NeuroOS: Building an Operating System with LFS

Esteban Bernal Correa
ebernalc@eafit.edu.co

Fidel Fernando Caicedo Castaño
ffcaicedoc@eafit.edu.co

University: EAFIT

October 29, 2023

Abstract

NeurOS is a showcase of how one with the required time and effort can build a functional operating system. It is built around 2 main components which are LFS and mate. The Linux From Scratch (LFS) project offers enthusiasts a unique opportunity to construct a Linux system entirely from source code, providing unparalleled insights into the inner workings of the operating system. This paper delves into the intricate process of building an LFS system, emphasizing the challenges and rewards of such an endeavor. Following the establishment of a basic LFS system, we transition into the integration of the MATE desktop environment, a continuation of the GNOME 2 desktop. MATE offers a traditional user experience, balancing performance with user-friendliness. Our exploration covers the dependencies and configurations required to seamlessly integrate MATE into an LFS build, culminating in a fully functional Linux system with a familiar graphical interface. This journey underscores the flexibility of LFS and showcases the potential for customization and optimization, catering to specific user preferences and hardware considerations.

Introduction

The journey of constructing a Linux system from the ground up is both intricate and enlightening. The Linux From Scratch (LFS) project offers a structured path to this endeavor, guiding enthusiasts through distinct stages: setting up a host environment, downloading and validating source packages, systematically building and installing these packages, and configuring foundational system settings. Each stage unveils layers of the Linux architecture, offering a deep understanding of the system's inner mechanics.

However, while the completion of an LFS system is an achievement in itself, the journey can be further enriched by integrating a user-friendly graphical interface. Enter the MATE desktop environment—a continuation of the GNOME 2 desktop. MATE not

only brings a traditional user experience but also complements the foundational strength of an LFS build with its blend of performance and aesthetics.

This paper delves into the step-by-step process of building an LFS system, followed by the integration of the MATE desktop environment. Through this exploration, we aim to showcase the synergy between the robustness of a handcrafted Linux system and the elegance of a tailored graphical interface.

Important: *We are not providing a tutorial on how to build LFS, if you want to complete the build yourself we highly recommend following the documentation provided by the LFS book. We are going to be explaining what each component of the process of building LFS does, and providing information on some usefull commands.*

1 Preparing for the Build

Preparing host system

The initial stage in constructing an LFS system involves setting up the necessary environment and ensuring all prerequisites are met. We are assuming your current host system is a Linux distro, or even a Linux virtual machine. We ran the whole LFS project in EndeavourOS and GentOS, you don't need to be on these particular distros, but need a Linux distro which will include the following packages:

- Bash-3.2
- Binutils-2.25
- Bison-2.7
- Bzip2-1.0.4
- Coreutils-6.9
- Diffutils-2.8.1
- Findutils-4.2.31
- Gawk-4.0.1
- GCC-6.2
- Glibc-2.11
- Grep-2.5.1a
- Gzip-1.3.12
- Linux Kernel-3.2
- M4-1.4.10
- Make-4.0

- Patch-2.5.4
- Perl-5.8.8
- Python-3.4
- Sed-4.1.5
- Tar-1.22
- Texinfo-4.7
- Xz-5.0.0

Important: *The version of the packages mentioned is the last known configuration to have worked together. This doesn't mean that this are the only versions that will work, but a good metric would be to have the version mentioned or above.*

LFS is recommended to be built in one session, this is where having the host system on a virtual machine helps a lot, as we can save snapshots of current states and if the build fails in the future, we can just fallback to a saved state. For the people building it on your main OS... *Good luck*!!!

LFS, like many operating systems, is typically installed on a dedicated partition, with a recommended size of 10 GB for a minimal setup and 30 GB for primary systems to allow for growth and additional software. While compiling packages, a significant amount of space is used, which is later reclaimed. Due to potential RAM constraints, using a small disk partition as swap space is advised, and the LFS system can share the host's swap partition. Disk partitioning tools like cfdisk or fdisk can assist in this process. If your host system has less than 4GB of RAM, you might need to reconsider building LFS, as it will most likely present you a very pretty blue screen.

System partitioning is a subjective topic, but a 20 GB root LFS partition is suggested as a balance. The swap partition's size is often debated, but it should be at least as large as the RAM for hibernation purposes. Optional partitions, such as /boot, /home, /usr, /opt, /tmp, and /usr/src, offer various benefits and can be tailored based on user needs. All partitions intended for automatic boot mounting should be specified in /etc/fstab.

