, this isn't a problem because we later install the Sysvinit package, which provides a better **pidof** program. But if you're not going to use Sysvinit, you should complete the installation of Psmisc by creating the following symlink:

```
ln -s killall /bin/pidof
```

Contents of Psmisc

Installed programs: fuser, killall, pstree and pstree.x11 (link to pstree)

Short descriptions

fuser reports the PIDs of processes that use the given files or file systems.

killall kills processes by name. It sends a signal to all processes running any of the given commands.

pstree displays running processes as a tree.

pstree.x11 same as pstree except that it waits for confirmation before exiting.

Shadow-4.0.4.1

The Shadow package contains programs for handling passwords in a secure way.

```
Approximate build time:  0.4 SBU
Required disk space:     11 MB
```

Shadow installation depends on: Bash, Binutils, Bison, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Shadow

Shadow hard-wires the path to the **passwd** binary within the binary itself, but does this the wrong way. If a **passwd** binary is not present before installing Shadow, the package incorrectly assumes it is going to be located at `/bin/passwd`, but then installs it as `/usr/bin/passwd`. This will lead to errors about not finding `/bin/passwd`. To work around this bug, create a dummy `passwd` file, so that it gets hard-wired properly:

```
touch /usr/bin/passwd
```

Now prepare Shadow for compilation:

```
./configure --libdir=/usr/lib --enable-shared
```

Work around a problem that prevents Shadow's internationalization from working:

```
echo '#define HAVE_SETLOCALE 1' >> config.h
```

Compile the package:

```
make
```

Then install it:

```
make install
```

Shadow uses two files to configure authentication settings for the system. Install these two config files:

```
cp etc/{limits,login.access} /etc
```

Instead of using the default *crypt* method, we want to use the more secure *MD5* method of password encryption, which also allows passwords longer than 8 characters. We also need to change the obsolete `/var/spool/mail` location for user mailboxes that Shadow uses by default to the `/var/mail` location used currently. We accomplish both these things by changing the relevant configuration file while copying it to its destination (it's probably better to cut-and-paste this rather than try and type it all in):

```
sed -e 's%#MD5_CRYPT_ENAB.no%MD5_CRYPT_ENAB yes%' \
    -e 's%/var/spool/mail%/var/mail%' \
    etc/login.defs.linux > /etc/login.defs
```

Move some misplaced symlinks to their proper locations:

```
mv /bin/sg /usr/bin
mv /bin/vigr /usr/sbin
```

And move Shadow's dynamic libraries to a more appropriate location:

```
mv /usr/lib/lib{shadow,misc}.so.0* /lib
```

As some packages expect to find the just-moved libraries in `/usr/lib`, create the following symlinks:

```
ln -sf ../../lib/libshadow.so.0 /usr/lib/libshadow.so
ln -sf ../../lib/libmisc.so.0 /usr/lib/libmisc.so
```

The -D option of the `useradd` program requires this directory for it to work properly:

```
mkdir /etc/default
```

Coreutils has already installed a better **groups** program in `/usr/bin`. Remove the one installed by Shadow:

```
rm /bin/groups
```

Configuring Shadow

This package contains utilities to add, modify and delete users and groups, set and change their passwords, and other such administrative tasks. For a full explanation of what *password shadowing* means, see the `doc/HOWTO` file within the unpacked source tree. There's one thing to keep in mind if you decide to use Shadow support: programs that need to verify passwords (display managers, ftp programs, pop3 daemons, and the like) need to be *shadow-compliant*, that is they need to be able to work with shadowed passwords.

To enable shadowed passwords, run the following command:

```
pwconv
```

To enable shadowed group passwords, run:

```
grpconv
```

Under normal circumstances, you won't have created any passwords yet. However, if returning to this section later to enable shadowing, you should reset any current user passwords with the **passwd** command or any group passwords with the **gpasswd** command.

Setting the root password

Choose a password for user root and set it via:

```
passwd root
```

Contents of Shadow

Installed programs: chage, chfn, chpasswd, chsh, dpasswd, expiry, faillog, gpasswd, groupadd, groupdel, groupmod, groups, grpck, grpconv, grpunconv, lastlog, login, logoutd, mkpasswd, newgrp, newusers, passwd, pwck, pwconv, pwunconv, sg (link to newgrp), useradd, userdel, usermod, vigr (link to vipw) and vipw

Short descriptions

chage is used to change the maximum number of days between obligatory password changes.

chfn is used to change a user's full name and some other info.

chpasswd is used to update the passwords of a whole series of user accounts in one go.

chsh is used to change a user's default login shell.

dpasswd is used to change dial-up passwords for user login shells.

expiry checks and enforces the current password expiration policy.

faillog is used to examine the log of login failures, to set a maximum number of failures before an account is blocked, or to reset the failure count.

gpasswd is used to add and delete members and administrators to groups.

groupadd creates a group with the given name.

groupdel deletes the group with the given name.

groupmod is used to modify the given group's name or GID.

groups reports the groups of which the given users are members.

grpck verifies the integrity of the group files, `/etc/group` and `/etc/gshadow`.

grpconv creates or updates the shadow group file from the normal group file.

grpunconv updates `/etc/group` from `/etc/gshadow` and then deletes the latter.

lastlog reports the most recent login of all users, or of a given user.

login is used by the system to let users sign on.

logoutd is a daemon used to enforce restrictions on log-on time and ports.

mkpasswd encrypts the given password using the also given perturbation.

newgrp is used to change the current GID during a login session.

newusers is used to create or update a whole series of user accounts in one go.

passwd is used to change the password for a user or group account.

pwck verifies the integrity of the password files, `/etc/passwd` and `/etc/shadow`.

pwconv creates or updates the shadow password file from the normal password file.

pwunconv updates `/etc/passwd` from `/etc/shadow` and then deletes the latter.

sg executes a given command while the user's GID is set to that of the given group.

useradd creates a new user with the given name, or updates the default new-user information.

userdel deletes the given user account.

usermod is used to modify the given user's login name, UID (User Identification), shell, initial group, home directory, and the like.

vigr can be used to edit the `/etc/group` or `/etc/gshadow` files.

vipw can be used to edit the `/etc/passwd` or `/etc/shadow` files.

libmisc...

libshadow contains functions used by most programs in this package.

