DIY: Build a Custom Minimal Linux Distribution from Source

by Petros Koutoupis on July 3, 2018

Follow along with this step-by-step guide to build your own distribution from source and learn how it installs, loads and runs.

When working with Linux, you easily can download any of the most common distributions to install and configure—be it Ubuntu, Debian, Fedora, OpenSUSE or something entirely different. And although you should give several distributions a spin, building your own custom, minimal Linux distribution is also a beneficial and wonderful learning exercise.

When I say "build a custom and minimal Linux distribution", I mean from *source* packages—that is, start with a cross-compiling toolchain and then build a target image to install on a physical or virtual hard disk drive (HDD).

So, when I think of the ultimate Do-It-Yourself (DIY) guide related to Linux, it's got to be exactly this: *building a Linux distribution from source*. The entire process will take at least a couple hours on a decently powered host machine.

If you follow along with this exercise, you'll learn what it takes to build a custom distribution, and you'll also learn how that distribution installs, loads and runs. You can run this exercise on either a physical or virtual machine.

I'd be lying if I said that this process wasn't partly inspired by the wonderful Linux From Scratch (LFS) project. The LFS project proved to be an essential tool in my understanding of how a standard Linux operating system is built and functions. Using a similar philosophy, I hope to instill some of the same wisdom to you, the reader, if you'd like to follow along.

Terms

- **Host:** the *host* signifies the very machine on which you'll be doing the vast majority of the work, including cross compilation and installation of the target image.
- **Target:** the *target* is the final cross-compiled operating system that you'll be building from source packages. It'll be built using the cross compiler on the host

machine.
- **Cross compiler:** you'll be building and using a *cross compiler* to create the *target* image on the *host* machine. A cross compiler is built to run on a host machine, but it's used to compile for a target architecture or microprocessor that isn't compatible with the host machine.

Prerequisites and Tools

To continue with this tutorial, you'll need to have GCC, make, ncurses, Perl and grub tools (specifically grub-install) installed on the host machine.

In order to build anything, you'll also need to download and build all the packages for the cross compiler and the target image. I'm using the following open-source packages and versions for this tutorial:

- binutils-2.30.tar.xz
- busybox-1.28.3.tar.bz2
- clfs-embedded-bootscripts-1.0-pre5.tar.bz2
- gcc-7.3.0.tar.xz
- glibc-2.27.tar.xz
- gmp-6.1.2.tar.bz2
- linux-4.16.3.tar.xz
- mpc-1.1.0.tar.gz
- mpfr-4.0.1.tar.xz
- zlib-1.2.11.tar.gz

Configuring the Environment

Before beginning this process, you need to configure the build environment. First, turn on Bash hash functions:

```
$ set +h
```

Make sure that newly created files/directories are writable only by the owner (for example, the currently logged in user account):

```
$ umask 022
```

You'll use your home directory as the main build directory. (this isn't a requirement). This is where the cross-compilation toolchain and target image will be installed and put into the lj-os subdirectory. If you prefer to install it elsewhere, make the adjustment to the code section below:

```
$ export LJOS=~/lj-os
$ mkdir -pv ${LJOS}
```

Finally, export some remaining variables:

```
$ export LC_ALL=POSIX
$ export PATH=${LJOS}/cross-tools/bin:/bin:/usr/bin
```

After setting the above environment variables, create the target image's filesystem hierarchy:

```
$ mkdir -pv ${LJOS}/{bin,boot{,grub},dev,{etc/,}opt,home,
↪lib/{firmware,modules},lib64,mnt}
$ mkdir -pv ${LJOS}/{proc,media/{floppy,cdrom},sbin,srv,sys}
$ mkdir -pv ${LJOS}/var/{lock,log,mail,run,spool}
$ mkdir -pv ${LJOS}/var/{opt,cache,lib/{misc,locate},local}
$ install -dv -m 0750 ${LJOS}/root
$ install -dv -m 1777 ${LJOS}{/var,}/tmp
$ install -dv ${LJOS}/etc/init.d
$ mkdir -pv ${LJOS}/usr/{,local/}{bin,include,lib{,64},sbin,sr
c}
$ mkdir -pv ${LJOS}/usr/{,local/}share/{doc,info,locale,man}
$ mkdir -pv ${LJOS}/usr/{,local/}share/{misc,terminfo,zoneinfo}
$ mkdir -pv ${LJOS}/usr/{,local/}share/man/man{1,2,3,4,5,6,7,8}
$ for dir in ${LJOS}/usr{,/local}; do
     ln -sv share/{man,doc,info} ${dir}
   done
```

This directory tree is based on the Filesystem Hierarchy Standard (FHS), which is defined and hosted by the Linux Foundation:

Create the directory for a cross-compilation toolchain:

```
$ install -dv ${LJOS}/cross-tools{,/bin}
```

Use a symlink to /proc/mounts to maintain a list of mounted filesystems properly in the /etc/mtab file:

```
$ ln -svf ../proc/mounts ${LJOS}/etc/mtab
```

Then create the /etc/passwd file, listing the root user account (note: for now, you won't be setting the account password; you'll do that after booting up into the target image for the first time):

```
$ cat > ${LJOS}/etc/passwd << "EOF"
root::0:0:root:/root:/bin/ash
EOF
```

