Setting up your own bootable linux OS

Thomas Mortensson,
thomasmortensson@googlemail.com

September 1, 2016

1 Aim of this workshop

The aim of this workshop is to give you a better understanding of Linux, the kernel and the typical BIOS boot process for x86 and x86_64 computers.

We cover extensions to this workshop at the end to enhance our linux image to allow us to additionally boot EFI mode computers. This is useful with newer machines which may not support older BIOS boot.

2 Pre-requisites

This is an intermediate level workshop and NOT designed for beginners. People who will get the most from attending this workshop are people early in their Linux experience with basic knowledge who wish to understand more about the guts of the system (specifically the boot process).

- Some simple familiarity of Linux and commands required to navigate the CLI
- Access to an x86 or x86_64 architecture PC (Basically any modern laptop or desktop machine excluding chromebooks)
- A Debian or Ubuntu based build machine with an Internet connection (Can be virtual)
- A USB stick for final testing (Unneccessaty if testing on virtual machine)

3 Setting up the build environment

For this workshop I will be assuming at a minimum, a bash like interface to the CLI and installation of the build-essential tools and grub. We will be using tools such as Make and GCC to compile both Linux kernel and busybox for inclusion in our initrd. Additionally, grub will be required to make our image bootable.

When you have all required packages, you can generate yourself a folder ~/build. We will refer to this folder throughout this tutorial as $BUILD

4 Compiling the Linux Kernel

To get the latest version of the Linux kernel, head on over to the kernel.org page `https://www.kernel.org` and download the latest Stable kernel. If you copy the link provided, you can wget this into your build machine. For example with Kernel 4.7.2 I run:

```
wget https://cdn.kernel.org/pub/linux/kernel/v4.x/
    linux-4.7.2.tar.xz
```

Chances are that the file will come down as a XZ zipped file. You can unpack this using the command:

```
thomas@ubuntu:~/ tar -xf linux-4.7.2.tar.xz
```

We can now change into the linux src folder by issuing:

```
cd linux-4.7.2
```

We build the kernel by issuing:

```
make defconfig && make
```

Once this has completed we have our new Linux kernel compiled here: arch/x86/boot/bzImage

5 Compiling busybox

Busybox is an essential component in any small linux distro or pre-boot environment. Busybox is a collection of core components from the GNU toolchain linked into a single binary totalling only ~500KB. Busybox achieves its tiny footprint by statically linking all libraries required into a single executable binary and implementing a subset of the optional arguments of core programs. This is a design concept which is slightly at odds with the core linux design methodology of modularity however the choices taken have been made to minimize filesize for a very specific use case. Interestingly to call a core tool such as ls, we pass the ls argument to the busybox executable and busybox interprets the command argument list as an arg list for requested executable. Thus we can generate symlinks to the busybox executable to populate our bin directory within our linux system.

We're going to move back to the $BUILD folder to download the latest version of busybox. Head on over to `https://busybox.net/downloads/` and download the latest version. For the purposes of this document we will download busybox-1.25.0.tar.bz2

```
wget https://busybox.net/downloads/busybox-1.25.0.
    tar.bz2
```

Once you have the tar archive in your build directory extract it with the following command:

```
tar -jxf busybox-1.25.0.tar.bz2
```

This should extarct a directory on your local machine which we can change into

```
cd busybox-1.25.0/
```

We can now compile busybox with the make command. NOTE: we choose to statically compile the busybox executable.

```
make defconfig && make LDFLAGS=-static
```

We now have a binary in the current directory called busybox. You can test it by running its 'ls' command:

```
./busybox ls
```

6 Generating an Initial Ramdisk

We're now going to build the initial ram disk for our linux system. When we boot up the operating system the linux kernel is invoked by the bootloader and it needs an initial rootfs environment to process with. The Initial RAM Disk (initrd) is a file containing this initial ram filesystem zipped up using the cpio archive mechanism. The file at the root of this filesystem labelled /init is the initial program which is executed by the linux kernel. The initial ram system is typically used only as a pre-loader to the main Linux root filesystem and as such should provide all drivers necessary to boot on your particular devices and filesystems. An example of this can be seen on entry level HP servers making use of the B120i RAID controller. This RAID 'implementation' is in fact a software abstraction of an Intel AHCI controller using drivers chainloaded by the initramfs (hpvsa.ko) to provide a RAID interface to the linux operating system. As this kernel driver is not natively supported in the main linux kernel tree in order to be able to boot via this interface a user must inject this kernel module into the initramfs.