Commands for Creating a Partition

```
mkfs -v -t ext4 /dev/<xxx>
```

Replace <xxx> with the name of the LFS partition.

If you are using an existing swap partition, there is no need to format it. If a new swap partition was created, it will need to be initialized with this command:

```
mkswap /dev/<yyy>
```

Replace <yyy> with the name of the swap partition.

Exporting the LFS Variable

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

After creating the partitions, we now mount them to our system to make them available and accessible.

```
mkdir -pv $LFS
mount -v -t ext4 /dev/<xxx> $LFS
```

Replace <xxx> with the designation of the LFS partition.

If using multiple partitions for LFS (e.g., one for / and another for /home), mount them using:

```
mkdir -pv $LFS
mount -v -t ext4 /dev/<xxx> $LFS
mkdir -v $LFS/home
mount -v -t ext4 /dev/<yyy> $LFS/home
```

Replace <xxx> and <yyy> with the appropriate partition names.

If you made a swap partition make it available with the swapon command:

```
/sbin/swapon -v /dev/<zzz>
```

Replace <zzz> with the name of the swap partition.

Packages

The fastest way to download the packages at the versions known to work is using the wget-list provided by LFS. First, create a sources directory on our LFS partition as root.

```
mkdir -v $LFS/sources
chmod -v a+wt $LFS/sources
```

After downloading the LFS wget-list, download all the files with:

```
wget --input-file=wget-list --continue --directory-prefix=$LFS/sources
```

Additionally, starting with LFS-7.0, there is a separate file, md5sums, which can be used to verify that all the correct packages are available before proceeding. Place that file in $LFS/sources and run:

```
pushd $LFS/sources
md5sum -c md5sums
popd
```

Final preparations

In this final section, we will finish setting up our environment for the proper installation of the packages. A set of directories in $LSF will be created. We will also explain the measurements of "SBU's", which are the way we we will make an estimated measure on how long it will take to build all of the required packages.

We start by creating the required directory layout by running the following as root:

```
mkdir -pv $LFS/{etc,var} $LFS/usr/{bin,lib,sbin}
for i in bin lib sbin; do
    ln -sv usr/$i $LFS/$i
done
case $(uname -m) in
    x86_64) mkdir -pv $LFS/lib64 ;;
esac
```

We will also create a directory where the process of the cross-compiler will take place on.

```
mkdir -pv $LFS/tools
```

We have to be very care full while using root, so for the following installations it is recommended to make a new user so we don't destroy the current system we are on.

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

We also grant this user full access to all the directories under $LFS.

```
chown -v lfs $LFS/{usr{,/*},lib,var,etc,bin,sbin,tools}
case $(uname -m) in
    x86_64) chown -v lfs $LFS/lib64 ;;
esac
```

We now create 2 startup profiles for the bash shell.

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

Now log in as the lsf user we created. And we will create the apropiate .bashrc file:

```
cat > ~/.bashrc << "EOF"
set +h
umask 022
LFS=/mnt/lfs
LC_ALL=POSIX
LFS_TGT=$(uname -m)-lfs-linux-gnu
PATH=/usr/bin
```

```
if [ ! -L /bin ]; then PATH=/bin:$PATH; fi
PATH=$LFS/tools/bin:$PATH
CONFIG_SITE=$LFS/usr/share/config.site
export LFS LC_ALL LFS_TGT PATH CONFIG_SITE
EOF
```

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

```
source ~/.bash_profile
```

SBUs

LFS can take a lot of time to build depending on the computing speed of your host system, with the largest package, Glibc, taking from 20 minutes to three days, depending on the system's speed. Instead of exact times, the Standard Build Unit (SBU) is used for reference. The first package, binutils, sets the SBU based on its compilation time. Subsequent packages are measured relative to this SBU. For instance, if binutils takes 10 minutes, a package with a time of 4.5 SBUs will take roughly 45 minutes. However, SBU estimates can vary significantly based on factors like the host system's GCC version.

For a faster time of compilation, we will export a make flag with the amount of cores we will use for the making of our packages. It is recommended to use all the cores you have availble in your system. This example provides 4 cores for the compilation.

```
export MAKEFLAGS='-j4'
```

Be wary of specifying more cores than your system has, as it will crash instantly while making the packages.

