Work Log – M2M
Linux Distribution – Part 1
2020

1 Preface by Pr. Olivier Gruber

This document is a guide through the building our own small Linux Distribution.

You have been given a fully functional script to build a bootable disk image with a Linux kernel mounting an absolutely minimal root file system. The goal is therefore to understand how it is done.

This document is meant to help you navigate through the different required steps and tools. It has a structure, it points to important tricks or skills or tools, it also ask questions. If you are discovering this domain, we encourage you to follow the overall guiding contents of this document. If you are more of an expert, feel free to re-organize this document into something that suits your skills and expertise.

The starting point is to look at the overall structure in directories and files. Then, immediately look at the shell script "mkdist.sh", it is a completely functional script:

1. Build the "hello" directory
    1. Compile and link the "init process"
    2. Generate the initial ramdisk
2. Creates a bootable disk image with GNU MBR stage1
3. Partition the disk and create a file system
4. Populate the file system with enough to boot the Linux kernel from the directory "MiniDist".
5. Installing GRUB (manually or automatically)
6. Boot on QEMU

Try to get a general feel of these steps. Overtime, the goal is about making sense of these steps and learning the required skills and necessary tools to achieve them.

A word about safety:

Make sure that you do regular backups of your Linux install.

Be mindful and rigorous... shell scripting is potentially harmful.

Below is an example of a a simple typo mistake that is fatal:

```
DIR="~john/M2M/tmp"
rm -rf $(DOR)/
```

The script above will result in wiping out your entire disk!!!! Bye-bye your Linux and your data!

2 QEMU

The emulated platform is a regular Intel-based Personal Computer (PC).

You will notice that we use "*qemu-system-i386*".

You will notice that a new window pops up. This new window is the "screen and keyboard" of the PC. Indeed, as you know, a PC has a screen and a keyboard, always.

The terminal window in which you launched QEMU remains the terminal connected via a serial line, like before when we worked with the Versatile-PB board.

FYI: Keyboard Emulation with QEMU:

You can ask QEMU to emulate a specific keyboard. Indeed, often, French users have an AZERTY keyboard and GRUB behaves as it is QWERTY.

AZERTY keyboard:

```
$ qemu-system-i386 -k fr -m 256 -serial stdio -drive format=raw,file=disk.img
```

QWERTY keyboard:

```
# qemu-system-i386 -k en-us -m 256 -serial stdio -drive format=raw,file=disk.img
```

3 Minimal File-System Layout

Look at the script "mkdist.sh" and figure out the rôle of the directory "MiniDist".

4 GRUB Bootloader

When launching the script "mkdist.sh", you get a GRUB menu on the screen of the emulated PC.

Look at the script "mkdist.sh" and figure out how GRUB is installed on the created virtual disk for the emulated PC. You have been given the GRUB manual in the directory "grub-0.97", there are a few sections on installing GRUB.

The GRUB menu is in MiniDist/boot/grub/menu.lst

Look in that menu at the two boot options, with or without the loading of an initial ramdisk:

1. Linux 4.4 - Initial RamDisk Boot
2. Linux 4.4. - HardDisk Boot

Explain the related options given to the Linux kernel:

root=/dev/ram rdinit=/hello

root=/dev/sda1 init=/hello

Look at this init process called hello and discuss the way we compiled and linked this executable “hello”. Also discuss what is the makefile target “initrd.hello” about. Look up in Google “Linux initrd” and “Linux initial ramdisk”.

5 Minimal File-System Layout

Under the directory "MiniDist", we have very little, essentially a directory "boot" containing the GRUB boot loader and two Linux kernels, with two distinct build configurations.

You can compare the two configurations, there are text files, using a tool like "meld" for instance. Look at CONFIG_PRINTK, see it is turned on in config-4.4.113-disk and not in config-4.4.113-ram. Now, reboot on QEMU the two kernels and notice how one is more verbose than the other?

