


SREcon25 Europe/Middle East/Africa registration is open!
Register Now



Menu

 [Join the conversation](#)
[Back to ;login: Online](#)

A Tutorial on Building Custom Linux Appliances

December 14, 2021

Tutorial

Authors:

[Han Dong](#), [Jonathan Appavoo](#)

Article shepherded by:

Rik Farrow

Experimentally evaluating what runtime effects a change to a software component is a surprisingly difficult task. This is true regardless if the

component being changed is within the kernel, a kernel module, user library or application software. As is the case with all experimental efforts, one important step is to control as many external and non-deterministic perturbations as possible. This allows one to gather base line results and gain confidence that measured values are causally related to the change itself and not simply the result of system noise. After this, one can then evaluate the change in progressively noisy settings that may reflect more realistic deployments with the knowledge of the base line results. This approach is especially important when doing systems research to ensure that effect of proposed changes are soundly quantified both in terms of reproducible and causally explainable results.

As discussed in a recent *login*: [article](#), almost all modern systems research is conducted on Linux; which typically implies a flavor such as Ubuntu, Fedora, etc. Today's Linux software environment is typically packaged in a complex standard distribution, furthermore, it is also not always clear how much effort researchers have taken to remove as many extraneous processes and kernel modules as possible to ensure a clean and stable Linux environment. At first glance, it might seem that it requires a heroic effort to construct a minimal execution setup; this tutorial demonstrates that it is surprisingly easy to get simple environment setup. As a motivating example, my ThinkPad laptop running Fedora 24 with Linux v5.14.15 idles with 297 processes and 159 kernel modules loaded. On the same laptop, I also booted a custom Linux appliance running Linux v5.14.1 that idles with 100 processes and 0 kernel modules loaded.

Linux appliances are a relatively old idea [[10](#)], often understood as a self-contained system image containing just enough software and operating systems support to run a single application. In this article, I explain how to create such a Linux appliance suitable for running benchmarks on a minimal system, thereby avoiding running the long list

of standard processes that can perturb systems tests. Furthermore, as the root filesystem of the Linux appliance is loaded as a RAM disk by default, this can further reduce system noises such as disk paging.

Goals

There are many tutorials online to build your own Linux kernel and a root filesystem used for booting the kernel, often called the initramfs [1, 2, 3, 5, 6, 7]. However, their use cases are typically too general and the steps involved can be quite complex. This tutorial will demonstrate that it is in fact surprisingly easy to get a barebones Linux up and running that is ready to execute some simple programs. Concretely, here are the general steps that this tutorial covers: 1) Creating an initramfs, 2) Building/Configuring a Linux kernel from source and device drivers, 3) Getting programs to run, and 4) Booting the appliance.

Preparation

Here's a general breakdown of what is required on your end: 1) you should have a testing machine that will be booting the Linux appliance, 2) you should install a pre-existing Linux flavor on your machine (i.e. Fedora, etc), and 3) have your machine connected via Ethernet (optional).

Notes

Having a pre-existing OS on your testing machine is important. You want your system to be in a state to test out the intended programs and to install extra packages.

One key difference between this tutorial and many others is that we will be simply copying existing system libraries and programs to get a functional Linux system rather than using a tool such as [busybox](#).

Step 1: Create initial initramfs structure

On your testing machine, open a terminal and run *sudo -s* to start working as *root*. Next, export the name for the initramfs by running *export LFS=~/.initfs*. Run the code snippet below to create an initial directory structure; these directories are typically where default system libraries are placed (details [here](#)).

```
mkdir -pv $LFS

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
```

Create initial directory structure

Step 2: Getting programs to run

After you've created the directory structure above, the *chroot* program can then be used to test out *\$LFS* filesystem. However, as the filesystem itself is bare without any programs in it, you should see the following error when running *chroot \$LFS*:

```
chroot: failed to run command '/bin/bash': No such file or directory
```