Create the /etc/group file with the following command:

```
$ cat > ${LJOS}/etc/group << "EOF"
root:x:0:
bin:x:1:
sys:x:2:
kmem:x:3:
tty:x:4:
daemon:x:6:
disk:x:8:
dialout:x:10:
video:x:12:
utmp:x:13:
usb:x:14:
EOF
```

The target system's /etc/fstab:

```
$ cat > ${LJOS}/etc/fstab << "EOF"
# file system  mount-point  type   options          dump  fsck
#                                                         order

rootfs          /               auto    defaults        1
1
proc            /proc           proc    defaults        0
0
sysfs           /sys            sysfs   defaults        0
0
devpts          /dev/pts        devpts  gid=4,mode=620  0
0
tmpfs           /dev/shm        tmpfs   defaults        0
0
EOF
```

The target system's /etc/profile to be used by the Almquist shell (ash) once the user is logged in to the target machine:

```
$ cat > ${LJOS}/etc/profile << "EOF"
export PATH=/bin:/usr/bin

if [ `id -u` -eq 0 ] ; then
        PATH=/bin:/sbin:/usr/bin:/usr/sbin
        unset HISTFILE
fi


# Set up some environment variables.
export USER=`id -un`
export LOGNAME=$USER
export HOSTNAME=`/bin/hostname`
export HISTSIZE=1000
export HISTFILESIZE=1000
export PAGER='/bin/more '
export EDITOR='/bin/vi'
EOF
```

The target machine's hostname (you can change this any time):

```
$ echo "ljos-test" > ${LJOS}/etc/HOSTNAME
```

And, /etc/issue, which will be displayed prominently at the login prompt:

```
$ cat > ${LJOS}/etc/issue<< "EOF"
Linux Journal OS 0.1a
Kernel \r on an \m

EOF
```

You won't use systemd here (this wasn't a political decision; it's due to convenience and for simplicity's sake). Instead, you'll use the basic `init` process provided by BusyBox. This requires that you define an /etc/inittab file:

```
$ cat > ${LJOS}/etc/inittab<< "EOF"
::sysinit:/etc/rc.d/startup

tty1::respawn:/sbin/getty 38400 tty1
tty2::respawn:/sbin/getty 38400 tty2
tty3::respawn:/sbin/getty 38400 tty3
tty4::respawn:/sbin/getty 38400 tty4
tty5::respawn:/sbin/getty 38400 tty5
tty6::respawn:/sbin/getty 38400 tty6

::shutdown:/etc/rc.d/shutdown
::ctrlaltdel:/sbin/reboot
EOF
```

Also as a result of leveraging BusyBox to simplify some of the most common Linux system functionality, you'll use `mdev` instead of `udev`, which requires you to define the following /etc/mdev.conf file:

```
$ cat > ${LJOS}/etc/mdev.conf<< "EOF"
# Devices:
# Syntax: %s %d:%d %s
# devices user:group mode

# null does already exist; therefore ownership has to
# be changed with command
null    root:root 0666  @chmod 666 $MDEV
zero    root:root 0666
grsec   root:root 0660
full    root:root 0666

random  root:root 0666
urandom root:root 0444
hwrandom root:root 0660

# console does already exist; therefore ownership has to
# be changed with command
console root:tty 0600 @mkdir -pm 755 fd && cd fd && for x
 ↪in 0 1 2 3 ; do ln -sf /proc/self/fd/$x $x; done

kmem    root:root 0640
mem     root:root 0640
port    root:root 0640
ptmx    root:tty 0666

# ram.*
ram([0-9]*)     root:disk 0660 >rd/%1
loop([0-9]+)    root:disk 0660 >loop/%1
sd[a-z].*       root:disk 0660 */lib/mdev/usbdisk_link
hd[a-z][0-9]*   root:disk 0660 */lib/mdev/ide_links

tty             root:tty 0666
tty[0-9]        root:root 0600
tty[0-9][0-9]   root:tty 0660
ttyO[0-9]*      root:tty 0660
pty.*           root:tty 0660
vcs[0-9]*       root:tty 0660
vcsa[0-9]*      root:tty 0660

ttyLTM[0-9]     root:dialout 0660 @ln -sf $MDEV modem
ttySHSF[0-9]    root:dialout 0660 @ln -sf $MDEV modem
```

```
slamr           root:dialout 0660 @ln -sf $MDEV slamr0
slusb           root:dialout 0660 @ln -sf $MDEV slusb0
fuse            root:root  0666

# misc stuff
agpgart         root:root 0660  >misc/
psaux           root:root 0660  >misc/
rtc             root:root 0664  >misc/

# input stuff
event[0-9]+     root:root 0640 =input/
ts[0-9]         root:root 0600 =input/

# v4l stuff
vbi[0-9]        root:video 0660 >v4l/
video[0-9]      root:video 0660 >v4l/

# load drivers for usb devices
usbdev[0-9].[0-9]       root:root 0660 */lib/mdev/usbdev
usbdev[0-9].[0-9]_.*    root:root 0660
EOF
```

You'll need to create a /boot/grub/grub.cfg for the GRUB bootloader that will be installed on the target machine's physical or virtual HDD (note: the kernel image defined in this file needs to reflect the image built and installed on the target machine):

```
$ cat > ${LJOS}/boot/grub/grub.cfg<< "EOF"

set default=0
set timeout=5

set root=(hd0,1)

menuentry "Linux Journal OS 0.1a" {
        linux   /boot/vmlinuz-4.16.3 root=/dev/sda1 ro quiet
}
EOF
```