2 Building the LFS Cross Toolchain and Temporary Tools

Once the environment is prepared, the next step is to build the cross toolchain and temporary tools. These tools serve as a foundation for the subsequent stages and ensure that the building process is isolated from the host system.

This section is split into three phases: initially constructing a cross compiler along with its related libraries; next, utilizing this cross toolchain to create various utilities, ensuring they're separated from the host setup; and finally, transitioning into the chroot setting to enhance host separation and construct the remaining tools essential for the final system.

Compiling a Cross-Toolchain

Cross compiling is the process of building software on one system (the "host") such that it can run on a different system (the "target"). This approach is especially useful when the target system has limited resources or lacks a native compiler. For instance, developers

might use a powerful desktop machine to compile software for embedded devices or different operating systems. The cross compiler, a specialized tool, is designed to produce executables for the target system while running on the host system, taking into account the differences in architecture, system libraries, and other platform-specific characteristics.

To ensure separation from subsequent installations, programs compiled in this section are installed under the $LFS/tools directory, while libraries are placed in their final locations. It is important to install the packages in the following order:

- **Binutils-2.37 - Pass 1:** This package contains tools like a linker and an assembler. It's crucial to compile Binutils first, as other packages like Glibc and GCC test the available linker and assembler to determine feature enablement.
- **GCC-11.2.0 - Pass 1:** The GNU compiler collection, which includes the C and C++ compilers, is covered here. The chapter emphasizes the importance of ensuring that the basic functions of the new toolchain are working as expected.
- **Linux-5.13.12 API Headers:** These headers expose the kernel's API for Glibc's use. The chapter explains the process of extracting and installing these headers.
- **Glibc-2.34:** This package contains the main C library, which provides basic routines like memory allocation, file handling, arithmetic, and more. A sanity check is recommended after its installation to ensure the toolchain's proper functioning.
- **Libstdc++ from GCC-11.2.0, Pass 1:** This is the standard C++ library. Its installation was deferred during the gcc-pass1 build because it depends on glibc.

Cross Compiling Temporary Tools

Here we will start cross-compiling basic utilities using the built in cross-toolchain. The utilities we will be installing will reside in their final location, but cannot be used yet. Most of the task will still be done by the hosts own tools and utilities. This utilities, will become available when we enter the chroot environment.

We will start by compiling the following packages, and remember to log in as lfs user:

- **M4-1.4.19:** Contains a macro processor.
- **Ncurses-6.2:** Contains libraries for terminal-independent handling of character screens.
- **Bash-5.1.8:** Contains the Bourne-Again Shell.
- **Coreutils-8.32:** Contains utilities for showing and setting basic system characteristics.
- **Diffutils-3.8:** Contains programs that show differences between files or directories.
- **File-5.40:** Contains a utility for determining file types.
- **Findutils-4.8.0:** Contains programs to find files and maintain/search a database.

- **Gawk-5.1.0:** Contains programs for manipulating text files.
- **Grep-3.7:** Contains programs for searching through file contents.
- **Gzip-1.10:** Contains programs for compressing and decompressing files.
- **Make-4.3:** Contains a program for controlling the generation of executables from source files.
- **Patch-2.7.6:** Contains a program for modifying or creating files by applying a "patch".
- **Sed-4.8:** Contains a stream editor.
- **Tar-1.34:** Provides the ability to create tar archives and other archive manipulations.
- **Xz-5.2.5:** Contains programs for compressing and decompressing files using lzma and xz formats.
- **Binutils-2.37 - Pass 2:** Contains a linker, assembler, and other tools for handling object files.
- **GCC-11.2.0 - Pass 2:** Contains the GNU compiler collection, including C and C++ compilers.

Entering Chroot and building Additional Temporary Tools

This section focuses on constructing the final components of the temporary system: tools required by the build machinery of various packages. A "chroot" environment is used, which is isolated from the host operating system, except for the running kernel. This means, that when we are logged in via chroot, we are now using all the tools that we previously installed in our host system. Although communication with the running kernel is established through Virtual Kernel File Systems.

From this point onward, we will issue every command on root, with the LFS variable set. After entering chroot, all commands are run as root without access to the host OS.