To get *chroot \$LFS* working, follow the snippet of code below:

```
[root@ ~]# chroot $LFS
```

```
chroot: failed to run command '/bin/bash': No such file or directory
```

```
## figure out where bash lives
```

```
[root@ ~]# which bash
```

```
/usr/bin/bash
```

```
## copy bash to correct directory in $LFS
```

```
[root@ ~]# cp /usr/bin/bash $LFS/usr/bin/
```

```
## get library dependencies of bash
```

```
[root@ ~]# ldd /usr/bin/bash
```

```
linux-vdso.so.1 [0x00007ffd6ffa6000]
```

```
libtinfo.so.6 => /lib64/libtinfo.so.6 [0x00007f8e3751b000]
```

```
libdl.so.2 => /lib64/libdl.so.2 [0x00007f8e37514000]
```

```
libc.so.6 => /lib64/libc.so.6 [0x00007f8e37345000]
```

```
/lib64/ld-linux-x86-64.so.2 [0x00007f8e376bc000]
```

```
## copy over libraries to correct location, skip linux-vdso.so.1 as the
```

```
[root@ ~]# cp /lib64/libtinfo.so.6 $LFS/lib64/
```

```
[root@ ~]# cp /lib64/libdl.so.2 $LFS/lib64/
```

```
[root@ ~]# cp /lib64/libc.so.6 $LFS/lib64/
```

```
[root@ ~]# cp /lib64/ld-linux-x86-64.so.2 $LFS/lib64/
```

```
## /bin/bash should work now with $LFS
```

```
[root@ ~]# chroot $LFS
```

```
bash-5.1#
```

Getting /bin/bash to work

A script to automate the copying of programs and its libraries

To get other programs running, you will need to follow similar steps as shown above and these programs will have different dependencies on libraries and other files, etc. To help automate these steps, a simple script is provided below:

```
[root@ ~]# cat copy_appliance_libs

#!/bin/bash

export MYINIT=${MYINIT:=''}

export BINS=${BINS:=''}

for bins in ${BINS}; do

    echo $bins

    ## figure out where program lives

    bins_loc=$(which $bins)

    bins_dir_loc=$(which $bins | xargs -I '{}' dirname '{}')

    ## if program does not exist in initramfs

    if [[ ! -e "${MYINIT}/${bins_loc}" ]]; then

        echo $bins_loc

        echo "${MYINIT}/${bins_dir_loc}"
```

```
## set up directory and copy programs to initramfs

[[ -e "${MYINIT}/${bins_dir_loc}" ]] || mkdir -p "${MYINIT}/${bins_

cp $bins_loc "${MYINIT}/${bins_dir_loc}"

## figure out library dependencies of $bins

libs_loc=$(ldd $bins_loc | grep "=> /" | awk '{print $3}')

## set up directory and copy libs to initramfs

for libs in ${libs_loc}; do

    libs_dir=$(dirname $libs)

    ## if library does not exist in initramfs

    if [[ ! -e "${MYINIT}/${libs}" ]]; then

        echo $libs

        [[ -e "${MYINIT}/${libs_dir}" ]] || mkdir -p "${MYINIT}/${libs_

        cp $libs "${MYINIT}/${libs_dir}"

    fi

done

echo "+++++"

fi

done
```


Example usage: `MYINIT=$LFS BINS="ls cd" ./copy_appliance_libs`

Notes

Just copying the libraries may not be enough to get all programs running, sometimes *strace* is needed to figure out what other files your program is accessing through system calls such as *openat()*, *read()*, *access()*. Then you'll need to either create or copy these files from the existing system - this part can get tricky!

Keep in mind that sometimes the system libraries can also have dependencies on other libraries.