Also look at CONFIG_BLK_DEV_RAM, see the support for a ramdisk is only included in the kernel vmlinuz-4.4.113-ram which is consistent with the GRUB menu configuration. Now look at CONFIG_MSDOS_PARTITION, CONFIG_IDE, CONFIG_BLK_DEV_SD, and CONFIG_BLK_DEV_RD... it is clear that one kernel has support for IDE/SCSI mass storage peripherals (hard disks) and not the other.

Note: you do not have to build your own kernel here, but it helps understand that the two kernels have been built with a different configuration that explains why one is suited to boot only an initial ramdisk while the other is suited to boot directly from the hard disk and does not support

the use of an initial ramdisk.

6 The "init" Process

Look into the folder “hello” to see how we build our own executable for the “init” process.

Look at the makefile and explain however how the "hello" executable is linked to become a standalone executable?

Explain why it is so big? Around 700KB...

Explain why it is necessary to do so? Knowing that we will boot a Linux kernel on an almost-empty file system, including the hello executable and the “/boot/” directory.

Look at the GRUB menu and at the corresponding "init" options passed to the kernel:

root=/dev/ram rdinit=/hello
root=/dev/sda1 init=/hello

Look rapidly at the code, it is an easy read. The hello program is a simple parot-like console that repeats whatever you type. Notice that you are back to regular programming here, unlike our previous experiments with bare-metal programming. Why? The reason is that you have the regular libC and a regular Linux kernel available.

To make sure you understood the compilation/linking/shared library, let's do a small modification here.

1. Compile the hello program normally, no more static linking of libraries
2. Check what libraries it now depends on (use ldd)
3. Copy the necessary libraries in the right place

Where can you find the needed libraries?

Could you use the ones from your development distribution, the one that you have installed on your personal machine? Well... we hope that you know you cannot: 64-bit vs 32-bit as a starter... and maybe not the same version of the kernel... and maybe not the same version of the toolchain, not configured the same way...

We gave you some libraries and binaries that are compatible in the directory "debian-initrd". Where did we find them? Well, the kernel is from a Debian distribution, so we opened up the initrd from that distribution, and it obviously contains compatible binaries and libraries.

Try out your work now, your recompiled hello program. Does it work with both boot options? It may, if you were thorough. Most probably, it runs only when booting directly from the disk and not when booting from the inital ramdisk. **Explain why?**

7 The Shell Setup

Let's configure our minimal distribution to have a regular shell on the console, that is, the screen of the PC.

7.1 The script "init"

So we will use the following script, as our "init" process:

```
#!/bin/sh
exec /bin/sh +m
```

This is possibly the smallest init shell script ever !

Put this script in the file MiniDist/init

Try it out...

Well?

Did you remember to update the GRUB menu, creating a new entry?

Did you make sure the "init" script is executable (rwxr-x-r-x)

Try it again, it should work now.

Still does not work? Well, look at the first line of the script: **#!/bin/sh**

Well, do you have a shell at /bin/sh ? Nope. In fact, do you have any commands available, right?

7.2 The shell

So we need a shell program... but wait, modern Linux distributions use something called busybox... we gave you a compatible version of it in the directory "debian-initrd".

So go ahead, research what busybox is about and install the core commands in your minimal distribution:

MiniDist/bin/busybox

MiniDist/bin/sh // as a symbolic link to busybox

MiniDist/bin/ln // as a symbolic link to busybox

Try again to boot... It works, great!

Well, not so great since you cannot do much, can you? There are no commands available.

Why? Is it a path problem, well try it, once booted, print the environment variable PATH:

```
 / # echo $PATH
/sbin:/usr/sbin:/bin:/usr/bin
```

So it is not a PATH problem, the path is quite fine. It is just we do not have much commands under "/bin".

By the way, before we solved that problem, what is this weird prompt?

```
 / # cd bin
/bin # echo $PS1
\w \$
/ #
```

Well, you can change it by changing the environment variable:

```
 /bin # export PS1="$ "
 $
```

Or you can leave it alone, at least, you learned about the existence of the PS1 environment variable that controls the prompts of your shell, even in your regular Linux.

7.3 The basic commands

So let's get back at not having commands. Well? Do you know how to fix the problem?

