1 Linux Kernel

The latest kernel at the time of writing the lab was 3.18.8 which was released on February 27, 2015. There is most likely a new kernel at the time you are doing this but for consistancy please use the 3.18.8 kernel.

First, download and unzip the kernel by running these commands:

```
$ wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.18.8.tar.xz
$ wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.18.8.tar.sign
$ unxz linux-3.18.8.tar.xz
```

Next we should verify the kernel to make sure that it has not been tampered with. This is done using `gpg`. Run these commands:

```
$ gpg --verify linux-3.18.8.tar.sign
gpg: Signature made Thu 26 Feb 2015 07:50:04 PM CST using RSA key ID 6092693E
gpg: Cant check signature: No public key
```

If you see "Can't check signature" that means that you do not have the key installed. To install the key run this line where the key matches the key mentioned in the above line.

```
$ gpg --keyserver hkp://keys.gnupg.net --recv-keys 6092693E
$ gpg --verify linux-3.18.8.tar.sign
gpg: assuming signed data in 'linux-3.18.8.tar'
gpg: Signature made Thu 26 Feb 2015 07:50:04 PM CST using RSA key ID 6092693E
gpg: Good signature from "Greg Kroah-Hartman (Linux kernel stable release signing key) <greg@kroah.com>
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 647F 2865 4894 E3BD 4571  99BE 38DB BDC8 6092 693E
```

Ideally we would now contact that the listed person or go and talk to other kernel developers in the "Web of Trust" to verify that this is a valid signature, but for this class that is entirely unesseccary. We will assume that this signature is valid and move forward with the lab.

```
$ tar -xvf linux-3.18.8.tar
$ cd linux-3.18.8
$ ls
```

Now look around a little bit at the code. Contained in this folder is the entire source code of the Linux kernel and all Linux drivers that are included with it. Most of this is not important for this class, but let's go over a few of the more important directories.

```
$ ls -l
drwxr-xr-x  31 vens vens   4096 Feb 26 19:49 arch
drwxr-xr-x   3 vens vens   4096 Feb 26 19:49 block
-rw-r--r--   1 vens vens  18693 Feb 26 19:49 COPYING
-rw-r--r--   1 vens vens  96089 Feb 26 19:49 CREDITS
drwxr-xr-x   4 vens vens   4096 Feb 26 19:49 crypto
drwxr-xr-x 106 vens vens  12288 Feb 26 19:49 Documentation
drwxr-xr-x 120 vens vens   4096 Feb 26 19:49 drivers
drwxr-xr-x  36 vens vens   4096 Feb 26 19:49 firmware
drwxr-xr-x  75 vens vens   4096 Feb 26 19:49 fs
```

```
drwxr-xr-x  28 vens vens   4096 Feb 26 19:49 include
drwxr-xr-x   2 vens vens   4096 Feb 26 19:49 init
drwxr-xr-x   2 vens vens   4096 Feb 26 19:49 ipc
-rw-r--r--   1 vens vens   2536 Feb 26 19:49 Kbuild
-rw-r--r--   1 vens vens    252 Feb 26 19:49 Kconfig
drwxr-xr-x  15 vens vens   4096 Feb 26 19:49 kernel
drwxr-xr-x  11 vens vens  12288 Feb 26 19:49 lib
-rw-r--r--   1 vens vens 292390 Feb 26 19:49 MAINTAINERS
-rw-r--r--   1 vens vens  54431 Feb 26 19:49 Makefile
drwxr-xr-x   2 vens vens   4096 Feb 26 19:49 mm
drwxr-xr-x  58 vens vens   4096 Feb 26 19:49 net
-rw-r--r--   1 vens vens  18736 Feb 26 19:49 README
-rw-r--r--   1 vens vens   7485 Feb 26 19:49 REPORTING-BUGS
drwxr-xr-x  13 vens vens   4096 Feb 26 19:49 samples
drwxr-xr-x  13 vens vens   4096 Feb 26 19:49 scripts
drwxr-xr-x   9 vens vens   4096 Feb 26 19:49 security
drwxr-xr-x  22 vens vens   4096 Feb 26 19:49 sound
drwxr-xr-x  19 vens vens   4096 Feb 26 19:49 tools
drwxr-xr-x   2 vens vens   4096 Feb 26 19:49 usr
drwxr-xr-x   3 vens vens   4096 Feb 26 19:49 virt
```