At the moment, the entire directory structure in $LFS is under the ownership of the 'lfs' user, which is exclusive to the host system. Retaining the directories and files in $LFS in their current state means they'll be associated with a user ID that doesn't have a matching account. This poses a risk, as any future user account might acquire this user ID, gaining control over all $LFS files, making them vulnerable to potential harmful alterations.

The easy fix for this is changing the ownership of all of lfs directories ($LFS/*) to root by running the following command.

```
chown -R root:root $LFS/{usr,lib,var,etc,bin,sbin,tools}
case $(uname -m) in
    x86_64) chown -R root:root $LFS/lib64 ;;
esac
```

Preparing Virtual Kernel File Systems

All of the communication we will be having with the kernel from this point onward is virtual, which basically means that the kernel will be residing on memory, not on disk. Which means that any unexpected shutdown from this point onward could be catastrophic. For this is necessary the creation of a directory where these virtual file systems will be mounted on:

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

Whenever we boot our LFS system, one of the first steps will always be to mount whatever is on /dev to memory, and we instantiate the udev daemon. This daemon is in charge of administering devices, and will have the permissions necessary to create and modify device nodes.

The following commands will take charge of populating our $LFS/dev directory:

```
mount -v --bind /dev $LFS/dev
mount -v --bind /dev/pts $LFS/dev/pts
mount -vt proc proc $LFS/proc
mount -vt sysfs sysfs $LFS/sys
mount -vt tmpfs tmpfs $LFS/run
```

On certain host systems, /dev/shm is set up as a symbolic link that points to /run/shm. Since the /run tmpfs has already been mounted, you only need to create a directory in this scenario. However, on other host systems, /dev/shm serves as a mount point for a tmpfs. In such cases, mounting /dev as described earlier will result in /dev/shm being created as a directory within the chroot environment. Under these circumstances, it becomes necessary to explicitly mount a tmpfs.

```
if [ -h $LFS/dev/shm ]; then
    mkdir -pv $LFS/$(readlink $LFS/dev/shm)
else
    mount -t tmpfs -o nosuid,nodev tmpfs $LFS/dev/shm
fi
```

Entering the Chroot Environment

Now that the necessary tool for installation are available, we can access our system from the chroot environment as root user. From this point on we will build the rest of the tools. We will also be using this environment for the installation of the final build of our system.

The command to access the chroot environment is as follows:

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

From this point on we will be restricted to the LFS system. So there is no need to use the LFS variable as this is the root of the directory we will be in. Avid readers might have noticed that /tools/bin is not in our root, this is because we will not be using the cross tool chain anymore, as we are inside of the LFS system there is no need for this.

It is important to note that if we ever leave the chroot environment, we will have to remount the virtual kernel.

We will now create the full directory structure of our LFS system:

```
mkdir -pv /{boot,home,mnt,opt,srv}

mkdir -pv /etc/{opt,sysconfig}
mkdir -pv /lib/firmware
mkdir -pv /media/{floppy,cdrom}
mkdir -pv /usr/{,local/}{include,src}
mkdir -pv /usr/local/{bin,lib,sbin}
mkdir -pv /usr/{,local/}share/{color,dict,doc,info,locale,man}
mkdir -pv /usr/{,local/}share/{misc,terminfo,zoneinfo}
mkdir -pv /usr/{,local/}share/man/man{1..8}
mkdir -pv /var/{cache,local,log,mail,opt,spool}
mkdir -pv /var/lib/{color,misc,locate}

ln -sfv /run /var/run
ln -sfv /run/lock /var/lock

install -dv -m 0750 /root
install -dv -m 1777 /tmp /var/tmp
```

Directories are, by default, created with permission mode 755, but this is not desirable everywhere. In the commands above, two changes are made—one to the home directory of user root, and another to the directories for temporary files. The first mode change ensures that not just anybody can enter the /root directory—just like a normal user would do with his or her own home directory. The second mode change makes sure that any user can write to the /tmp and / var/tmp directories, but cannot remove another user's files from them. The latter is prohibited by the so-called "sticky bit," the highest bit (1) in the 1777 bit mask.