Sysklogd-1.4.1

The Sysklogd package contains programs for logging system messages, such as those given by the kernel when unusual things happen.

```
Approximate build time:  0.1 SBU
Required disk space:     0.5 MB
```

Sysklogd installation depends on: Binutils, Coreutils, GCC, Glibc, Make.

Installation of Sysklogd

Compile Sysklogd:

```
make
```

Now install it:

```
make install
```

Configuring Sysklogd

Create a new `/etc/syslog.conf` file by running the following:

```
cat > /etc/syslog.conf << "EOF"
# Begin /etc/syslog.conf

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 *

# End /etc/syslog.conf
EOF
```

Contents of Sysklogd

Installed programs: klogd and syslogd

Short descriptions

klogd is a system daemon for intercepting and logging kernel messages.

syslogd logs the messages that system programs offer for logging. Every logged message contains at least a date stamp and a hostname, and normally the program's name too, but that depends on how trusting the logging daemon is told to be.

Sysvinit-2.85

The Sysvinit package contains programs for controlling the startup, running, and shutdown of your system.

```
Approximate build time:  0.1 SBU
Required disk space:     0.9 MB
```

Sysvinit installation depends on: Binutils, Coreutils, GCC, Glibc, Make.

Installation of Sysvinit

When run-levels are changed (for example, when halting the system), **init** sends termination signals to those processes that **init** itself started and that shouldn't be running in the new run-level. While doing this, **init** outputs messages like “Sending processes the TERM signal” which seem to imply that it is sending these signals to all currently running processes. To avoid this misinterpretation, you can modify the source so that these messages read like “Sending processes started by init the TERM signal” instead:

```
cp src/init.c{,.backup}
sed 's/Sending processes/& started by init/g' \
    src/init.c.backup > src/init.c
```

Compile Sysvinit:

```
make -C src
```

Then install it:

```
make -C src install
```

Configuring Sysvinit

Create a new `/etc/inittab` file by running the following:

```
cat > /etc/inittab << "EOF"
# Begin /etc/inittab

id:3:initdefault:

si::sysinit:/etc/rc.d/init.d/rc sysinit

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 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

# End /etc/inittab
EOF
```

Contents of Sysvinit

Installed programs: halt, init, killall5, last, lastb (link to last), mesg, pidof (link to killall5), poweroff (link to halt), reboot (link to halt), runlevel, shutdown, sulogin, telinit (link to init), utmpdump and wall

Short descriptions

halt normally invokes shutdown with the -h flag, except when already in run-level 0, then it tells the kernel to halt the system. But first it notes in the file `/var/log/wtmp` that the system is being brought down.

init is the mother of all processes. It reads its commands from `/etc/inittab`, which normally tell it which scripts to run for which run-level, and how many gettys to spawn.

killall5 sends a signal to all processes, except the processes in its own session -- so it won't kill the shell running the script that called it.

last shows which users last logged in (and out), searching back through the file `/var/log/wtmp`. It can also show system boots and shutdowns, and run-level changes.

lastb shows the failed login attempts, as logged in `/var/log/btmp`.

mesg controls whether other users can send messages to the current user's terminal.

pidof reports the PIDs of the given programs.

poweroff tells the kernel to halt the system and switch off the computer. But see halt.

reboot tells the kernel to reboot the system. But see halt.

runlevel reports the previous and the current run-level, as noted in the last run-level record in `/var/run/utmp`.

shutdown brings the system down in a secure way, signaling all processes and notifying all logged-in users.

sulogin allows the superuser to log in. It is normally invoked by init when the system goes into single user mode.

telinit tells init which run-level to enter.

utmpdump displays the content of the given login file in a friendlier format.

wall writes a message to all logged-in users.

Tar-1.13.94

The Tar package contains an archiving program.

```
Approximate build time:  0.2 SBU
Required disk space:     10 MB
```

Tar installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Sed.

Installation of Tar

Prepare Tar for compilation:

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

Compile the package:

```
make
```

To test the results, issue: **make check**.

Now install the package:

```
make install
```

Contents of Tar

Installed programs: rmt and tar

Short descriptions

rmt is used to remotely manipulate a magnetic tape drive, through an interprocess communication connection.

tar is used to create and extract files from archives, also known as tarballs.

Util-linux-2.12a

The Util-linux package contains miscellaneous utility programs. Among them are utilities for handling file systems, consoles, partitions, and messages.

```
Approximate build time:  0.2 SBU
Required disk space:     16 MB
```

Util-linux installation depends on: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Ncurses, Sed, Zlib.

FHS compliance notes

The FHS recommends that we use `/var/lib/hwclock`, instead of the usual `/etc`, as the location for the `adjtime` file. To make the **hwclock** program FHS-compliant, run the following:

```
cp hwclock/hwclock.c{,.backup}
sed 's%etc/adjtime%var/lib/hwclock/adjtime%' \
    hwclock/hwclock.c.backup > hwclock/hwclock.c
mkdir -p /var/lib/hwclock
```

Installation of Util-linux

Prepare Util-linux for compilation:

```
./configure
```

Compile the package:

```
make HAVE_KILL=yes HAVE_SLN=yes
```

The meaning of the make parameters:

- **`HAVE_KILL=yes`**: This prevents the **kill** program (already installed by Procps) from being built and installed again.
- **`HAVE_SLN=yes`**: This prevents the **sln** program (a statically linked **ln** already installed by Glibc) from being built and installed again.