Step 3: Create the rest of the initramfs structure

First, use the script above to automatically copy the following programs to the initramfs: *ls cd pwd cat mount umount mkdir mknod cp mv install ln touch chgrp chmod poweroff reboot readlink ip dhclient ps wc uname hostname more tail head grep find df free*

Next, run

chroot \$LFS

to get safely inside the *chroot* environment and run the following snippets of code to create the rest of the filesystem structure. The

steps below are slightly modified from Chapter 7 of Linux From Scratch [7].

Important: The following steps involve creating files and folders under the / directory, so be sure to first run *chroot \$LFS* to get safely inside the *chroot* environment, else you may accidentally wipe your existing Linux system.

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

```
mkdir -pv /dev/pts
```

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

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

Setup virtual kernel filesystems

```
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}
```

```
install -dv -m 0750 /root
```

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

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

```
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
```

Create the rest of the directories

```
cat > /etc/hosts << EOF
```

```
127.0.0.1 localhost myinitfs
```

```
::1      localhost
```

```
EOF
```

```
cat > /etc/passwd << EOF
```

```
root:x:0:0:root:/root:/bin/bash
```

```
EOF
```

```
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:
```

```
users:x:999:
```

```
EOF
```

Set up root user and groups

Step 4: Create startup /init script for the appliance

After following the previous [steps](#) to ensure a basic set of programs are runnable in your appliance, the next step is to create the startup file that is essentially the program that Linux runs to initiate the rest of the system. Modern systems have generally migrated to use [systemd](#) as the bootstrapping program due to its comprehensive set of tools. However, this tutorial will instead use the older [init](#) script as it is 1) simpler to edit, and 2) enables greater control to begin automating experiments. While still in *chroot* environment, run the code below to create the */init* file:

```
cat > /init << EOF

#!/bin/bash

export HOME=/root

export LOGNAME=root

export TERM=vt100

export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin

export ENV="HOME=\$HOME LOGNAME=\$LOGNAME TERM=\$TERM P/

# setup standard file system view

mount -t proc /proc /proc

mount -t sysfs /sys /sys

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

```
# Some things don't work properly without /etc/mtab.
```

```
ln -sf /proc/mounts /etc/mtab
```

```
# if we get here then we might as well start a shell :-]
```

```
/bin/bash
```

```
# if bash fails, shuts off machine
```

```
poweroff -f
```

```
EOF
```

/init file creation

After that, set permissions by running `chmod 755 /init`. Next, exit out of *chroot* environment and `cd $LFS` in order to compress the initramfs into the cpio format by running:

```
find . | cpio -o -H newc > ../myinitfs.cpio
```

To make this as a bootable option, run

```
cp ../myinitfs.cpio /boot
```

Step 5: Building the Linux kernel

First, make sure you have the necessary packages installed to build a Linux kernel from source, see [8, 9], you can skip this step by taking an existing Linux kernel image at `/boot/vmlinuz-*` and head to [Step 6](#) to boot it. Though, building from source enables greater control over its configuration setup.

To start, run `uname -r` on your existing system to get its version information and download a tarball of that version at the kernel.org website. This doesn't need to be *exactly* the same, i.e. my machine runs 5.14.15-200.fc34.x86_64 and the 5.14.1 tarball still worked.

After you download and unzip the Linux kernel, `cd` into the kernel directory and run

```
make menuconfig
```

to generate a default `.config` file. Next, to prep the kernel build run:

```
make prepare && make modules_prepare
```

Then, to build the bootable kernel image, run

```
make -j bzImage
```

This process will take a while and eventually you will see the following success message (you may see an error(s) regarding the need to disable certain options in `menuconfig`; if so, do it and then rerun `make -j bzImage`):

```
Kernel: arch/x86/boot/bzImage is ready
```

At this point, make the `bzImage` be bootable by copying it to `/boot`:

```
cp arch/x86/boot/bzImage /boot
```

Custom kernel configurations

Not covered in this tutorial is the importance to customize your Linux kernel configurations. A very interesting study: "[An analysis of performance evolution of Linux's core operations](#)", has provided a nice list of kernel configuration options you may want to disable for performance reasons.

Step 6: Booting the Linux appliance

Next, the created initramfs and Linux kernel image are added to GRUB as bootable options. Dependent on the Linux flavor, these approaches might be slightly different (e.g. [Ubuntu](#), [Fedora](#)). On my Fedora install, the file at `/etc/grub.d/40_custom` was modified with a new menuentry option that contains our custom Linux image and initramfs; the snippet below shows contents of that file. After this, update GRUB by running

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

```
#!/bin/sh
```

```
exec tail -n +3 $0
```

```
# This file provides an easy way to add custom menu entries. Simply
```

```
# menu entries you want to add after this comment. Be careful not to
```

```
# the 'exec tail' line above.
```

```
menuentry 'Linux_appliance' {
```

```
    linux ($root)/bzImage root=/dev/ram0 rw
```



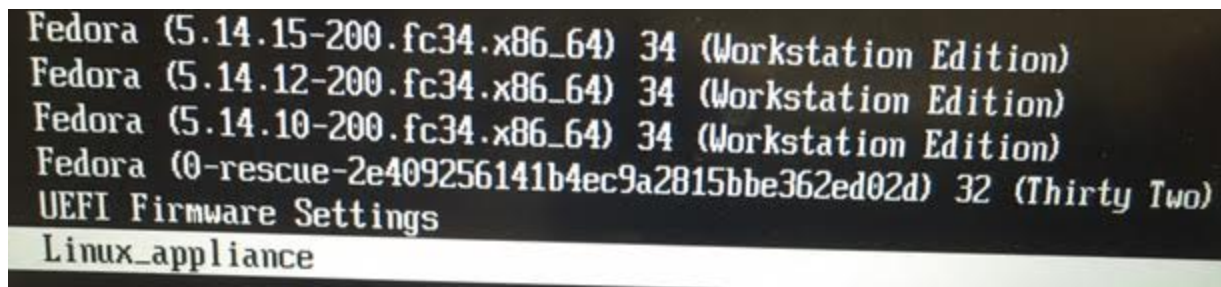
```
initrd  ($root)/myinitfs.cpio  
  
}
```

Contents of custom GRUB entry at `/etc/grub.d/40_custom`

Notes

I have found differences between Linux flavors with respect to how the `($root)` or `/boot` variables are used by GRUB to locate the `initramfs` and kernel images (details [here](#)). As a rough rule of thumb, I simply take a look at the `grub.cfg` file under `/boot` and copy the examples of how other boot options are defined.

At this point, restart your testing machine, select `Linux_appliance` as boot option in the GRUB menu (you may need to manually enable showing GRUB menu in your Linux install) and the Linux appliance should then be booted and you will be presented with a simple bash prompt:



Enable GRUB menu select and you should then see the appliance as a bootable option

```

3.3404611 hid-generic 0003:7844:6060.0003: input,hidraw2: USB HID v1.11 Mouse [XIU
bash-5.1# ls
bin boot dev etc home init lib lib64 media mnt opt proc root run sbin src
bash-5.1# uname
Linux
bash-5.1# cat /proc/cmdline
BOOT_IMAGE=(hd1,gpt2)/bzImage_5_14_2 root=/dev/ram0 rw
bash-5.1# ps aux | wc -l
103
bash-5.1# free -h
               total          used          free      shared  buff/cache   available
Mem:           38Gi          79Mi          38Gi           0B           37Mi           38Gi
Swap:            0B             0B             0B
bash-5.1# reboot -f_

```

Booted appliance - run `reboot -f` to restart machine again

Booting using PXE

While booting with GRUB enables a quick way to test the appliance, the [PXE](#) protocol is preferable for setting up experiments as it allows a single master node to coordinate and boot multiple servers in a more programmed fashion.

Step 7 (Optional): Getting an ethernet device running

In this section, we will demonstrate an example of how chroot can be used to scope out and get a slightly advanced portion of Linux running. In this example, we will be enabling the Ethernet device in order to send DHCP requests for a new IP address; in this case, your machine should be either hooked up to a local LAN or another machine that is running a DHCP server.