1. Copy more executable under MiniDist/bin?
2. Create symbolic links?

If you picked (1), you missed the busybox thing entirely.

So now, you can create the symbolic links under MiniDist/bin... manually, or we can do it from the init script:

```
#!/bin/sh
LINKS="echo sleep ls mkdir rmdir rm mv cp cat more dmesg mknod"
for link in $LINKS ; do
```

```
  [ ! -e /bin/$link ] && ln -s /bin/busybox /bin/$link
done
sleep 2
echo "\n\nWelcome!\n"
exec /bin/sh +m
```

So? Oh well, did we forgot to say that the root file system is mounted read-only by the Linux kernel? Yep, it is...

7.4 Remounting the root file system

So we need to remount it read-write, like this:

```
echo "Remounting /dev/root in rw mode"
mount -o rw,remount /dev/root
```

So go ahead, add these two lines to your script "init".

Works?

Not really, right? Unless you remembered to check that the command "mount" is provided by busybox and that you created the right symbolic link. You are lucky, our busybox provides the command "mount".

Still not working? What is the error message telling you? The message is about a missing /proc/mounts. We discussed the special file system /proc. As a special file system, it is mounted with a special arguments to the command "mount":

```
mount -t proc none /proc
```

Now it should all work fine.

7.5 Device Files

Now that we have a read/write root file system, we can add some code to the script "init" to create the necessary device files:

```
mknod -m 666 /dev/ttyS0 c 4 64
mknod -m 666 /dev/tty0 c 4 0
mknod -m 666 /dev/tty1 c 4 1

mknod -m 666 /dev/null c 1 3
mknod -m 666 /dev/zero c 1 5

mknod -m 444 /dev/random c 1 8
mknod -m 444 /dev/urandom c 1 9

mknod -m 666 /dev/tty c 5 0
mknod -m 622 /dev/console c 5 1

mknod /dev/sda b 8 0
mknod /dev/sda1 b 8 1
```

Don't forget to do what is necessary for these lines to succeed and not fail.

7.6 The nano editor

Let's add a small editor, called "nano", to our small distribution. The binary is given in the directory "debian-initrd", it is an executable of its own right and not a symbolic link to busybox.

You should be able to do it on your own...

Note: since we did bare metal programming and you understood better the concept of a terminal and its relationship with the standard input/output of a C program, now you understand how the nano editor works by sending escape sequences to the terminal to move the cursor or change the colors.

Note: The same is true for programs like "top" or "htop" that show you the activity on your machine. Even better, you know now that these programs rely on /proc to obtain the information that they display.

7.7 The Console

Let's discuss the console, as the terminal used to interact with our Linux kernel. We use QEMU to emulate a Intel-based personal computer, which means QEMU emulates a screen and a keyboard, by default. This is why QEMU opens a new window named "QEMU" when launched.

So let's not confuse the two windows. One emulates the screen and keyboard devices of the emulated computer. The other, where you launch QEMU could be used as a terminal if QEMU was launch asking for a serial line on its standard input and output (-serial stdio).

That being clear, you notice that when booting the Linux kernel configured to boot directly from disk, it prints a lot of stuff but only the last 24 lines or so remain visible. Also notice that the kernel prints stuff later one, once our shell has started, garbling the output.

So let's try to work with the console option passed to the kernel by the boot loader. If you pass this option, nothing changes:

console=tty0

What happens if you pass this option instead?

console=ttyS0,115200n8

What happens if you pass this option instead?

console=ttyS0,115200n8 console=tty0

Let's try to separate the output from the kernel and the one from our init shell. The kernel would print to the serial line, while the shell would be on the screen. For this, we need to combine a change in the GRUB menu with a change to the script "init".

For the GRUB menu, we want to pass the following option to the kernel:

```
console=ttyS0,115200n8
```

So the console will remain on the serial line. If we do nothing, the script "init" will also use the console as input/output. In fact, we cannot change that for the script "init" itself, but at the end, when it executes another shell, we can set a different tty for execution, like this:

```
exec /bin/sh +m </dev/tty1 >/dev/tty1 2>&1
```