Now install the package:

```
make HAVE_KILL=yes HAVE_SLN=yes install
```

Contents of Util-linux

Installed programs: agetty, arch, blockdev, cal, cfdisk, chkdupexe, col, colcrt, colrm, column, ctrlaltdel, cytune, ddate, dmesg, elvtune, fdformat, fdisk, fsck.cramfs, fsck.minix, getopt, hexdump, hwclock, ipcrm, ipcs, isosize, line, logger, look, losetup, mcookie, mkfs, mkfs.bfs, mkfs.cramfs, mkfs.minix, mkswap, more, mount, namei, pg, pivot_root, ramsize (link to rdev), raw, rdev, readprofile, rename, renice, rev, rootflags (link to rdev), script, setfdprm, setsid, setterm, sfdisk, swapoff (link to swapon), swapon, tunelp, ul, umount, vidmode (link to rdev), whereis and write

Short descriptions

agetty opens a tty port, prompts for a login name, and then invokes the login program.

arch reports the machine's architecture.

blockdev allows you to call block device ioctls from the command line.

cal displays a simple calendar.

cfdisk is used to manipulate the partition table of the given device.

chkdupexe finds duplicate executables.

col filters out reverse line feeds.

colcrt is used to filter nroff output for terminals that lack some capabilities such as overstriking and half-lines.

colrm filters out the given columns.

column formats a given file into multiple columns.

ctrlaltdel sets the function of the Ctrl+Alt+Del key combination to a hard or a soft reset.

cytune was used to tune the parameters of the serial line drivers for Cyclades cards.

ddate gives the Discordian date, or converts the given Gregorian date to a Discordian one.

dmesg dumps the kernel boot messages.

elvtune can be used to tune the performance and interactivity of a block device.

fdformat low-level formats a floppy disk.

fdisk could be used to manipulate the partition table of the given device.

fsck.cramfs performs a consistency check on the Cramfs file system on the given device.

fsck.minix performs a consistency check on the Minix file system on the given device.

getopt parses options in the given command line.

hexdump dumps the given file in hexadecimal, or in another given format.

hwclock is used to read or set the system's hardware clock, also called the RTC (Real-Time Clock) or BIOS (Basic Input-Output System) clock.

ipcrm removes the given IPC resource.

ipcs provides IPC status information.

isosize reports the size of an iso9660 file system.

line copies a single line.

logger enters the given message into the system log.

look displays lines that begin with the given string.

losetup is used to set up and control loop devices.

mcookie generates magic cookies, 128-bit random hexadecimal numbers, for xauth.

mkfs is used to build a file system on a device (usually a hard disk partition).

mkfs.bfs creates an SCO (Santa Cruz Operations) bfs file system.

mkfs.cramfs creates a cramfs file system.

mkfs.minix creates a Minix file system.

mkswap initializes the given device or file to be used as a swap area.

more is a filter for paging through text one screen full at a time. But less is much better.

mount attaches the file system on the given device to a specified directory (thus hiding the contents of that directory) in the file-system tree.

namei shows the symbolic links in the given pathnames.

pg displays a text file one screen full at a time.

pivot_root makes the given file system the new root file system of the current process.

ramsize is used to set the size of the RAM disk in a bootable image.

rdev is used to query and set the root device and other things in a bootable image.

readprofile reads kernel profiling information.

rename renames the given files, replacing a given string with another.

renice is used to alter the priority of running processes.

rev reverses the lines of a given file.

rootflags is used to set the rootflags in a bootable image.

script makes a typescript of a terminal session, of everything printed to the terminal.

setfdprm sets user-provided floppy disk parameters.

setsid runs the given program in a new session.

setterm is used to set terminal attributes.

sfdisk is a disk partition table manipulator.

swapdev is used to set the swap device in a bootable image.

swapoff disables devices and files for paging and swapping.

swapon enables devices and files for paging and swapping.

tunelp is used to tune the parameters of the line printer.

ul is a filter for translating underscores into escape sequences indicating underlining for the terminal in use.

umount disconnects a file system from the system's file tree.

vidmode could be used to set the video mode in a bootable image.

whereis reports the location of binary, the source, and the manual page for the given command.

write sends a message to the given user, *if* that user has not disabled such messages.

GCC-2.95.3

```
Approximate build time:  1.5 SBU
Approximate build time:  130 MB
```

Installation of GCC

This package is known to behave badly when you have changed its default optimization flags (including the -march and -mcpu options). Therefore, if you have defined any environment variables that override default optimizations, such as CFLAGS and CXXFLAGS, we recommend un-setting or modifying them when building GCC.

This is an older release of GCC which we are going to install for the purpose of compiling the Linux kernel in Chapter 8[p.168]. This version is recommended by the kernel developers when you need absolute stability. Later versions of GCC have not received as much testing for Linux kernel compilation. Using a later version is likely to work, however, we recommend adhering to the kernel developer's advice and using the version here to compile your kernel.

Note

We don't install the C++ compiler or libraries here. However, there may be reasons why you would want to install them. More information can be found at http://www.linuxfromscratch.org/blfs/view/stable/general/gcc2.html.

We'll install this older release of GCC into the non-standard prefix of `/opt` so as to avoid interfering with the system GCC already installed in `/usr` .

Apply the patches and make a small adjustment:

```
patch -Np1 -i ../gcc-2.95.3-2.patch
patch -Np1 -i ../gcc-2.95.3-no-fixinc.patch
patch -Np1 -i ../gcc-2.95.3-returntype-fix.patch
echo timestamp > gcc/cstamp-h.in
```

The GCC documentation recommends building GCC outside of the source directory in a dedicated build directory:

```
mkdir ../gcc-2-build
cd ../gcc-2-build
```

Compile and install the compiler:

```
../gcc-2.95.3/configure --prefix=/opt/gcc-2.95.3 \
    --enable-shared --enable-languages=c \
    --enable-threads=posix
make bootstrap
make install
```

About debugging symbols

Most programs and libraries are, by default, compiled with debugging symbols included (with **gcc**'s *-g* option). This means that, when debugging a program or library that was compiled with debugging information included, the debugger can give you not only memory addresses but also the names of the routines and variables.

The inclusion of these debugging symbols, however, enlarges a program or library significantly. To get an idea of the amount of space these symbols occupy, have a look at the following:

- a bash binary with debugging symbols: 1200 KB
- a bash binary without debugging symbols: 480 KB
- Glibc and GCC files (/lib and /usr/lib) with debugging symbols: 87 MB
- Glibc and GCC files without debugging symbols: 16 MB

Sizes may vary somewhat, depending on which compiler was used and which C library, but when comparing programs with and without debugging symbols the difference will generally be a factor between 2 and 5.

As most people will probably never use a debugger on their system software, a lot of disk space can be regained by removing these symbols. For your convenience, the next section shows how to strip all debugging symbols from all programs and libraries. Information on other ways of optimizing your system can be found in the hint at http://www.linuxfromscratch.org/hints/downloads/files/optimization.txt.