```
## find your device
```

```
[root@ ~]# lspci | grep Ethernet
```

```
00:1f.6 Ethernet controller: Intel Corporation Ethernet Connection (6
```

```
## gets its device driver information
```

```
[root@ ~]# lspci -s 00:1f.6 -vvv | grep "Kernel driver"
```

```
Kernel driver in use: e1000e
```

Find out device drivers info

Building device drivers into Linux kernel

The simplest way to automatically enable the Ethernet device in the appliance is to manually set the module as "Y" after searching for "e1000e" in the Linux kernel *make menuconfig* menu shown in [Step 5](#). After this, you'll need to run *make -j bzImage* again to build the new kernel that now has the e1000e device driver automatically built in with the kernel image. You can also manually build the device drivers and use *insmod* to insert them manually (details [here](#)), though if you are going down this route, use *modinfo* to resolve potential kernel module inter-dependencies.

```
[root@ ~]# chroot $LFS
```

```
bash-5.1# ip a
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNl
```

```
....
```

```
2: enp0s31f6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
```

```
....
```

```
...
```

```
## Run dhclient on the interface connected to ethernet, e.g. enp0s3
```

```
bash-5.1# dhclient -v enp0s31f6
```

```
Can't create /var/run/dhclient.pid: No such file or directory
```

```
Internet Systems Consortium DHCP Client 4.4.2b1
```

```
Copyright 2004-2019 Internet Systems Consortium.
```

```
All rights reserved.
```

```
For info, please visit https://www.isc.org/software/dhcp/
```

```
can't create /var/lib/dhclient/dhclient.leases: No such file or directo
```

```
execve [/usr/sbin/dhclient-script, ...]: No such file or directory
```

```
.....
```

```
.....
```

```
bash-5.1# exit
```

```
exit
```

```
## Error messages above indicate we are missing the following direc
```

```
[root@ ~]# mkdir $LFS/var/lib/dhclient
```

```
[root@ ~]# mkdir $LFS/var/run/
```

```
[root@ ~]# cp /usr/sbin/dhclient-script $LFS/usr/sbin/
```

```
## Retry running dhclient
```

```
[root@ ~]# chroot $LFS
```

```
bash-5.1# dhclient -v enp0s31f6
```

```
.....
```

```
.....
```

```
/usr/sbin/dhclient-script: line 281: ipcalc: command not found
```

```
/usr/sbin/dhclient-script: line 281: cut: command not found
```

```
/usr/sbin/dhclient-script: line 878: arping: command not found
```

```
/usr/sbin/dhclient-script: line 882: arping: command not found
```

```
/usr/sbin/dhclient-script: line 882: grep: command not found
```

```
/usr/sbin/dhclient-script: line 882: awk: command not found
```

```
/usr/sbin/dhclient-script: line 882: cut: command not found
```

```
/usr/sbin/dhclient-script: line 882: cut: command not found
```

```
/usr/sbin/dhclient-script: line 883: grep: command not found
```

```
/usr/sbin/dhclient-script: line 883: awk: command not found
```

```
/usr/sbin/dhclient-script: line 883: uniq: command not found
```

```
/usr/sbin/dhclient-script: line 268: ipcalc: command not found

/usr/sbin/dhclient-script: line 268: cut: command not found

/usr/sbin/dhclient-script: line 268: ipcalc: command not found

/usr/sbin/dhclient-script: line 268: cut: command not found

/usr/sbin/dhclient-script: line 108: mktemp: command not found

....
```

Using chroot to get DHCP working

At this point, */usr/sbin/dhclient-script* is indicating that the following programs are missing in the initramfs: *ipcalc*, *cut*, *arping*, *grep*, *awk*, *uniq* and *mktemp*. Use the *copy_appliance_libs* script [above](#) to get those programs running and rerun *dhclient* in order to see that the DHCP protocol works and you should be able to see a new IP address assigned to your Ethernet interface.

Step 8 (Optional): Example of scripting experiments with */init*

The */init* script can be easily extended for automating experiments, as an example, the snippets below show a modified

/etc/grub.d/40_custom and */init* file that parses the extra GRUB arguments in order to customize the bash environment upon boot.