The `arch` directory contains architecture specific code. In your report list **five** architectures that the Linux kernel supports. `Crypto` contains the code for cryptography used in security features built into the kernel. `Documentation` contains a number of example code and description of how to do kernel development and module development. `drivers` and `firmware` contain the code for hardware specific drivers and precompiled code for those drivers. `fs` contains the code for filesystems. In your report list **three** filesystem types Linux supports. `include` contains the header files which are included to interface with the kernel such as to use system calls or pthreads. `init` is the initialization and start up code that is used during the kernel boot sequence. `ipc` contains the code for interprocess comunication. `kernel` contains the actual kernel code. All of the core features of the kernel are contained in this directory. `net` contains all of the networking code. `samples` has example code and documentation. `scripts` contains helper scripts which are used during the build process along with some of the `tools`. And finally `virt` contains the virtualization code that allows one operating system to run inside of another.

2 Compile the kernel

Now that you have looked around the source code a little bit let's compile the kernel. The steps are fairly straight forward but if you fail to type a command exactly correct you may waste a lot of time or run into problems later on. Please read all instructions very carefully. (Note, you really can't break anything in this process so don't worry about that).

Because you don't have the privilages to install your own kernel on the lab machines we are going to compile the kernel for `user mode`. This is different from a virtualized kernel. Instead we are running the kernel as if it was another program on the system. This will be easier to debug with and test in lab. In the Linux kernel user mode is a architecture so we simple need to compile the kernel for that architecture by including `ARCH=um` in all of the make steps. First we will make a default configuration by running the following in the root directory of the kernel:

```
$ make defconfig ARCH=um
```

Now we need to configure the kernel. There are a number of tools built into the kernel to do this, but one of the easiest and most common is the ncurses GUI called menu config. To enter the confguration run the following

```
$ make menuconfig ARCH=um
```

The default configuration is mostly good enough for what we are doing, but there are a few things we might like to change. Use the arrow keys to move around, enter to go to submenues, and spacebar to select or deselect options. Exit returns up one level. Change the following options: (replace `NETID` with your NetID and `HOSTNAME` with whatever you would like)

- General Setup
  - Local Version: Set to `NETID`
  - Default Hostname: Set to `HOSTNAME`
- Enable loadable modules support
  - Forced module unloading: Enable

Now select `Save` to save the configuration and then `Exit`.

Now it is time to actually compile the kernel. For fun we can time the compile time using the `time` program. The output is a little cryptic; the `real` time is the time from when it started to when it finished, the `user` time is the time it spent in user mode, and `sys` is the time it spent making system calls. One thing we can do to greatly speed up the compile process is to use more than one thread to do the compilation. A good rule of thumb is a couple more threads than the number of cores the CPU has. The lab machines have 8 core processors so we will compile with 10 threads using the `-j` flag. The compile command is thus

```
$ time make -j10 ARCH=um
```

3 Filesystem

You now have a compiled kernel but to run it you need a file system. For this lab we will use the Debian Jessie image. Note that at the end of the lab is a number of other file systems that you are welcome to try out if you are so inclined. All of them are `bz2` and have the `md5` checksum by appending `.md5`. First return to the `Lab7` directory and then run the following lines.

```
$ wget http://fs.devloop.org.uk/filesystems/Debian-Jessie/Debian-Jessie-AMD64-root_fs.bz2
$ wget http://fs.devloop.org.uk/filesystems/Debian-Jessie/Debian-Jessie-AMD64-root_fs.bz2.md5
```