Stripping again

If you are not a programmer and don't plan to do any debugging on your system software, you can shrink your system by about 200 MB by removing the debugging symbols from binaries and libraries. This causes no inconvenience other than not being able to debug the software fully any more.

Most people who use the command mentioned below don't experience any problems. But it is easy to make a typo and render your new system unusable, so before running the strip command it is probably a good idea to make a backup of the current situation.

If you are going to perform the stripping, special care is needed to ensure you're not running any of the binaries that are about to be stripped. If you're not sure whether you entered chroot with the command given in the section called “Entering the chroot environment”[p.68], then first exit from chroot:

```
logout
```

Then reenter it with:

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

Now you can safely strip the binaries and libraries:

```
/tools/bin/find /{,usr/}{bin,lib,sbin} -type f \
   -exec /tools/bin/strip --strip-debug '{}' ';'
```

A large number of files will be reported as having their file format not recognized. These warnings can be safely ignored, they just mean that those files are scripts instead of binaries, no harm is done.

If you are really tight on disk space, you may want to use *--strip-all* on the binaries in `/{,usr/}{bin,sbin}` to gain several more megabytes. But do *not* use this option on libraries: they would be destroyed.

Cleaning up

From now on, when you exit the chroot environment and wish to reenter it, you should use the following modified chroot command:

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

The reason for this is that, since the programs in `/tools` are no longer needed, you may want to delete the whole directory and regain the space. Before actually deleting the directory, exit from chroot and reenter it with the above command. Also, before removing `/tools`, you may want to tar it up and store it in a safe place, in case you want to build another LFS system soon.

Note

Removing `/tools` will also remove the temporary copies of Tcl, Expect and DejaGnu, which were used for running the toolchain tests. If you want to use these programs later on, you will need to recompile and re-install them. The installation instructions are the same as in Chapter 5[p.26], apart from changing the prefix from */tools* to */usr*. The BLFS book discusses a slightly different approach to installing Tcl, see http://www.linuxfromscratch.org/blfs/.

You may also want to move the packages and patches stored in `/sources` to a more usual location, such as `/usr/src/packages`, and remove the directory -- or simply delete the whole directory if you've burned its contents on a CD).

Chapter 7. Setting up system boot scripts

Introduction

In this chapter we will install the bootscripts and set them up properly. Most of these scripts will work without needing to modify them, but a few require additional configuration files, since they deal with hardware dependent information.

We have chosen to use System-V style init scripts simply because they are widely used and we feel comfortable with them. If you would prefer to try something else: Marc Heerdink has written a hint about BSD style init scripts, to be found at http://www.linuxfromscratch.org/hints/downloads/files/bsd-init.txt. And if you'd like something more radical, search the LFS mailing lists for “depinit”.

If you decide to use some other style of init scripts, you can skip this chapter and move on to Chapter 8[p.168].

LFS-Bootscripts-2.0.5

The LFS-Bootscripts package contains a set of bootscripts.

```
Approximate build time:  0.1 SBU
Required disk space:     0.3 MB
```

LFS-Bootscripts installation depends on: Bash, Coreutils.

Installation of LFS-Bootscripts

Installation of the bootscripts is very simple:

```
make install
```

Contents of LFS-bootscripts

Installed scripts: checkfs, cleanfs, functions, halt, ifdown, ifup, loadkeys, localnet, mountfs, mountkernfs, network, rc, reboot, sendsignals, setclock, static, swap, sysklogd and template

Short descriptions

The **checkfs** script checks the file systems just before they are mounted (with the exception of journal and network based file systems).

The **cleanfs** script removes files that shouldn't be preserved between reboots, such as those in `/var/run/` and `/var/lock/`. It re-creates `/var/run/utmp` and removes the possibly present `/etc/nologin`, `/fastboot` and `/forcefsck` files.

The **functions** script contains functions shared among different scripts, such as error and status checking.

The **halt** script halts the system.

The **ifdown** and **ifup** scripts assist the network script with network devices.

The **loadkeys** script loads the keymap table you specified as proper for your keyboard layout.

The **localnet** script sets up the system's hostname and local loopback device.

The **mountfs** script mounts all file systems that aren't marked *noauto* or aren't network based.

The **mountkernfs** script is used to mount kernel-provided file systems, such as /proc.

The **network** script sets up network interfaces, such as network cards, and sets up the default gateway where applicable.

The **rc** script is the master run-level control script. It is responsible for running all the other scripts one-by-one, in a sequence determined by the name of the symbolic links being processed.

The **reboot** script reboots the system.

The **sendsignals** script makes sure every process is terminated before the system reboots or halts.

The **setclock** script resets the kernel clock to localtime in case the hardware clock isn't set to GMT time.

The **static** script provides the functionality needed to assign a static IP address to a network interface.

The **swap** script enables and disables swap files and partitions.

The **sysklogd** script starts and stops the system and kernel log daemons.

The **template** script is a template you can use to create your own bootscripts for your other daemons.

How does the booting process with these scripts work?

Linux uses a special booting facility named SysVinit. It's based on a concept of *run-levels*. It can be widely different from one system to another, so it can't be assumed that because things worked in <insert distro name> they should work like that in LFS too. LFS has its own way of doing things, but it respects generally accepted standards.

SysVinit (which we'll call *init* from now on) works using a run-levels scheme. There are 7 (from 0 to 6) run-levels (actually, there are more run-levels but they are for special cases and generally not used. The init man page describes those details), and each one of those corresponds to the things the computer is supposed to do when it starts up. The default run-level is 3. Here are the descriptions of the different run-levels as they are often implemented:

```
0: halt the computer
1: single-user mode
2: multi-user mode without networking
3: multi-user mode with networking
4: reserved for customization, otherwise does the same as 3
5: same as 4, it is usually used for GUI login (like X's xdm or KDE's kdm)
6: reboot the computer
```

The command used to change run-levels is **init <runlevel>** where <runlevel> is the target run-level. For example, to reboot the computer, a user would issue the **init 6** command. The **reboot** command is just an alias for it, as is the **halt** command an alias for **init 0**.