```
#!/usr/bin/sh
```

```
exec tail -n +3 $0
```

```
# This file provides an easy way to add custom menu entries. Simply
```

```
# menu entries you want to add after this comment. Be careful not to
```

```
# the 'exec tail' line above.
```

```
menuentry 'Linux_appliance' {
```

```
    linux ($root)/bzImage_5_14_2 root=/dev/ram0 rw appNode=\'nodeC
```

```
    initrd ($root)/myinitfs.cpio
```

```
}
```

Updated */etc/grub.d/40_custom* file

```
#!/bin/bash
```

```
export HOME=/root
```

```
export LOGNAME=root
```

```
export TERM=vt100
```

```
export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
```

```
export ENV="HOME=\$HOME LOGNAME=\$LOGNAME TERM=\$TERM P/
```

```
# setup standard file system view
```

```
mount -t proc /proc /proc
```

```
mount -t sysfs /sys /sys
```

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

```
# Some things don't work properly without /etc/mtab.
```

```
ln -sf /proc/mounts /etc/mtab
```

```
# APP PARAMETERS
```

```
export APP_MYNODE=""
```

```
cmdline=${cat /proc/cmdline}
```

```
# PARSE OUT APP ARGUMENTS
```

```
# set a unique name for this node so that it can id itself
```

```
if [[ ${cmdline} =~ ^.*appNode=\\\\"[.]*\\\\".*$ ]]; then
```

```
    APP_MYNODE="${BASH_REMATCH[1]}"
```

```
    APP_MYNODE=${APP_MYNODE%%\\\\"*}
```

```
## customize node name
```

```
hostname ${APP_MYNODE}
```

```
## customize bash prompt
```

```
export PS1="${APP_MYNODE}> "
```


fi

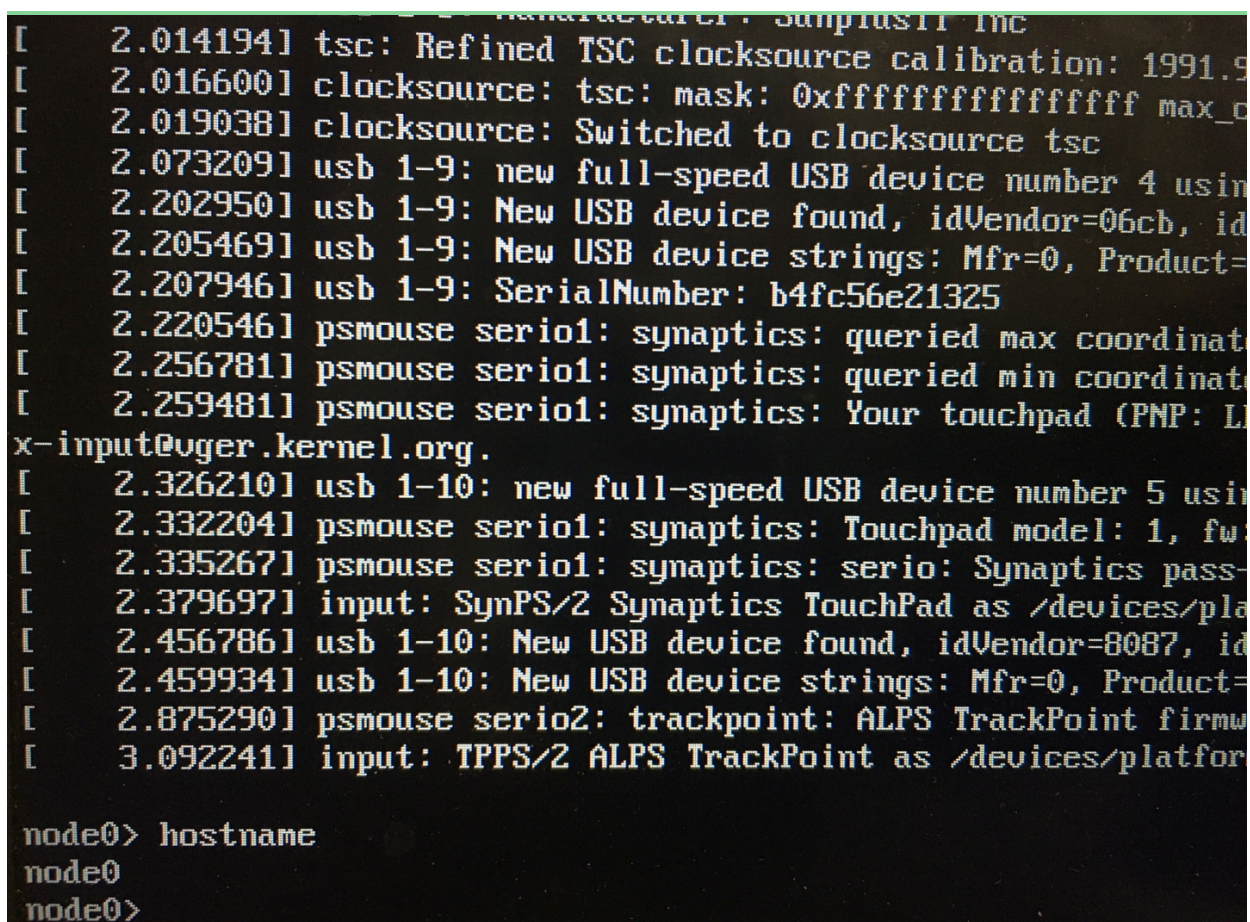