Next we want to check that the download's checksum to make sure the download was successful. To do this, run the following command:

```
$ md5sum -c Debian-Jessie-AMD64-root_fs.bz2.md5
Debian-Jessie-AMD64-root_fs.bz2: OK
```

You should see the `OK` indicating that the checksum passed. Next we can uncompress the image using `bzip2`:

```
$ bzip2 -d Debian-Jessie-AMD64-root_fs.bz2
Debian-Jessie-AMD64-root_fs.bz2: done
```

4 Running The Kernel

So you now have a root partition and a compiled kernel. It is now time to boot it! Go to the `Lab7` directory and run the following command:

```
$ ./linux/linux ubda=Debian-Jessie-AMD64-root_fs mem=512M
```

You will see a lot of kernel boot messages followed by a login request. To login type `root` as the user name. Now you can use the normal `ls` and `cd` to look around. You should notice that you are not on the host file system but instead you are inside the Debian filesystem. Note that you have root permissions on this user mode kernel which gives you unlimited control. This is shown by the the prompt ending with a `#` instead of the `$`. You will be able to tell which system to type commands into by checking if the command starts with `#` or `$`.

4.1 Mount Host Filesystem

We now want to be able to access the UDrive from within the usermode linux kernel so we can write our own driver. To do that we need to mount the host file system inside the user mode filesystem. This is accomplished by running the following commands changing `NETID` to **your** NetID.

```
# cd /
# mkdir /host
# mount none /host -t hostfs -o /home/NETID
# cd /host
# ls
```

If this worked correctly you should now see the contents of your home drive in the `/host` directory. The problem is that it is only mounted for this boot and if we restarted it would no longer be mounted. In order to automount at each boot we need to add it to the `fstab` table. To edit this table run `nano /etc/fstab` and add the last line to make it match this:

```
LABEL=ROOT      /              auto    defaults               1   1
#none          /dev/pts       devpts  gid=5,mode=620         0   0
none           /proc          proc    defaults               0   0
tmpfs          /tmp           tmpfs   defaults,size=768M     0   0
none           /host          hostfs  defaults,/home/NETID   0   0
```

Now each time you boot into usermode linux it will mount your UDrive onto `/host`.

Whenever you want to shutdown the usermode linux run the command `halt` in the usermode linux terminal.

5 Kernel Modules

The linux kernel by itself doesn't do very much. Most of the interesting stuff happens inside kernel modules which are drivers or other low-level pieces of software. This lab just scratches the surface on how to do kernel module development. If you are interested an want to learn more a great resource is http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf.

5.1 Hello World

As is the tradition among all software developers we shall start with the sacred hello world program. On the host machine go into the `hello-world` directory and look at `hello_world.c`. Change the author to yourself and change it to printing out something interesting in the `init` and `exit` functions. Note that there is no comma after the `KERN_INFO` macro. Now we can compile the kernel module. A makefile has been provided to compile the module for you. In the host terminal run the command

```
$ make ARCH=um
```

You shouldn't see any errors or warnings if everything is correct. Now let's test it out! On the usermode terminal `cd` into the `hello-world` directory in the `/host` filetree. In that directory run the following commands and include their outputs in your lab report.

```
# lsmod
# modinfo hello_world.ko
# insmod hello_world.ko
# lsmod
# rmmod hello_world.ko
# lsmod
```

5.2 File Example

Now that you have a little bit of experience with loading and unloading a custom made module it is time to do something more interesting than just printing hello world. In UNIX *everything* is a file where the kernel is concerned. For example, to write data to a serial port is just writing the data to the file `/dev/USB0`. There are two types of devices in Linux: block devices and character devices. Block devices are things like filesystems which buffer a lot of data and write it all at once whereas character devices handle the data as it comes in. For this lab we will limit ourselves to character devices only. Go to the `hello-file` directory in the `Lab7` directory and open `hello_file.c`.