There are a number of directories under `/etc/rc.d` that look like like rc?.d (where ? is the number of the run-level) and rcsysinit.d all containing a number of symbolic links. Some begin with a K, the others begin with an S, and all of them have two numbers following the initial letter. The K means to stop (kill) a service, and the S means to start a service. The numbers determine the order in which the scripts are run, from 00 to 99; the lower the number the sooner it gets executed. When init switches to another run-level, the appropriate services get killed and others get started.

The real scripts are in /etc/rc.d/init.d. They do all the work, and the symlinks all point to them. Killing links and starting links point to the same script in /etc/rc.d/init.d. That's because the scripts can be called with different parameters like start, stop, restart, reload, status. When a K link is encountered, the appropriate script is run with the stop argument. When an S link is encountered, the appropriate script is run with the start argument.

There is one exception. Links that start with an S in the rc0.d and rc6.d directories will not cause anything to be started. They will be called with the parameter *stop* to stop something. The logic behind it is that when you are going to reboot or halt the system, you don't want to start anything, only stop the system.

These are descriptions of what the arguments make the scripts do:

- *start*: The service is started.
- *stop*: The service is stopped.
- *restart*: The service is stopped and then started again.
- *reload*: The configuration of the service is updated. This is used after the configuration file of a service was modified, when the service doesn't need to be restarted.
- *status*: Tells if the service is running and with which PIDs.

Feel free to modify the way the boot process works (after all, it's your own LFS system). The files given here are just an example of how it can be done in a nice way (well, what we consider nice -- you may hate it).

Configuring the setclock script

This setclock script reads the time from your hardware clock, also known as BIOS or CMOS (Complementry Metal-Oxide Semiconductor) clock, and either converts that time to localtime using the `/etc/localtime` file (if the hardware clock is set to GMT) or not (if the hardware clock is already set to localtime). There is no way to auto-detect whether the hardware clock is set to GMT or not, so we need to configure that here ourselves.

Change the value of the *UTC* variable below to a *0* (zero) if your hardware clock is not set to GMT time.

Create a new file `/etc/sysconfig/clock` by running the following:

```
cat > /etc/sysconfig/clock << "EOF"
# Begin /etc/sysconfig/clock

UTC=1

# End /etc/sysconfig/clock
EOF
```

Now, you may want to take a look at a very good hint explaining how we deal with time on LFS at http://www.linuxfromscratch.org/hints/downloads/files/time.txt. It explains issues such as time zones, UTC, and the TZ environment variable.

Do I need the loadkeys script?

If you plan to compile the keymap directly in the kernel during Chapter 8[p.168] (see Kbd[p.128]), then strictly speaking you don't need to run this loadkeys script, since the kernel will set up the keymap for you. If you wish, you can still run the script, it isn't going to hurt you. Keeping it could even be beneficial, in case you run a lot of different kernels and can't be sure that the keymap is compiled into every one of them.

If you decided you don't need or don't want to use the loadkeys script, remove the `/etc/rc.d/rcsysinit.d/S70loadkeys` symlink.

Configuring the sysklogd script

The `sysklogd` script invokes the **syslogd** program with the *-m 0* option. This option turns off the periodic timestamp mark that **syslogd** writes to the log files every 20 minutes by default. If you want to turn on this periodic timestamp mark, edit the `sysklogd` script and make the changes accordingly. See **`man syslogd`** for more information.

Configuring the localnet script

Part of the localnet script is setting up the system's hostname. This needs to be configured in the /etc/sysconfig/network.

Create the /etc/sysconfig/network file and enter a hostname by running:

```
echo "HOSTNAME=lfs" > /etc/sysconfig/network
```

“lfs” needs to be replaced with the name the computer is to be called. You should not enter the FQDN (Fully Qualified Domain Name) here. That information will be put in the `/etc/hosts` file later on.

Creating the /etc/hosts file

If a network card is to be configured, you have to decide on the IP-address, FQDN and possible aliases for use in the /etc/hosts file. The syntax is:

```
<IP address> myhost.example.org aliases
```

Unless your computer is to be visible to the Internet (e.g. you have a registered domain and a valid block of assigned IP addresses - most of us don't have this)you should make sure that the IP-address is in the private network IP-address range. Valid ranges are:

```
        Class Networks
        A     10.0.0.0
        B     172.16.0.0 through 172.31.0.0
        C     192.168.0.0 through 192.168.255.0
```

A valid IP address could be 192.168.1.1. A valid FQDN for this IP could be www.linuxfromscratch.org (not recommended as this is a valid registered domain address and could cause your domain name server problems).

If you aren't going to use a network card, you still need to come up with a FQDN. This is necessary for certain programs to operate correctly.

If a network card is not going to be configured, create the `/etc/hosts` file by running:

```
cat > /etc/hosts << "EOF"
# Begin /etc/hosts (no network card version)

127.0.0.1 <value of HOSTNAME>.example.org <value of HOSTNAME> localhost

# End /etc/hosts (no network card version)
EOF
```

If a network card is to be configured, create the `/etc/hosts` file by running:

```
cat > /etc/hosts << "EOF"
# Begin /etc/hosts (network card version)

127.0.0.1 localhost
192.168.1.1 <value of HOSTNAME>.example.org <value of HOSTNAME>

# End /etc/hosts (network card version)
EOF
```

Of course, the 192.168.1.1 and <value of HOSTNAME>.example.org have to be changed to your liking (or requirements if assigned an IP-address by a network/system administrator and this machine is planned to be connected to an existing network).

Configuring the network script

This section only applies if you're going to configure a network card.

If you don't have any network cards, you are most likely not going to create any configuration files relating to network cards. If that is the case, you must remove the `network` symlinks from all the run-level directories (`/etc/rc.d/rc*.d`)

Configuring default gateway

If you're on a network you may need to set up the default gateway (a node on your network that provides access to other networks) for this machine. This is done by adding the proper values to the /etc/sysconfig/network file by running the following:

```
cat >> /etc/sysconfig/network << "EOF"
GATEWAY=192.168.1.2
GATEWAY_IF=eth0
EOF
```

The values for GATEWAY and GATEWAY_IF need to be changed to match your network setup. GATEWAY contains the IP address of the default gateway, and GATEWAY_IF contains the network interface through which the default gateway can be reached.