if we get here then we might as well start a shell :-)

/bin/bash

if bash fails, shuts off machine

poweroff -f

Updated /init file



The screenshot shows a terminal window with a black background and green text. It displays a series of kernel boot messages, including clocksource calibration, USB device detection, and touchpad initialization. At the bottom, the user 'node0' is shown typing the command 'hostname'.

```
[ 2.014194] tsc: Refined TSC clocksource calibration: 1991.9
[ 2.016600] clocksource: tsc: mask: 0xffffffffffffffff max_c
[ 2.019038] clocksource: Switched to clocksource tsc
[ 2.073209] usb 1-9: new full-speed USB device number 4 usin
[ 2.202950] usb 1-9: New USB device found, idVendor=06cb, id
[ 2.205469] usb 1-9: New USB device strings: Mfr=0, Product=
[ 2.207946] usb 1-9: SerialNumber: b4fc56e21325
[ 2.220546] psmouse serio1: synaptics: queried max coordinat
[ 2.256781] psmouse serio1: synaptics: queried min coordinat
[ 2.259481] psmouse serio1: synaptics: Your touchpad (PNP: L
x-input@vger.kernel.org.
[ 2.326210] usb 1-10: new full-speed USB device number 5 usin
[ 2.332204] psmouse serio1: synaptics: Touchpad model: 1, fw:
[ 2.335267] psmouse serio1: synaptics: serio: Synaptics pass-
[ 2.379697] input: SynPS/2 Synaptics TouchPad as /devices/pla
[ 2.456786] usb 1-10: New USB device found, idVendor=8087, id
[ 2.459934] usb 1-10: New USB device strings: Mfr=0, Product=
[ 2.875290] psmouse serio2: trackpoint: ALPS TrackPoint firmw
[ 3.092241] input: TPPS/2 ALPS TrackPoint as /devices/platfor