The first thing you should notice about this file is the line `static struct file_operations fops`. This structure defines how the driver should handle operations on the file associated with it. It contains a number of function pointers which are called during a system call on that file. For example, when the file is read from the function pointed to by the `.read` element is invoked. The prototype of this function is `static ssize_t read(struct file *file, char *buffer, size_t length, loff_t *offset)`. You should notice the similarity of this prototype to the system call `read` which has prototype `ssize_t read(int fd, void *buf, size_t count);`. In your lab report comment on why the prototypes are not identical and what the extra paramaters are for (use your intuition to help answer this question).

The `struct file_operations` has quite a few operations that can be supported; however, in `hello_file.c` only a few are being used. Any function pointers not assigned are treated as invalid operations on the file and will return errors. The entire `struct file_operations` can be found in the kernel source tree under `include/linux/fs.h` at line 1486. In your report list six operations that files can support.

Before running the module please read through the code and make sure you understand what it should do. In your report give your expected output when the file is opened, written to, read from, and closed. Now in the directory run `make ARCH=um` to compile the module. From the um kernel `cd` into the `hello-file` directory and run `insmod hello_file.ko`. In your lab report include the major number of the module. In your lab report mention the output of each of these:

```
# mknod /dev/cpre308-0 c 254 0
# cat /dev/cpre308-0
# cat /dev/cpre308-0
```

```
# echo "hello" > /dev/cpre308
# mknod /dev/cpre308-1 c 254 1
# cat /dev/cpre308-0
# cat /dev/cpre308-1
# rmmod hello_file.ko
# cat /dev/cpre308-0
```

5.3 Printer Module

Now we can write a kernel module for the print server you have been working on for the last few labs. Create a new directory in `Lab7` called `printer-driver`. In that folder create a new file called `printer_driver.c`. This driver should be a character device loaded at `/dev/printer-n` where `n` is a consecutive number for each printer of this type installed. The driver should only support being written to and should return error if a user tries to read from it. When data (the post script data) is written to the file it will be of the following format: the first line is the print job name and starting from the second line to the end is the post script data. You should use `kprint` to print a meassage that the job has been printed. You should then just discard the rest of the data (unless you really want to send it on to a real printer). To test you can create a test script file such as this:

```
Print Job Name
Lorem ipsum dolor sit amet, mel ea wisi augue appareat,
eu assentior comprehensam sea, primis graeco molestiae ex
mei. Cum noster insolens no, tantas platonem scriptorem sed
ne, sit in graeco nominavi persecuti. Sed denique copiosae
cu, has persius ornatus salutandi et. Quo propriae adolescens ut.
```

And send this to the driver using `cat test_file > /dev/printer-0`.

6 Extra credit

Create another backend driver in the Lab 5 / Lab 6 code that sends the print job to the Linux kernel module created in this lab in addition to the pdf printer.

7 Conclusion

With the extra credit included you will have developed an almost complete print server application which clients can use the server by including a single `.h` file and linking against a single shared library. When they print a file that job is sent to the server running as a system daemon. The daemon chooses which printer it can send the job to and handles getting it to that printer. The job is sent to the kernel through a character device and in theory from the kernel it could easily be transmitted over a USB port or network to a physical printer. Note that this is not just a made up lab example. Although somewhat simplified, this is in fact how many systems actually work on Linux including `CUPS` print server, `Wayland` display server, `X11` display server, `Apache2` web server, `DBUS` IPC protocol, and many other Linux programs.

8 Apendix A: Other Filesystems

- BusyBox: http://fs.devloop.org.uk/filesystems/BusyBox-1.13.2/BusyBox-1.13.2-amd64-root_fs.bz2
- Damn Small Linux: http://fs.devloop.org.uk/filesystems/DSL-4.4/DSL-4.4-root_fs.bz2

- Fedora: http://fs.devloop.org.uk/filesystems/Fedora20/Fedora20-AMD64-root_fs.bz2
- CentOS: http://fs.devloop.org.uk/filesystems/CentOS-6.x/CentOS6.x-AMD64-root_fs.bz2
- OpenSuse: http://fs.devloop.org.uk/filesystems/OpenSUSE-12.1/OpenSuse-12.1-amd64-root_fs.bz2

9 License