Finally, initialize the log files and give them proper permissions:

```
$ touch ${LJOS}/var/run/utmp ${LJOS}/var/log/{btmp,lastlog,wtm
p}
$ chmod -v 664 ${LJOS}/var/run/utmp ${LJOS}/var/log/lastlog
```

Building the Cross Compiler

If you recall, the cross compiler is a toolchain of various compilation tools built for the system on which it's executing but designed to compile for an architecture or microprocessor that's not necessarily compatible with the system on which you're using it. In my environment, I'm running a 64-bit x86 architecture (x86-64) and will be cross compiling to a generic x86-64 target architecture. Sure, this section is somewhat redundant considering the environment I am running in, but the tutorial is designed to ensure that you are able to build for an x86-64 target, regardless of the machine type that you are using (for example, PowerPC, ARM, x86 and so on).

You never can be too sure with what is set in a currently running environment, which is why you'll unset the following C and C++ flags:

```
$ unset CFLAGS
$ unset CXXFLAGS
```

Next, define the most vital parts of the host/target variables needed to create the cross-compiler toolchain and target image:

```
$ export LJOS_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
$ export LJOS_TARGET=x86_64-unknown-linux-gnu
$ export LJOS_CPU=k8
$ export LJOS_ARCH=$(echo ${LJOS_TARGET} | sed -e
 ↪'s/-.*//' -e 's/i.86/i386/')
$ export LJOS_ENDIAN=little
```

Kernel Headers

The kernel's standard header files need to be installed for the cross compiler. Uncompress the kernel tarball and change into its directory. Then run:

```
$ make mrproper
$ make ARCH=${LJOS_ARCH} headers_check && \
make ARCH=${LJOS_ARCH} INSTALL_HDR_PATH=dest headers_install
$ cp -rv dest/include/* ${LJOS}/usr/include
```

Binutils

Binutils contains a linker, assembler and other tools needed to handle compiled object files. Uncompress the tarball. Then create the binutils-build directory and change into it:

```
$ mkdir binutils-build
$ cd binutils-build/
```

Then run:

```
$ ../binutils-2.30/configure --prefix=${LJOS}/cross-tools \
--target=${LJOS_TARGET} --with-sysroot=${LJOS} \
--disable-nls --enable-shared --disable-multilib
$ make configure-host && make
$ ln -sv lib ${LJOS}/cross-tools/lib64
$ make install
```

Copy over the following header file to the target's filesystem:

```
$ cp -v ../binutils-2.30/include/libiberty.h ${LJOS}/usr/includ
e
```

GCC (Static)

Before building the final cross-compiler toolchain, you first must build a statically compiled toolchain to build the C library (glibc) to which the final GCC cross compiler will link.

Uncompress the GCC tarball, and then uncompress the following packages and move them into the GCC root directory:

```
$ tar xjf gmp-6.1.2.tar.bz2
$ mv gmp-6.1.2 gcc-7.3.0/gmp
$ tar xJf mpfr-4.0.1.tar.xz
$ mv mpfr-4.0.1 gcc-7.3.0/mpfr
$ tar xzf mpc-1.1.0.tar.gz
$ mv mpc-1.1.0 gcc-7.3.0/mpc
```

Now create a gcc-static directory and change into it:

```
$ mkdir gcc-static
$ cd gcc-static/
```

Run the following commands:

```
$ AR=ar LDFLAGS="-Wl,-rpath,${LJOS}/cross-tools/lib" \
../gcc-7.3.0/configure --prefix=${LJOS}/cross-tools \
--build=${LJOS_HOST} --host=${LJOS_HOST} \
--target=${LJOS_TARGET} \
--with-sysroot=${LJOS}/target --disable-nls \
--disable-shared \
--with-mpfr-include=$(pwd)/../gcc-7.3.0/mpfr/src \
--with-mpfr-lib=$(pwd)/mpfr/src/.libs \
--without-headers --with-newlib --disable-decimal-float \
--disable-libgomp --disable-libmudflap --disable-libssp \
--disable-threads --enable-languages=c,c++ \
--disable-multilib --with-arch=${LJOS_CPU}
$ make all-gcc all-target-libgcc && \
make install-gcc install-target-libgcc
$ ln -vs libgcc.a `${LJOS_TARGET}-gcc -print-libgcc-file-name |
 ↪sed 's/libgcc/&_eh/'`
```

Do not delete these directories; you'll need to come back to them from the final version of GCC.