In order for user root to be able to login and for the name "root" to be recognized, there must 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
bin:x:1:1:bin:/dev/null:/usr/bin/false
daemon:x:6:6:Daemon User:/dev/null:/usr/bin/false
messagebus:x:18:18:D-Bus Message Daemon User:/run/dbus:/usr/bin/false
uuidd:x:80:80:UUID Generation Daemon User:/dev/null:/usr/bin/false
nobody:x:65534:65534:Unprivileged User:/dev/null:/usr/bin/false
EOF
```

The actual password for root will be set later.

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

```
cat > /etc/group << "EOF"
root:x:0:
bin:x:1:daemon
sys:x:2:
kmem:x:3:
tape:x:4:
tty:x:5:
daemon:x:6:
floppy:x:7:
disk:x:8:
lp:x:9:
dialout:x:10:
audio:x:11:
video:x:12:
utmp:x:13:
usb:x:14:
cdrom:x:15:
adm:x:16:
messagebus:x:18:
input:x:24:
mail:x:34:
kvm:x:61:
uuidd:x:80:
wheel:x:97:
users:x:999:
nogroup:x:65534:
EOF
```

The groups created do not follow a universal standard, being shaped partly by Udev configuration requirements and partly by conventions from various Linux distributions. Some test suites also depend on certain users or groups. The Linux Standard Base (LSB) only strongly suggests having a root group with a Group ID (GID) of 0 and a bin group with a GID of 1. The GID of 5 is commonly used for the tty group and in /etc/fstab for the devpts file system. However, system administrators are free to choose other group names and GIDs, as programs should rely on group names, not GID numbers.

We can now remove the "I have no name!" prompt from our shell, by running the following:

```
exec /usr/bin/bash --login
```

Lastly, with the following commands we can give access so the system keeps the logs of who was logged into our system and when:

```
touch /var/log/{btmp,lastlog,faillog,wtmp}
```

```
chgrp -v utmp /var/log/lastlog
chmod -v 664 /var/log/lastlog
chmod -v 600 /var/log/btmp
```

Now you must install the following packages in the indicated order, this is a brief description of what each of them do:

- **Gettext-0.21:** Provides utilities for internationalization and localization, enabling programs to output messages in the user's native language.
- **Bison-3.7.6:** Contains a parser generator, a tool that converts a sequence of characters into a structured format.
- **Perl-5.34.0:** Includes the Practical Extraction and Report Language, a versatile scripting language used for various tasks.
- **Python-3.9.6:** Provides the Python development environment, useful for object-oriented programming, scripting, prototyping, and application development.
- **Texinfo-6.8:** Contains programs for reading, writing, and converting info pages, a common format for software documentation.
- **Util-linux-2.37.2:** Includes miscellaneous utility programs that provide a variety of functions not included in the core utilities.

Its recommended (but not necessary) to create a backup of the current state of our LFS system. For this, unmount the virtual kernel, exit the chroot environment, and zip the contents of $LFS/HOME. Restoring it is pretty straightforward too: delete all the files, and unzip the backup we did.

3 Building the LFS System

With the cross toolchain and temporary tools in place, the final stage involves building the LFS system itself. This process transforms the temporary environment into a standalone Linux system, tailored to the user's specific needs.

All of the programs that we are going to compile we will be removing or disabling installation of most static libraries. This is done because the purpose of the majority of static libraries has become obsolete in modern Linux systems. The only exceptions we will have are glibc and gcc, as the use of these static libraries still remains essentially for the build.

Then, we stumble upon the topic of package management. Package management has been widely neglected by LFS, as it takes all of the fun away! The purpose of the LFS journey, is to you yourself compile the packages you need of might want in your system. This, to learn what the system actually needs with each library we will build. A package manager will take a lot of control from us! So our chosen package manager will be our head. If you please, you very well (and easily) install any common package manager to aid you with the installations.

In this point of the book, the only thing left is to install all of the packages, this will be the most time consuming part of our build (as it includes glibc that takes 21 SBUs and gcc with 164 SBUs !!!). The remaining packages are the following:

- **Man-pages-5.13:** Contains manual pages for Linux, covering system calls, library routines, and other system documentation.
- **Iana-Etc-20210611:** Provides data for network services and protocols, translating protocol and service names to numbers.
- **Glibc-2.34:** The GNU C Library, fundamental for system's performance and functionality, providing core libraries for various system operations.
- **Zlib-1.2.11:** A data-compression library used for compression and decompression tasks within software applications.
- **Bzip2-1.0.8:** A freely available, patent-free, high-quality data compressor that compresses files to within 10% to 15% of the achievable limit.
- **Xz-5.2.5:** A data compression tool and library which aims at providing better compression rates than gzip and bzip2.
- **Zstd-1.5.0:** A fast lossless compression algorithm, targeting real-time compression scenarios at zlib-level and better compression ratios.
- **File-5.40:** A utility for determining file types.
- **Readline-8.1:** A library that provides command-line editing and history capabilities.
- **M4-1.4.19:** A macro processor that's used as a front-end to a compiler.
- **Bc-5.0.0:** An arbitrary precision numeric processing language.
- **Flex-2.6.4:** A tool for generating text scanning programs.
- **Tcl-8.6.11:** A scripting language designed for embedding in applications and for writing quick-and-dirty scripts.
- **Expect-5.45.4:** A tool for automating interactive applications based on a script following the "Don't check, just do" principle.
- **DejaGNU-1.6.3:** A framework for testing other programs, its purpose is to provide a single front end for all tests.
- **Binutils-2.37:** A collection of binary tools for handling executable files, object code, libraries, profile data, and assembly source code.
- **GMP-6.2.1:** A library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers.
- **MPFR-4.1.0:** A library for multiple-precision floating-point computations with correct rounding.

- **MPC-1.2.1:** A library for the arithmetic of complex numbers with arbitrarily high precision.
- **Attr-2.5.1:** Utilities for managing file system extended attributes.
- **Acl-2.3.1:** Utilities for managing POSIX Access Control Lists.
- **Libcap-2.53:** A library for getting and setting POSIX.1e capabilities.
- **Shadow-4.9:** Utilities for managing accounts and shadow password files.
- **GCC-11.2.0:** The GNU Compiler Collection, which includes front ends for C, C++, Objective-C, Fortran, Ada, and Go, as well as libraries for these languages.
- **Pkg-config-0.29.2:** A script and set of conventions for determining the compile and link flags that you should use to compile and link system libraries.
- **Ncurses-6.2:** A library for writing text-based user interfaces.
- **Sed-4.8:** A stream editor for filtering and transforming text.
- **Psmisc-23.4:** Utilities for managing processes on your system.
- **Gettext-0.21:** Internationalization and localization system for easier string translations in programs.
- **Bison-3.7.6:** A parser generator that is compatible with YACC.
- **Grep-3.7:** A utility to search for text using patterns.
- **Bash-5.1.8:** The GNU Bourne Again shell, which provides a command line interface for various operating systems.
- **Libtool-2.4.6:** A generic library support script that abstracts the library building process across many different platforms.
- **GDBM-1.20:** The GNU Database Manager, a library of database functions that use extensible hashing.
- **Gperf-3.1:** A perfect hash function generator.
- **Expat-2.4.1:** An XML parser library written in C.
- **Inetutils-2.1:** A collection of common network programs.
- **Less-590:** A program for viewing text files, one screenful at a time.
- **Perl-5.34.0:** A high-level programming language suitable for a wide range of tasks.
- **XML::Parser-2.46:** A Perl module for parsing XML documents.
- **Intltool-0.51.0:** A set of tools for translating the contents of data files using gettext.
- **Autoconf-2.71:** An extensible package of M4 macros that produce shell scripts to automatically configure software source code packages.

- **Automake-1.16.4:** A tool for automatically generating Makefile.in files compliant with the GNU Coding Standards.
- **mod-29:** Tools and libraries for working with kernel modules.
- **Libelf from Elfutils-0.185:** A collection of utilities and libraries for handling the ELF (Executable and Linking Format) file format.
- **Libffi-3.4.2:** A portable, high-level programming interface to various calling conventions.
- **OpenSSL-1.1.1l:** A toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols.
- **Python-3.9.6:** A high-level, dynamically typed, and garbage-collected programming language suitable for a wide range of tasks.
- **Ninja-1.10.2:** A small build system focused on speed.
- **Meson-0.59.1:** An open-source build system meant to be both extremely fast and user-friendly.
- **Coreutils-8.32:** The basic file, shell, and text manipulation utilities of the GNU operating system.
- **Check-0.15.2:** A unit testing framework for C.
- **Diffutils-3.8:** Utilities for comparing and contrasting files.
- **Gawk-5.1.0:** The GNU version of the AWK text processing utility.
- **Findutils-4.8.0:** Utilities for finding files meeting specified criteria and performing various actions on the files which are found.
- **Groff-1.22.4:** A typesetting system that reads plain text mixed with formatting commands and produces formatted output.
- **GRUB-2.06:** The GRand Unified Bootloader, a bootloader capable of booting a wide variety of operating systems.
- **Gzip-1.10:** A compression utility designed to be a replacement for compress.
- **IPRoute2-5.13.0:** Utilities for controlling TCP/IP networking and traffic control in Linux.
- **Kbd-2.4.0:** Utilities for configuring and managing keyboard input devices.
- **Libpipeline-1.5.3:** A library for manipulating pipelines of subprocesses in a flexible and convenient way.
- **Make-4.3:** A utility that automatically builds executable programs and libraries from source code.
- **Patch-2.7.6:** A utility for applying changes to files based on differences listed in a patch file.

- **Tar-1.34:** A utility for archiving files.
- **Texinfo-6.8:** A documentation system for on-line information and printed output.
- **Vim-8.2.3337:** An improved version of the vi editor, an interactive text editor.
- **Eudev-3.2.10:** A fork of systemd providing the udev device manager.
- **Man-DB-2.9.4:** A database-driven man page indexer and a replacement for the traditional man command.
- **Procps-ng-3.3.17:** Utilities for browsing the /proc filesystem.
- **Util-linux-2.37.2:** A suite of essential utilities for any Linux system.
- **E2fsprogs-1.46.4:** Utilities for handling the ext2, ext3, and ext4 filesystems.
- **Sysklogd-1.5.1:** Utilities for system logging.
- **Sysvinit-2.99:** A System V-style init program for booting, initializing, and shutting down the system.

Wrapping up

We now have everything installed on our OS, and we just need a way to boot it. The journey through configuring and setting up our Linux From Scratch system has been intricate, but we are now at the final stages where we make everything come together for a smooth operation.

In the realm of system configuration, it is crucial to ensure that every component and service within our LFS installation is properly set up. This phase lays down the groundwork for a stable and efficient system operation, covering a wide array of configurations from boot scripts to device management. The LFS-Bootscripts package is a cornerstone in initializing our system, comprising scripts that manage the startup and shutdown of the LFS system. To install this package, simply execute:

```
make install
```

This package is pivotal as it ensures all necessary services and configurations are in place, ready for the system to operate seamlessly.

Transitioning to device and module handling, LFS adopts a dynamic approach, utilizing udev and sysfs for efficient device management. This ensures that only detected devices by the kernel are assigned device nodes, optimizing system resources and enhancing performance. Device management is a crucial aspect of system stability and performance. For network devices, LFS employs a persistent naming scheme, ensuring consistency across system reboots. Additionally, for devices like CD-ROMs and DVD-ROMs, udev provides scripts to create stable symlinks, with options to choose between “by-path” or “by-id” modes depending on the specific hardware setup.

As we approach the final stages of our LFS installation, making the system bootable is our primary focus. The /etc/fstab file plays a vital role in this process, as it is responsible for mounting filesystems at boot time. Proper configuration of this file is imperative to ensure all necessary filesystems are mounted correctly, paving the way for a successful boot process. Here is an example of what the /etc/fstab file might look like:

```
# Begin /etc/fstab

# file system  mount-point  type     options             dump  fsck
#                                                              order

/dev/sda2      /            ext4     defaults            1     1
/dev/sda1      /boot        ext2     defaults            1     2
/dev/sda3      swap         swap     pri=1               0     0
proc           /proc        proc     nosuid,noexec,nodev 0     0
sysfs          /sys         sysfs    nosuid,noexec,nodev 0     0
devpts         /dev/pts     devpts   gid=5,mode=620      0     0
tmpfs          /run         tmpfs    defaults            0     0
devtmpfs       /dev         devtmpfs mode=0755,nosuid    0     0

# End /etc/fstab
```

Ensuring that the Linux-5.13.12 kernel is correctly configured is another crucial step. Attention to detail is required, especially when it comes to setting up unique settings or modules specific to your hardware. GRUB comes into play as we set up the boot process. Installation and configuration of GRUB are essential to ensure that the system boots correctly, selecting the LFS kernel.

Congratulations are in order as we reach the conclusion of our LFS installation guide. With everything in place, it's now time to reboot your system and dive into your newly installed OS. Ensure that the installation media is removed, and your BIOS is set to boot from the hard drive. A simple 'reboot' command is all it takes to start exploring your new LFS system.