Creating network interface configuration files

Which interfaces are brought up and down by the network script depends on the files in the /etc/sysconfig/network-devices directory. This directory should contain files in the form of ifconfig.xyz, where xyz is a network interface name (such as eth0 or eth0:1)

If you decide to rename or move this /etc/sysconfig/network-devices directory, make sure you update the /etc/sysconfig/rc file as well and update the network_devices by providing it with the new path.

Now, new files are created in that directory. The following command creates a sample ifconfig.eth0 file:

```
cat > /etc/sysconfig/network-devices/ifconfig.eth0 << "EOF"
ONBOOT=yes
SERVICE=static
IP=192.168.1.1
NETMASK=255.255.255.0
BROADCAST=192.168.1.255
EOF
```

Of course, the values of those variables have to be changed in every file to match the proper setup. If the ONBOOT variable is set to yes, the network script will bring up the equivalent NIC (Network Interface Card) during the booting of the system. If set to anything but yes, the equivalent NIC will be ignored by the network script and not brought up.

The SERVICE entry defines the method of obtaining the IP address. The LFS bootscripts have a modular IP assignment format, and by creating additional files in /etc/sysconfig/network-devices/services, you can allow other IP assignment methods. This would commonly be used if you need DHCP, which is addressed in the BLFS book.

Creating the /etc/resolv.conf file

If you're going to be connected to the Internet then most likely you'll need some means of DNS name resolution to resolve Internet domain names to IP addresses. This is best achieved by placing the IP address of your DNS, available from your ISP (Internet Service Provider) or network administrator, into `/etc/resolv.conf`. Create the file by running the following:

```
cat > /etc/resolv.conf << "EOF"
# Begin /etc/resolv.conf

nameserver <IP address of your nameserver>

# End /etc/resolv.conf
EOF
```

Of course, replace <IP address of your nameserver> with the IP address of the DNS most appropriate for your setup. There will often be more than one entry (requirements demand secondary servers for fallback capability). The IP address may even be a router on your local network.

Chapter 8. Making the LFS system bootable

Introduction

This chapter will make LFS bootable. This chapter deals with creating a fstab file, building a kernel for the new LFS system and installing the Grub bootloader so that the LFS system can be selected for booting at startup.

Creating the /etc/fstab file

The `/etc/fstab` file is used by some programs to determine where file systems are to be mounted by default, which must be checked and in which order. Create a new file systems table like this:

```
cat > /etc/fstab << "EOF"
# Begin /etc/fstab

# file system  mount-point  fs-type  options         dump  fsck-order

/dev/xxx      /            fff      defaults        1     1
/dev/yyy      swap         swap     pri=1           0     0
proc          /proc        proc     defaults        0     0
devpts        /dev/pts     devpts   gid=4,mode=620  0     0
shm           /dev/shm     tmpfs    defaults        0     0

# End /etc/fstab
EOF
```

Of course, replace `xxx`, `yyy` and `fff` with the values appropriate for your system -- for example `hda2`, `hda5` and `reiserfs`. For all the details on the six fields in this table, see **man 5 fstab**.

When using a reiserfs partition, the *1 1* at the end of the line should be replaced with *0 0*, as such a partition does not need to be dumped or checked

The `/dev/shm` mount point for tmpfs is included to allow enabling POSIX shared memory. Your kernel must have the required support built into it for this to work -- more about this in the next section. Please note that currently very little software actually uses POSIX shared memory. Therefore you can consider the `/dev/shm` mount point optional. For more information, see `Documentation/filesystems/tmpfs.txt` in the kernel source tree.

There are other lines which you may consider adding to your `fstab` file. One example is a line to use if you intend to use USB devices:

```
usbfs       /proc/bus/usb  usbfs    defaults    0     0
```

This option will of course only work if you have the relevant support compiled into your kernel.

Linux-2.4.26

The Linux package contains the kernel and the header files.

```
Approximate build time:  All default options: 4.20 SBU
Required disk space:     All default options: 181 MB
```

Linux installation depends on: Bash, Binutils, Coreutils, Findutils, GCC, Glibc, Grep, Gzip, Make, Modutils, Perl, Sed.

Installation of the kernel

Building the kernel involves a few steps: configuration, compilation, and installation. If you don't like the way this book configures the kernel, view the `README` file in the kernel source tree for alternative methods.

Prepare for compilation by running the following command:

```
make mrproper
```

This ensures that the kernel tree is absolutely clean. The kernel team recommends that this command be issued prior to *each* kernel compilation. You shouldn't rely on the source tree being clean after un-tarring.

Configure the kernel via a menu-driven interface:

```
make menuconfig
```

`make oldconfig` may be more appropriate in some situations. See the `README` file for more information.

If you wish, you may skip kernel configuration by simply copying the kernel config file, `.config`, from your host system (assuming it is available) to the unpacked `linux-2.4.26` directory. However, we don't recommend this option. You're much better off exploring all the configuration menus and creating your own kernel configuration from scratch.

For POSIX shared memory support, ensure that the kernel config option “Virtual memory file system support” is enabled. It resides within the “File systems” menu and is normally enabled by default.

Verify dependencies and create dependency information files:

```
make CC=/opt/gcc-2.95.3/bin/gcc dep
```

Compile the kernel image:

```
make CC=/opt/gcc-2.95.3/bin/gcc bzImage
```

Compile the drivers which have been configured as modules:

```
make CC=/opt/gcc-2.95.3/bin/gcc modules
```

If you intend to use kernel modules, you will need an `/etc/modules.conf` file. Information pertaining to modules and to kernel configuration in general may be found in the kernel documentation, which is found in the `linux-2.4.26/Documentation` directory. The modules.conf man page and the kernel HOWTO at http://www.tldp.org/HOWTO/Kernel-HOWTO.html may also be of interest to you.

Install the modules:

```
make CC=/opt/gcc-2.95.3/bin/gcc modules_install
```

If you have a lot of modules and very little space, you may want to consider stripping and compressing the modules. For most people such compression isn't worth the trouble, but if you're really pressed for space, then have a look at http://www.linux-mips.org/archives/linux-mips/2002-04/msg00031.html.