Glibc

Uncompress the glibc tarball. Then create the glibc-build directory and change into it:

```
$ mkdir glibc-build
$ cd glibc-build/
```

Configure the following build flags:

```
$ echo "libc_cv_forced_unwind=yes" > config.cache
$ echo "libc_cv_c_cleanup=yes" >> config.cache
$ echo "libc_cv_ssp=no" >> config.cache
$ echo "libc_cv_ssp_strong=no" >> config.cache
```

Then run:

```
$ BUILD_CC="gcc" CC="${LJOS_TARGET}-gcc" \
AR="${LJOS_TARGET}-ar" \
RANLIB="${LJOS_TARGET}-ranlib" CFLAGS="-O2" \
../glibc-2.27/configure --prefix=/usr \
--host=${LJOS_TARGET} --build=${LJOS_HOST} \
--disable-profile --enable-add-ons --with-tls \
--enable-kernel=2.6.32 --with-__thread \
--with-binutils=${LJOS}/cross-tools/bin \
--with-headers=${LJOS}/usr/include \
--cache-file=config.cache
$ make && make install_root=${LJOS}/ install
```

GCC (Final)

As I mentioned previously, you'll now build the final GCC cross compiler that will link to the C library built and installed in the previous step. Create the gcc-build directory and change into it:

```
$ mkdir gcc-build
$ cd gcc-build/
```

Then run:

```
$ AR=ar LDFLAGS="-Wl,-rpath,${LJOS}/cross-tools/lib" \
../gcc-7.3.0/configure --prefix=${LJOS}/cross-tools \
--build=${LJOS_HOST} --target=${LJOS_TARGET} \
--host=${LJOS_HOST} --with-sysroot=${LJOS} \
--disable-nls --enable-shared \
--enable-languages=c,c++ --enable-c99 \
--enable-long-long \
--with-mpfr-include=$(pwd)/../gcc-7.3.0/mpfr/src \
--with-mpfr-lib=$(pwd)/mpfr/src/.libs \
--disable-multilib --with-arch=${LJOS_CPU}
$ make && make install
$ cp -v ${LJOS}/cross-tools/${LJOS_TARGET}/lib64/
↪libgcc_s.so.1 ${LJOS}/lib64
```

Now that you've built the cross compiler, you need to adjust and export the following variables:

```
$ export CC="${LJOS_TARGET}-gcc"
$ export CXX="${LJOS_TARGET}-g++"
$ export CPP="${LJOS_TARGET}-gcc -E"
$ export AR="${LJOS_TARGET}-ar"
$ export AS="${LJOS_TARGET}-as"
$ export LD="${LJOS_TARGET}-ld"
$ export RANLIB="${LJOS_TARGET}-ranlib"
$ export READELF="${LJOS_TARGET}-readelf"
$ export STRIP="${LJOS_TARGET}-strip"
```

Building the Target Image

The hard part is now complete—you have the cross compiler. Now, let's focus on building the components that will be installed on the target image. This includes various libraries and utilities and, of course, the Linux kernel itself.

BusyBox

BusyBox is one of my all-time favorite open-source projects. The project advertises itself to be the Swiss Army knife of open-source utilities, and that's probably the best description one could give the project. BusyBox combines a large collection of tiny versions of the most commonly used Linux utilities into a single distributed package. Those tools range from common binaries, text editors and command-line shells to filesystem and networking utilities, process management tools and many more.

Uncompress the tarball and change into its directory. Then load the default compilation configuration template:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-" defconfig
```

The default configuration template will enable the compilation of a default defined set of utilities and libraries. You can enable/disable whatever you see fit by running `menuconfig`:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-" menuconfig
```

Compile and install the package:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-"
$ make CROSS_COMPILE="${LJOS_TARGET}-" \
CONFIG_PREFIX="${LJOS}" install
```

Install the following Perl script, as you'll need it for the kernel build below:

```
$ cp -v examples/depmod.pl ${LJOS}/cross-tools/bin
$ chmod 755 ${LJOS}/cross-tools/bin/depmod.pl
```

The Linux Kernel

Change into the kernel package directory and run the following to set the default x86-64 configuration template:

```
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}- x86_64_defconfig
```

This will define a minimum set of modules and settings for the compilation process. You most likely will need to make the proper adjustments for the target machine's environment. This includes enabling modules for storage and networking controllers and more. You can do that with the `menuconfig` option:

```
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}- menuconfig
```

For instance, I'm going to be running this target image in a VirtualBox virtual machine where it will rely on an Intel `e1000` networking module (defaulted in defconfig) and an LSI `mpt2sas` storage controller for the operating system drive. For the sake of simplicity, these modules are configured to be compiled statically into the kernel image—that is, set to `*` instead of `m`. *Be sure to review what's needed and enable it, or your target environment will not operate properly when booted.*

Compile and install the kernel:

```
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}-
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}- \
INSTALL_MOD_PATH=${LJOS} modules_install
```

You'll need to copy a few files into the /boot directory for GRUB:

```
$ cp -v arch/x86/boot/bzImage ${LJOS}/boot/vmlinuz-4.16.3
$ cp -v System.map ${LJOS}/boot/System.map-4.16.3
$ cp -v .config ${LJOS}/boot/config-4.16.3
```

Then run the previously installed Perl script provided by the BusyBox package:

```
$ ${LJOS}/cross-tools/bin/depmod.pl \
-F ${LJOS}/boot/System.map-4.16.3 \
-b ${LJOS}/lib/modules/4.16.3
```

The Bootscripts

The Cross Linux From Scratch (CLFS) project (a fork of the original LFS project) provides a wonderful set of bootscripts that I use here for simplicity's sake. Uncompress the package and change into its directory. Out of box, one of the package's makefiles contains a line that may not be compatible with your current working shell. Apply the following changes to the package's root Makefile to ensure that you don't experience any issues with package installation:

```
@@ -19,7 +19,9 @@ dist:
        rm -rf "dist/clfs-embedded-bootscripts-$(VERSION)"

 create-dirs:
-       install -d -m ${DIRMODE}
 ↪${EXTDIR}/rc.d/{init.d,start,stop}
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/init.d
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/start
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/stop

 install-bootscripts: create-dirs
        install -m ${CONFMODE} clfs/rc.d/init.d/functions
         ↪${EXTDIR}/rc.d/init.d/
```

Then run the following commands to install and configure the target environment appropriately:

```
$ make DESTDIR=${LJOS}/ install-bootscripts
$ ln -sv ../rc.d/startup ${LJOS}/etc/init.d/rcS
```

Zlib

Now you're at the very last package for this tutorial. Zlib isn't a requirement, but it serves as a great guide for other packages you may want to install for your environment. Feel free to skip this step if you'd rather format and configure the physical or virtual HDD.

Uncompress the Zlib tarball and change into its directory. Then configure, build and install the package:

```
$ sed -i 's/-O3/-Os/g' configure
$ ./configure --prefix=/usr --shared
$ make && make DESTDIR=${LJOS}/ install
```

Now, because some packages may look for Zlib libraries in the /lib directory instead of the /lib64 directory, apply the following changes:

```
$ mv -v ${LJOS}/usr/lib/libz.so.* ${LJOS}/lib
$ ln -svf ../../lib/libz.so.1 ${LJOS}/usr/lib/libz.so
$ ln -svf ../../lib/libz.so.1 ${LJOS}/usr/lib/libz.so.1
$ ln -svf ../lib/libz.so.1 ${LJOS}/lib64/libz.so.1
```

Installing the Target Image

All of the cross compilation is complete. Now you have everything you need to install the entire cross-compiled operating system to either a physical or virtual drive, but before doing that, let's not tamper with the original target build directory by making a copy of it:

```
$ cp -rf ${LJOS}/ ${LJOS}-copy
```

Use this copy for the remainder of this tutorial. Remove some of the now unneeded directories:

```
$ rm -rfv ${LJOS}-copy/cross-tools
$ rm -rfv ${LJOS}-copy/usr/src/*
```

Followed by the now unneeded statically compiled library files (if any):

```
$ FILES="$(ls ${LJOS}-copy/usr/lib64/*.a)"
$ for file in $FILES; do
> rm -f $file
> done
```

Now strip all debug symbols from the installed binaries. This will reduce overall file sizes and keep the target image's overall footprint to a minimum:

```
$ find ${LJOS}-copy/{,usr/}{bin,lib,sbin} -type f
 ↪-exec sudo strip --strip-debug '{}' ';'
$ find ${LJOS}-copy/{,usr/}lib64 -type f -exec sudo
 ↪strip --strip-debug '{}' ';'
```

Finally, change file ownerships and create the following nodes:

```
$ sudo chown -R root:root ${LJOS}-copy
$ sudo chgrp 13 ${LJOS}-copy/var/run/utmp
 ↪${LJOS}-copy/var/log/lastlog
$ sudo mknod -m 0666 ${LJOS}-copy/dev/null c 1 3
$ sudo mknod -m 0600 ${LJOS}-copy/dev/console c 5 1
$ sudo chmod 4755 ${LJOS}-copy/bin/busybox
```

Change into the target copy directory to create a tarball of the entire operating system image:

```
$ cd {LJOS}-copy/
$ sudo tar cfJ ../ljos-build-21April2018.tar.xz *
```

Notice how the target image is less than 60MB. You built that—a minimal Linux operating system that occupies less than 60MB of disk space:

```
$ sudo du -h|tail -n1
58M     .
```

And, that same operating system compresses to less than 20MB:

```
$ ls -lh ljos-build-21April2018.tar.xz
-rw-r--r-- 1 root root 18M Apr 21 15:31
 ↪ljos-build-21April2018.tar.xz
```

For the rest of this tutorial, you'll need a disk drive. It will need to enumerate as a traditional block device (in my case, it's /dev/sdd):

```
$ cat /proc/partitions |grep sdd
   8       48     256000 sdd
```

That block device will need to be partitioned. A single partition should suffice, and you can use any one of a number of partition utilities, including `fdisk` or `parted`. Once that partition is created and detected by the host system, format the partition with an ext4 filesystem, mount that partition to a staging area and change into that directory:

```
$ sudo mkfs.ext4 /dev/sdd1
$ sudo mkdir tmp
$ sudo mount /dev/sdd1 tmp/
$ cd tmp/
```

Uncompress the operating system tarball of the entire target operating system into the root of the staging directory:

```
$ sudo tar xJf ../ljos-build-21April2018.tar.xz
```

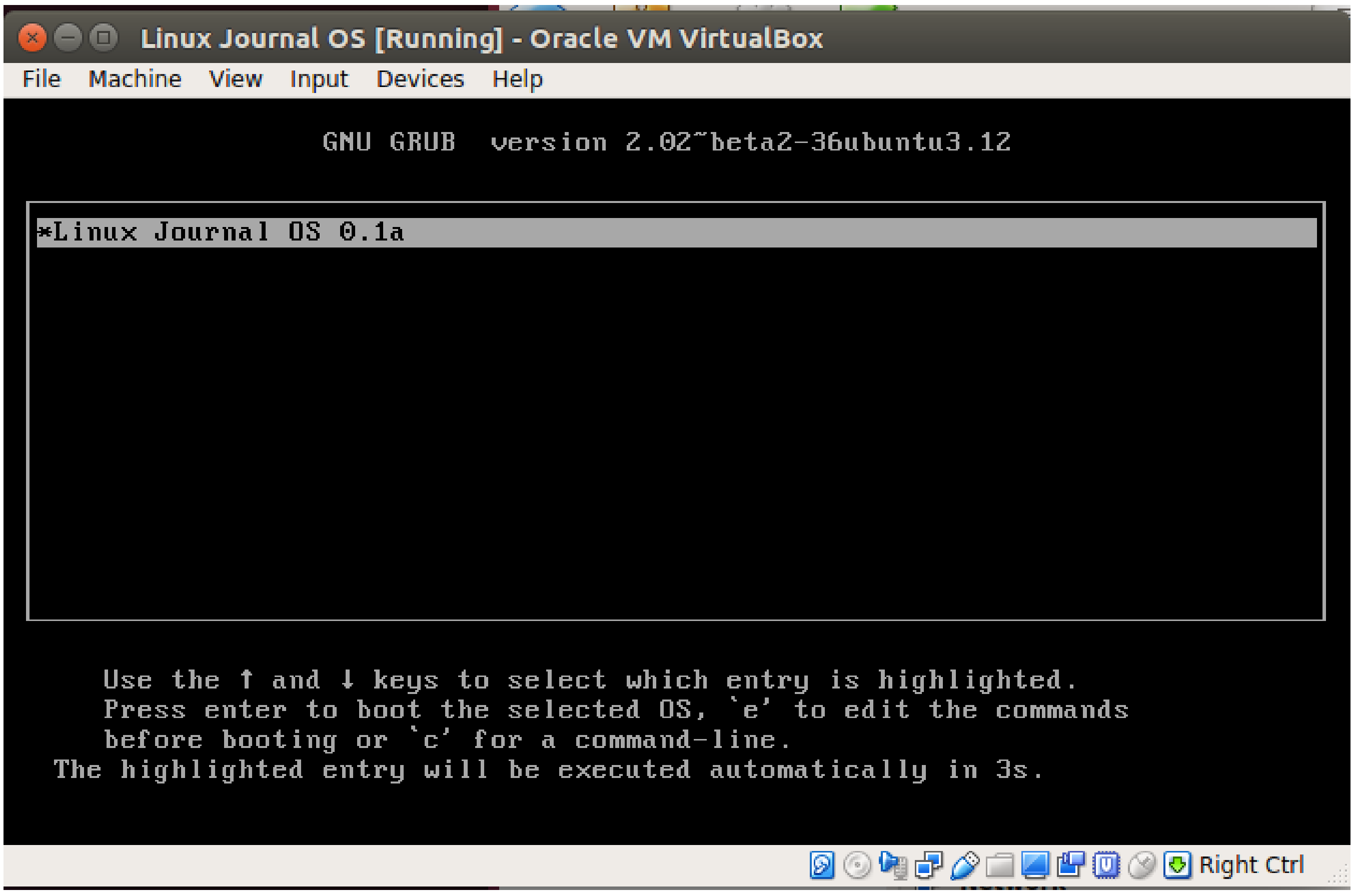
Now run `grub-install` to install all the necessary modules and boot records to the volume:

```
$ sudo grub-install --root-directory=/home/petros/tmp/ /dev/sdd
```

The `--root-directory` parameter defines the absolute path of the staging directory, while the last parameter is the block device without the partition's label.

Booting Up for the First Time

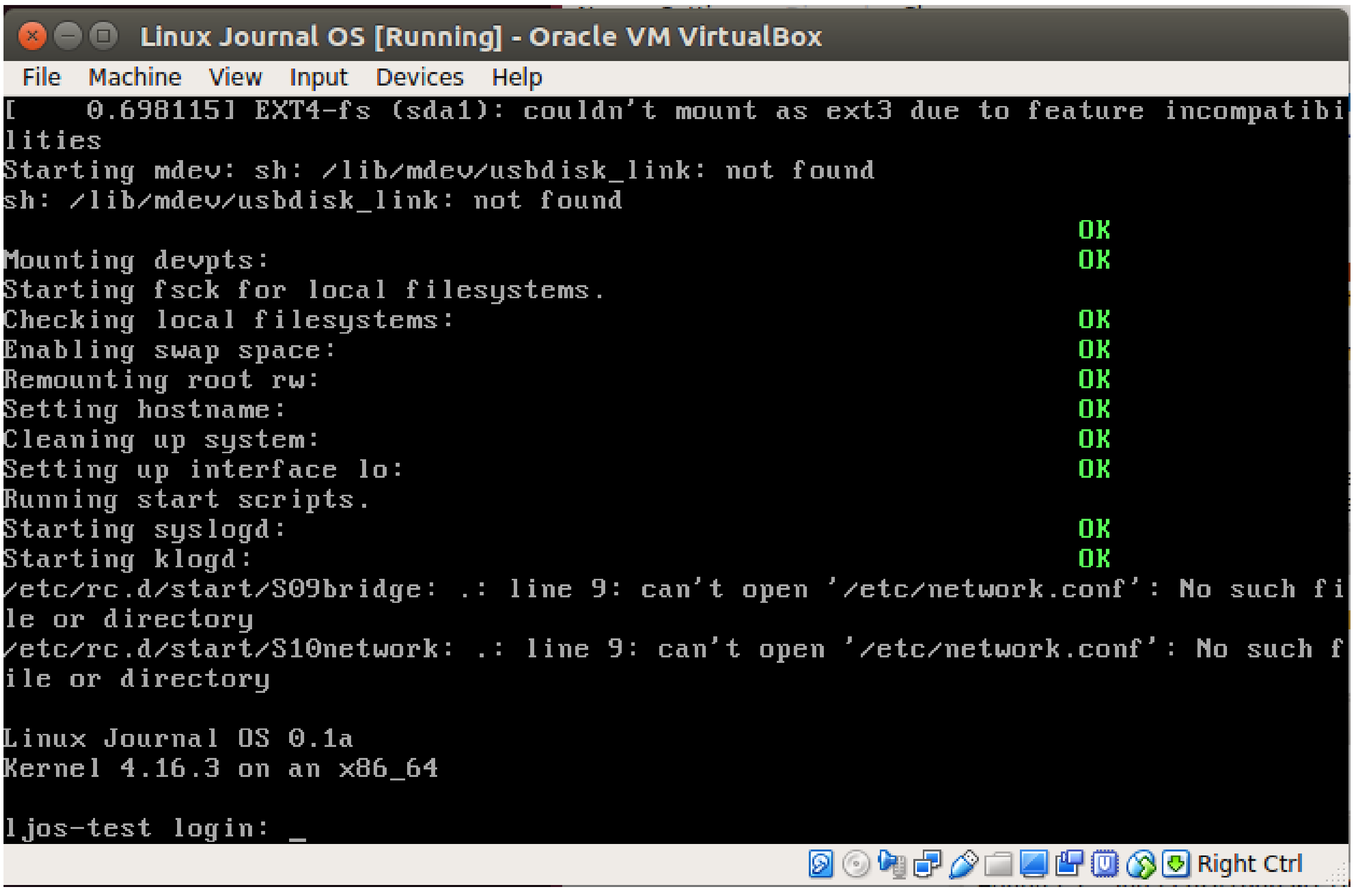
Now you're officially done. Install the HDD to the physical or virtual machine (as the primary disk drive) and power it up. You immediately will be greeted by the GRUB bootloader (Figure 1).



```
                 GNU GRUB  version 2.02~beta2-36ubuntu3.12

*Linux Journal OS 0.1a

      Use the ↑ and ↓ keys to select which entry is highlighted.
      Press enter to boot the selected OS, `e' to edit the commands
      before booting or `c' for a command-line.
   The highlighted entry will be executed automatically in 3s.
```

Figure 1. The GRUB Bootloader

And within one second (yes, you read that correctly, one second), you'll be at the operating system's login prompt (Figure 2).

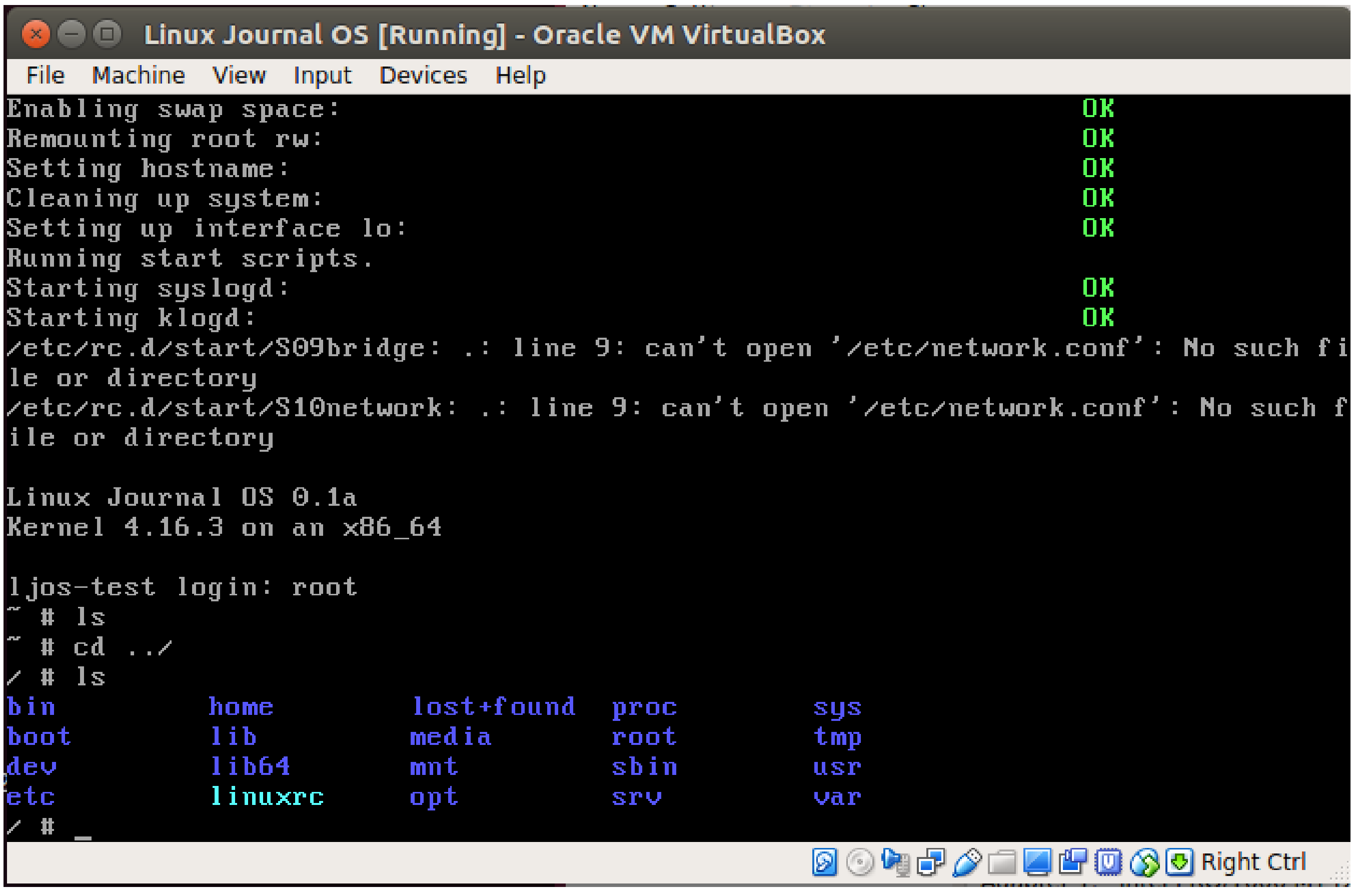

Linux Journal OS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
[    0.698115] EXT4-fs (sda1): couldn't mount as ext3 due to feature incompatibi
lities
Starting mdev: sh: /lib/mdev/usbdisk_link: not found
sh: /lib/mdev/usbdisk_link: not found
                                                                 OK
Mounting devpts:                                                 OK
Starting fsck for local filesystems.
Checking local filesystems:                                      OK
Enabling swap space:                                             OK
Remounting root rw:                                              OK
Setting hostname:                                                OK
Cleaning up system:                                              OK
Setting up interface lo:                                         OK
Running start scripts.
Starting syslogd:                                                OK
Starting klogd:                                                  OK
/etc/rc.d/start/S09bridge: .: line 9: can't open '/etc/network.conf': No such fi
le or directory
/etc/rc.d/start/S10network: .: line 9: can't open '/etc/network.conf': No such f
ile or directory

Linux Journal OS 0.1a
Kernel 4.16.3 on an x86_64

ljos-test login: _
```