```
reboot
```

Installing MATE

We finally have our LFS system up and running, but its not over. Here we will give you a quick guide on how to install MATE to your newly installed LFS system.

Prerequisites

Before you start, ensure that your LFS system is properly configured and that you have a working internet connection. You will also need to have the X Window System installed. If you haven't installed Xorg yet, you should do that first.

Step 1: Install Required Dependencies. MATE has a number of dependencies that need to be installed before you can install the desktop environment itself. Some of the main dependencies include:

- **GTK+3:** The GIMP Toolkit is used for creating graphical user interfaces.
- **GLib:** A low-level core library that forms the basis for projects such as GTK+ and GNOME.
- **D-Bus:** A message bus system that provides an easy way for inter-process communication.
- **libmatekbd:** A library to manage keyboard configuration.
- **libmatemixer:** An audio mixer library for MATE Desktop.
- **MATE Common:** A set of scripts and m4/autoconf macros that are used to build packages from the MATE project.

You can download the source code for these dependencies from their respective official websites or from the package manager if you have one installed.

Step 2: Install MATE Libraries. Once the dependencies are installed, you can proceed to install the MATE libraries. Some of the libraries you need to install include:

- **libmateweather:** Provides weather information for MATE.
- **mate-icon-theme:** The icon theme for the MATE desktop.
- **mate-polkit:** A MATE-specific wrapper around polkit.
- **mate-settings-daemon:** Manages the configuration settings on the MATE desktop.

Download the source code for these libraries and follow the installation instructions provided in their README or INSTALL files.

Step 3: Install MATE Applications. With the libraries in place, you can now install MATE applications. Some of the applications you might want to install include:

- **Caja:** The file manager for the MATE desktop.
- **Pluma:** A text editor for MATE.
- **Eye of MATE:** An image viewer.
- **MATE Terminal:** The terminal emulator for MATE.

Again, download the source code for these applications and follow the installation instructions.

Step 4: Install MATE Desktop Environment. Finally, you can install the MATE desktop environment itself. Download the source code for MATE from the official MATE website or from your package manager, and follow the installation instructions.

Step 5: Configure Xorg for MATE. Edit your /.xinitrc file to start MATE automatically when you start Xorg:

```
exec mate-session
```

Step 6: Start MATE. You can now start MATE by running the startx command:

```
startx
```

This should launch the MATE desktop environment. If everything is installed correctly, you should see the MATE desktop and be able to interact with it using your mouse and keyboard.

Conclusions

The complex and long installation process of LFS and MATE has culminated in a profound educational experience, meticulously unraveling the complexities inherent in Linux systems and their constituent components. Throughout this endeavor, we have learnt a lot spanning various domains, including but not limited to system compilation, package management, kernel configuration, and the intricacies of desktop environment installation. The process demanded a rigorous compilation and installation of each package, a prerequisite for ensuring the stability and functionality of the system. The subsequent integration of the MATE desktop environment further expanded our understanding, delving into the complexities associated with graphical user interfaces in Linux-based systems. This hands-on, immersive approach has significantly enhanced our technical acumen, providing a holistic understanding of the Linux operating system's architecture. The proficiency gained through this meticulous process is invaluable, establishing a robust foundation for advanced Linux administration and system customization. The journey, albeit challenging, has been immensely rewarding, offering a unique perspective and a comprehensive insight into the Linux operating system's inner workings.

Acknowledgments

We extent a hearty thanks to Professor Jose Luis Montoya Pareja, for dictating the Operating Systems course in an excellent manner and giving us the inspiration in developing our own OS. He held our hand through the whole process and gave us the relevant feedback every time we ran into a wall. To you, our thanks for making us fall more in love with computer architecture and operating systems.

We would also love to thank Professor Edison Valencia Diaz, as he was one to guide us into using an existing kernel. It may seem silly, but our first approach was to build our own kernel in a x86 processor.

Biography

The main books we used as support are:

- Arpaci-Dusseau, R.H. and Arpaci-Dusseau, A.C. (2018) Operating systems: Three easy pieces. Madison, WI: Arpaci-Dusseau Books.
- Beekmans, G. (2023) Linux from scratch: Version 11.0 Lodi (California): Clearly Open.