As nothing is complete without documentation, build the manual pages that come with the kernel:

```
make mandocs
```

And install these pages:

```
cp -a Documentation/man /usr/share/man/man9
```

Kernel compilation has finished but more steps are required to complete the installation. Some files need to be copied to the `/boot` directory.

The path to the kernel image may vary depending on the platform you're using. Issue the following command to install the kernel:

```
cp arch/i386/boot/bzImage /boot/lfskernel
```

`System.map` is a symbol file for the kernel. It maps the function entry points of every function in the kernel API (Application Programming Interface), as well as the addresses of the kernel data structures for the running kernel. Issue the following command to install the map file:

```
cp System.map /boot
```

`.config` is the kernel configuration file that was produced by the **make menuconfig** step above. It contains all the config selections for the kernel that was just compiled. It's a good idea to keep this file for future reference:

```
cp .config /boot/config-lfskernel
```

It is important to note that the files in the kernel source directory are not owned by *root*. Whenever you unpack a package as user *root* (like we did here inside chroot), the files end up having the user and group IDs of whatever they were on the packager's computer. This is usually not a problem for any other package you install because you remove the source tree after the installation. But the Linux source tree is often kept around for a long time, so there's a chance that whatever user ID the packager used will be assigned to somebody on your machine and then that person would have write access to the kernel source.

If you are going to keep the kernel source tree around, you may want to run **`chown -R 0:0`** on the `linux-2.4.26` directory to ensure all files are owned by user *root*.

Contents of Linux

Installed files: the kernel, the kernel headers, and the System.map

Short descriptions

The *kernel* is the engine of your GNU/Linux system. When switching on your box, the kernel is the first part of your operating system that gets loaded. It detects and initializes all the components of your computer's hardware, then makes these components available as a tree of files to the software, and turns a single CPU into a multi-tasking machine capable of running scores of programs seemingly at the same time.

The *kernel headers* define the interface to the services that the kernel provides. The headers in your system's `include` directory should *always* be the ones against which Glibc was compiled and should therefore *not* be replaced when upgrading the kernel.

The `System.map` file is a list of addresses and symbols. It maps the entry points and addresses of all the functions and data structures in the kernel.

Making the LFS system bootable

Your shiny new LFS system is almost complete. One of the last things to do is ensure you can boot it. The instructions below apply only to computers of IA-32 architecture, meaning mainstream PCs. Information on “boot loading” for other architectures should be available in the usual resource-specific locations for those architectures.

Boot loading can be a complex area. First, a few cautionary words. You really should be familiar with your current boot loader and any other operating systems present on your hard drive(s) that you might wish to keep bootable. Please make sure that you have an emergency boot disk ready, so that you can rescue your computer if, by any chance, your computer becomes unusable (un-bootable).

Earlier, we compiled and installed the Grub boot loader software in preparation for this step. The procedure involves writing some special Grub files to specific locations on the hard drive. Before we get to that, we highly recommend that you create a Grub boot floppy diskette just in case. Insert a blank floppy diskette and run the following commands:

```
dd if=/boot/grub/stage1 of=/dev/fd0 bs=512 count=1
dd if=/boot/grub/stage2 of=/dev/fd0 bs=512 seek=1
```

Remove the diskette and store it somewhere safe. Now we'll run the **grub** shell:

```
grub
```

Grub uses its own naming structure for drives and partitions, in the form of (hdn,m), where *n* is the hard drive number, and *m* the partition number, both starting from zero. This means, for instance, that partition `hda1` is (hd0,0) to Grub, and `hdb2` is (hd1,1). In contrast to Linux, Grub doesn't consider CD-ROM drives to be hard drives, so if you have a CD on `hdb`, for example, and a second hard drive on `hdc`, that second hard drive would still be (hd1).

Using the above information, determine the appropriate designator for your root partition (or boot partition, if you use a separate one). For the following example, we'll assume your root (or separate boot) partition is `hda4`.

First, tell Grub where to search for its `stage{1,2}` files -- you can use the Tab key everywhere to make Grub show the alternatives:

```
root (hd0,3)
```

Warning

The following command will overwrite your current boot loader. Don't run the command if this is not what you want. For example, you may be using a third party boot manager to manage your MBR (Master Boot Record). In this scenario, it would probably make more sense to install Grub into the “boot sector” of the LFS partition, in which case this next command would become: **`setup (hd0,3)`**.

Tell Grub to install itself into the MBR (Master Boot Record) of `hda`:

```
setup (hd0)
```

If all is well, Grub will have reported finding its files in `/boot/grub`. That's all there is to it:

```
quit
```

Now we need to create a “menu list” file, defining Grub's boot menu:

```
cat > /boot/grub/menu.lst << "EOF"
# Begin /boot/grub/menu.lst

# By default boot the first menu entry.
default 0

# Allow 30 seconds before booting the default.
timeout 30

# Use prettier colors.
```

```
color green/black light-green/black

# The first entry is for LFS.
title LFS 5.1.1
root (hd0,3)
kernel --no-mem-option /boot/lfskernel root=/dev/hda4
EOF
```

Note

By default, Grub will automatically pass a “mem=xxx” command line argument to the kernel. However, Grub occasionally gets the amount of memory wrong which can lead to problems in some circumstances. It's best to disable this functionality and let the kernel determine the amount of memory itself, hence the use of the *--no-mem-option* above.

You may want to add an entry for your host distribution. It might look like this:

```
cat >> /boot/grub/menu.lst << "EOF"
title Red Hat
root (hd0,2)
kernel /boot/kernel-2.4.20 root=/dev/hda3
initrd /boot/initrd-2.4.20
EOF
```

Also, if you happen to dual-boot Windows, the following entry should allow booting it:

```
cat >> /boot/grub/menu.lst << "EOF"
title Windows
rootnoverify (hd0,0)
chainloader +1
EOF
```

If **info grub** doesn't tell you all you want to know, you can find more information regarding Grub on its website, located at: http://www.gnu.org/software/grub/.

Chapter 9. The End

The End

Well done! You have finished installing your LFS system. It may have been a long process, but we hope it was worth it. We wish you a lot of fun with your new shiny custom built Linux system.