Figure 2. The User Login Prompt

You'll notice a couple boot "error" and warning messages. This is because you're missing a couple files. You can correct that as you continue to learn the environment and build more packages into the operating system.

If you recall, you never set a root password. This was intentional. Log in as root, and you'll immediately fall into a shell without needing to input a password. You can change this behavior by using BusyBox's `passwd` command, which should have been built in to this image.


Linux Journal OS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Enabling swap space:                                              OK
Remounting root rw:                                               OK
Setting hostname:                                                 OK
Cleaning up system:                                               OK
Setting up interface lo:                                          OK
Running start scripts.
Starting syslogd:                                                 OK
Starting klogd:                                                   OK
/etc/rc.d/start/S09bridge: .: line 9: can't open '/etc/network.conf': No such fi
le or directory
/etc/rc.d/start/S10network: .: line 9: can't open '/etc/network.conf': No such f
ile or directory

Linux Journal OS 0.1a
Kernel 4.16.3 on an x86_64

ljos-test login: root
~ # ls
~ # cd ../
/ # ls
bin         home        lost+found  proc        sys
boot        lib         media       root        tmp
dev         lib64       mnt         sbin        usr
etc         linuxrc     opt         srv         var
/ # _
```

Figure 3. Executing a Few Simple Tasks

Enjoy!

Next Steps

So, where does this leave you now? You were able to build a custom Linux distribution for the generic x86-64 architecture from open-source packages and load into it successfully. Employing the same cross-compilation toolchain, you can use a similar process to build more utilities and libraries into the operating system, such as networking utilities, storage volume management frameworks and more.

For future builds, be sure to keep the cross-compilation build directory and your headers, and be sure to continue exporting the following variables (which you probably can throw into a script file):

```
set +h
umask 022
export LJOS=~/lj-os
export LC_ALL=POSIX
export PATH=${LJOS}/cross-tools/bin:/bin:/usr/bin
unset CFLAGS
unset CXXFLAGS
export LJOS_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
export LJOS_TARGET=x86_64-unknown-linux-gnu
export LJOS_CPU=k8
export LJOS_ARCH=$(echo ${LJOS_TARGET} | sed -e 's/-.*//'
 ↪-e 's/i.86/i386/')
export LJOS_ENDIAN=little
export CC="${LJOS_TARGET}-gcc"
export CXX="${LJOS_TARGET}-g++"
export CPP="${LJOS_TARGET}-gcc -E"
export AR="${LJOS_TARGET}-ar"
export AS="${LJOS_TARGET}-as"
export LD="${LJOS_TARGET}-ld"
export RANLIB="${LJOS_TARGET}-ranlib"
export READELF="${LJOS_TARGET}-readelf"
export STRIP="${LJOS_TARGET}-strip"
```

Petros Koutoupis, *LJ* Editor at Large, is currently a senior performance software engineer at Cray for its Lustre High Performance File System division. He is also the creator and maintainer of the RapidDisk Project. Petros has worked in the data storage industry for well over a decade and has helped pioneer the many technologies unleashed in the wild today.

Load 16 comments