node0> hostname
node0
node0>
```

Booted Linux appliance output

Conclusion

This tutorial illustrates the initial steps to creating a stable and clean working environment for running experiments in Linux. Various pieces such as using chroot to scope out how to get complicated portions of Linux running and modifying init to automate experiments only scratch the surface of how users can customize their own Linux environments. In addition, there are hardware features and other components of Linux not covered in this tutorial that a user should account for in order to minimize overall system noise; examples of these include disabling hyper-threads, page sizes, pinning threads to cores, and many others.

References:

[1] Petros Koutoupis, DIY: Build a Custom Minimal Linux Distribution from Source,

<https://www.linuxjournal.com/content/diy-build-custom-minimal-linux-dist...>, 7/3/2018

[2] Debian, How initramfs works,

<https://wiki.debian.org/initramfs>, 5/10/2021

[3] Gentoo Foundation, Inc., Custom Initramfs,

https://wiki.gentoo.org/wiki/Custom_Initramfs, 6/24/2021

[4] Rob Landley, ramfs, rootfs and initramfs,

<https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initra...>, 10/17/2005

[5] Ole Andreas W. Lyngv  r, Creating a initramfs image from scratch, <https://lyngvaer.no/log/create-linux-initramfs>, Accessed 10/26/2021

[6] Werner Almesberger and Hans Lermen, Using the initial RAM disk (initrd), <https://www.kernel.org/doc/html/latest/admin-guide/initrd.html>, 1996, 2000

[7] Gerard Beekmans and Bruce Dubbs, Linux From Scratch, <https://www.linuxfromscratch.org/lfs/view/stable/index.html>, Published 9/1/2021

[8] Steve Scargall, How to build an upstream Fedora Kernel from source, <https://stevescargall.com/2020/09/14/how-to-build-an-upstream-fedora-ker...>, 9/14/2020

[9] rs2009, BuildYourOwnKernel, <https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel>, 2/3/2021

[10] Shaffer, Michael W. A Linux Appliance Construction Set. 14th Systems Administration Conference (LISA 2000). 2000.

Article Categories: Operating Systems, Programming, Linux

Last updated February 8, 2023

Authors:



Han Dong is currently pursuing his PhD at Boston University with a focus on analyzing performance and energy of

different Operating Systems through hardware tuning. As a Research Intern at Red Hat, he is also working on methods to combine systems level data analysis with machine learning techniques towards smarter hardware policies. In his free time, Han enjoys building projects with microcontrollers and reading.

handong@bu.edu



Jonathan Appavoo, PhD is an Associate Professor at Boston University in the department of Computer Science. Prior to that he was a Research Staff Member at IBM's T.J. Watson Research Center in New York. Professor Appavoo loves to hack on computers and dream about future systems and has surprisingly found a way to make a living at it. As a graduate student at the University of Toronto (UofT) he began his PhD working in Computer Vision hoping to build robots. He quickly realized the error of his ways and switched to working on

the Tornado operating systems – a novel multiprocessor OS for an ambitious large scale NUMA multiprocess being designed and built at UofT. He followed Tornado's journey to IBM and worked on IBM's K42 Research OS and then the Libra library OS. After this he helped found Project Kittyhawk to explore the construction of a global-scale computer and its attendant cloud based usage model. Through these experiences he nurtured a vision of a novel Programmable Smart Machine (PSM) computer model, that combines biologically inspired mechanisms, where the system's performance and efficiency grow automatically as a function of its size and usage. He received an National Science Foundation CAREER Award to pursue the PSM model. Professor Appavoo, along with his graduate students, continue to hack on OSES and work on the PSM model. Professor Appavoo has been very fortunate to have worked with amazing colleagues and students and is thankful to all of them (especially for their patience).

jappavoo@bu.edu

[Log in](#) to post comments



[Contact USENIX](#) • [Privacy Policy](#)

© USENIX 2025

EIN 13-3055038

Website designed and built

by Giant Rabbit LLC

Powered by Backdrop CMS