It may be a good idea to create an `/etc/lfs-release` file. By having this file it is very easy for you (and for us if you are going to ask for help with something at some point) to find out which LFS version you have installed on your system. Create this file by running:

```
echo 5.1.1 > /etc/lfs-release
```

Get Counted

Want to be counted as an LFS user now that you have finished the book? Head over to http://www.linuxfromscratch.org/cgi-bin/lfscounter.cgi and register as an LFS user by entering your name and the first LFS version you have used.

Let's reboot into LFS now...

Rebooting the system

Now that all of the software has been installed, it is time to reboot your computer. However, you should be aware of a few things. The system you have created in this book is quite minimal, and most likely will not have the functionality you would need to be able to continue forward. By installing a few extra packages from the BLFS book while still in our current chroot environment, you can leave yourself in a much better position to continue on once you reboot into your new LFS installation. Installing a text mode web browser, such as Lynx, you can easily view the BLFS book in one virtual terminal, while building packages in another. The GPM package will also allow you to perform copy/paste actions in your virtual terminals. Lastly, if you are in a situation where static IP configuration does not meet your networking requirements, installing packages such as dhcpcd or ppp at this point might also be useful.

Now that we have said that, lets move on to booting our shiny new LFS installation for the first time! First exit from the chroot environment:

```
logout
```

Then unmount the virtual files systems:

```
umount $LFS/dev/pts
umount $LFS/proc
```

And unmount the LFS file system:

```
umount $LFS
```

If at the start you decided to create multiple partitions, you'll need to unmount the other partitions before unmounting the main one, like this:

```
umount $LFS/usr
umount $LFS/home
umount $LFS
```

Now reboot your system with:

```
shutdown -r now
```

Assuming the Grub boot loader was set up as outlined earlier, the menu is set to boot *LFS 5.1.1* automatically.

When the reboot is complete, your LFS system is ready for use and you can start adding your own software.

What now?

We thank you for reading the LFS Book and hope that you've found this book useful and worth your time.

Now that you have finished installing your LFS system, you may be wondering “What now?”. To answer that question, we have composed a list of resources for you.

- Beyond Linux From Scratch

  The Beyond Linux From Scratch book covers installation procedures for a wide range of software beyond the scope of the LFS Book. The BLFS project can be found at http://www.linuxfromscratch.org/blfs/.

- LFS Hints

  The LFS Hints are a collection of small, educational documents submitted by volunteers in the LFS community. The Hints are available at http://www.linuxfromscratch.org/hints/list.html.

- Mailing lists

  There are several LFS mailing lists you may subscribe to if you are in need of help. See Chapter 1 - Mailing lists[p.6] for more information.

- The Linux Documentation Project

  The goal of the Linux Documentation Project is to collaborate in all of the issues of Linux documentation. The LDP features a large collection of HOWTOs, Guides and man pages; it may be found at http://www.tldp.org/.

Index of packages and important installed files

Packages

Programs

xargs:	Findutils-4.1.20[p.96]	description[p.96]
xgettext:	Gettext-0.14.1[p.109]	description[p.110]
xsubpp:	Perl-5.8.4[p.115]	description[p.116]
xtrace:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
xxd:	Vim-6.2[p.100]	description[p.101]
yacc:	Bison-1.875[p.103]	description[p.103]
yes:	Coreutils-5.2.1[p.88]	description[p.91]
ylwrap:	Automake-1.8.4[p.120]	description[p.120]
ypdomainname:	Net-tools-1.60[p.111]	description[p.112]
zcat:	Gzip-1.3.5[p.134]	description[p.134]
zcmp:	Gzip-1.3.5[p.134]	description[p.134]
zdiff:	Gzip-1.3.5[p.134]	description[p.134]
zdump:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
zegrep:	Gzip-1.3.5[p.134]	description[p.134]
zfgrep:	Gzip-1.3.5[p.134]	description[p.134]
zforce:	Gzip-1.3.5[p.134]	description[p.134]
zgrep:	Gzip-1.3.5[p.134]	description[p.134]
zic:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
zless:	Gzip-1.3.5[p.134]	description[p.135]
zmore:	Gzip-1.3.5[p.134]	description[p.135]
znew:	Gzip-1.3.5[p.134]	description[p.135]
zsoelim:	Groff-1.19[p.105]	description[p.106]

Libraries

ld.so:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libanl:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libasprintf:	Gettext-0.14.1[p.109]	description[p.110]
libbfd:	Binutils-2.14[p.84]	description[p.85]
libblkid:	E2fsprogs-1.35[p.130]	description[p.131]
libBrokenLocale:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libbsd-compat:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libbz2*:	Bzip2-1.0.2[p.124]	description[p.125]
libc:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libcom_err:	E2fsprogs-1.35[p.130]	description[p.131]
libcrypt:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libdl:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libe2p:	E2fsprogs-1.35[p.130]	description[p.131]
libext2fs:	E2fsprogs-1.35[p.130]	description[p.131]
libfl.a:	Flex-2.5.4a[p.108]	description[p.108]
libform*:	Ncurses-5.4[p.98]	description[p.99]
libg:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libgcc*:	GCC-3.3.3[p.86]	description[p.87]
libgettextlib:	Gettext-0.14.1[p.109]	description[p.110]
libgettextpo:	Gettext-0.14.1[p.109]	description[p.110]
libgettextsrc:	Gettext-0.14.1[p.109]	description[p.110]
libiberty:	Binutils-2.14[p.84]	description[p.85]
libieee:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libltdl:	Libtool-1.5.6[p.123]	description[p.123]
libm:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libmagic:	File-4.09[p.122]	description[p.122]
libmcheck:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libmemusage:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libmenu*:	Ncurses-5.4[p.98]	description[p.99]
libmisc:	Shadow-4.0.4.1[p.144]	description[p.146]
libncurses*:	Ncurses-5.4[p.98]	description[p.99]
libnsl:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libnss*:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libopcodes:	Binutils-2.14[p.84]	description[p.85]
libpanel*:	Ncurses-5.4[p.98]	description[p.99]
libpcprofile:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libproc:	Procps-3.2.1[p.142]	description[p.142]
libpthread:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]
libresolv:	Glibc-2.3.3-lfs-5.1[p.77]	description[p.80]

Scripts

Others

manual pages: Man-pages-1.66[p.76] description[p.76]