We are going to make use of a very basic initramfs for the purpose of this workshop. We will provide a minimal setup required to run a busybox environment on a computer to demonstrate the bootup sequence of a standard Linux machine. To start out change directory back to your $BUILD folder. We will now generate a folder to house our initramfs.

```
mkdir initramfs
```

Now change directory to the initramfs folder.

```
cd initramfs
```

We will now populate our base directory structure:

```
mkdir bin dev proc sys tmp
```

Once we have those folders set up we need to generate three special devices for the kernel to make use of:

```
mknod /dev/console c 5 1
mknod /dev/null c 1 3
mknod /dev/zero c 1 5
```

These special devices are required by the Linux kernel for a few different purposes. The null device acts as a black hole. Users can send data to the null device and it will destroy it. Think of it as a black hole for bit streams, anything that goes in never returns. The zero device will always generate the output zero when read and the console device allows the kernel to display messages to the user and represents the first tty (teletypewriter). For a full Linux system we should generate additional special devices however this is beyond the scope of this introduction. If you'd like to learn more, check out LFS `http://www.tldp.org/LDP/lfs/LFS-BOOK-6.1.1-HTML/chapter06/devices.html`

Next, we need to generate our init script. Use your favourite text editor to generate the below file called 'init'.

```
#!/bin/ash
mount -t proc none /proc
mount -t sysfs none /sys
/bin/ash
```

Make it executable:

```
chmod +x ./init
```

We can now add busybox to our initramfs. From your initramfs folder type:

```
cp ../busybox-1.25.0/busybox ./bin/
```

Change directory into the bin folder

```
cd bin
```

And we can generate symlinks to our executables. e.g. to generate the ls executable type:

```
ln busybox ls
```

We can do this for other useful executables. You can find the full list at `https://busybox.net/downloads/BusyBox.html` under the commands section. Lets generate some useful commands:

```
ln busybox ash
ln busybox cat
ln busybox chmod
ln busybox chown
ln busybox cp
ln busybox dd
ln busybox df
ln busybox echo
ln busybox grep
ln busybox kill
ln busybox ln
ln busybox ls
ln busybox mkdir
ln busybox mv
ln busybox pwd
ln busybox rm
ln busybox uptime
ln busybox vi
ln busybox wc
ln busybox which
```

You can add more of the commands described in the busybox documentation at any time while in your operating system by changing to the bin directory and running theis command against the executable you wish.

Finally we need to generate the initrd file by changing back to the root of the initramfs folder

```
cd ..
```

And issuing the following command

```
find . | cpio -H newc -o | gzip > ../initrd.cpio.gz
```

This will generate a file initrd.cpio.gz in the $BUILD folder.

7 Setting up our disk

We will now set up our disk. If you have a USB device, plug into your virtual machine now (taking care not to mount the partitions) otherwise generate a 2nd harddisk within Virtualbox. Make note of the device name by checking out the output of lsblk. e.g. for my 512MB disk it is listed as /dev/sdb. The output is as such:

```
thomas@ubuntu:~/build$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0    20G  0 disk
    sda1   8:1    0    19G  0 part /
    sda2   8:2    0     1K  0 part
    sda5   8:5    0  1022M  0 part [SWAP]
sdb      8:16   0   512M  0 disk
sr0     11:0    1  1024M  0 rom
```

We must partition our disk so that we can load on linux and our initrd files. Lets do this as follows:

```
sudo fdisk /dev/sdb
```

We wish to make a new primary partition on partition 1 using the whole disk, we will be asked a series of questions so follow as I do. fdisk will offer defaults which we will accept (most importantly the last sector). You can accept defaults by pressing enter however I have copied the values for completeness.

```
Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-1048575, default 2048): 2048
Last sector, +sectors or +size{K,M,G} (2048-1048575,
     default 1048575): 1048575
```

Next we need to set this drive as bootable. We do this as follows:

```
Command (m for help): a
Partition number (1-4): 1
```

Next we need to set the type of the filesystem on this 1st partition. We set it to the linux type by running:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
```

Finally, we write this to disk:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
```

We generate a filesystem on this partition as follows:

```
mkfs.ext3 /dev/sdb1
```

We now have our working disk. We need to mount it and copy our linux kernel onto it and our initramfs. We first make a mount point and then mount the disk.

```
mkdir /tmp/fs
mount /dev/sdb1 /tmp/fs
```

We can now copy our linux bzImage and our initramfs to the root of this filesystem.

```
cp ~/build/linux-4.7.2/arch/x86/boot/bzImage ~/build
    /initrd.cpio.gz /tmp/fs/
```

8 Setting up a bootloader

We're going to use the grub bootloader to load up our linux machine. We load this onto our disk as follows:

```
grub-install --root-directory=/tmp/fs /dev/sdb
```

This copies a bootloader into the first 440 bytes of our disk. The computers BIOS knows to use this code to help load up our operating system.

OK, we should now be all ready to boot.

9 Bootup and usage

Boot up the PC making sure to select the USB stick as the boot device or the 2nd hard disk as the boot device if you installed on a Hard drive. You should be shown a prompt titled grub.

At this prompt type (Be sure to run only one command per line)

```
set root=(hd0,msdos1)
linux /bzImage
initrd /initrd.cpio.gz
boot
```

You should now see some scrolling text and the linux system should boot up. eventually you'll be dropped to an ash command prompt from where you can execute your linked commands. Give it a go by executing

```
echo Hello World
```

If this worked, then hooray. You've just booted up your very own Linux distribution!

10 extension: Understanding the MBR

hexdump 0xaa55 boot signatures

11 extention: Making it EFI bootable!

- Eltorito alt boot, hybrid images - FAT EFI partition - Grub EFI image