

CSE-381: Systems 2

Exercise #12

Max Points: 20

Objective: The objective of this exercise is to explore the use of:

1. Gain some familiarity with a hypervisor (qemu)
2. Create a custom OS and run in in a virtual machine

Submission: Save this MS-Word document to your lab computer prior to proceeding with this exercise. Upload the following at the end of the lab exercise:

1. This document saved as a PDF with the convention `MUId_Exercise12.pdf` (Where `MUId` is your Miami University unique ID)
2. Your modified version of the startup script for your custom OS distribution in `etc/init.d/rcS`.

You may discuss any questions you may have with your instructor.



Wait for your instructor to review the concepts around `qemu` and briefly introduce some of the software tools that will be used in this exercise.

Name: **Noah Dunn**

Part #1: Creating a custom OS [20 points]

Background: Many modern operating systems are often created by combining the core set of GNU (acronym for GNU is not Unix) tools (namely: `gcc`, `binutils`, `emacs`, to name a few) along with the Linux kernel. Such GNU/Linux operating system distributions include: Fedora, Ubuntu, Mint, and Android. These distributions are created using the following two major components:

- i. The core GNU tools (<http://www.gnu.org/software/software.html>)
- ii. The Linux kernel (obtained from <https://www.kernel.org/>) **compiled using GNU tools**

The core set of components remains the same in the various distributions. However, the differences arise in the way other software (such as: open office, Java, Eclipse, chrome, etc.) are packaged, distributed, installed, and managed by the various distributions. One of the most conspicuous differences is usually in the graphical Desktop and window management systems supported by the various distributions. Of course Wikipedia has a nice page about Linux distributions at http://en.wikipedia.org/wiki/Linux_distribution.

In this exercise, you will be creating a Linux distribution of your own using `qemu` and `busybox`:

- i. **QEMU** (<http://www.qemu.org/>): QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run an OS and underlying programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance. When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU.
- ii. **Busybox**: BusyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts. BusyBox provides a fairly complete environment for any small or embedded system.
- iii. **Prebuilt Linux kernel**: You may download a custom kernel and compile it. However, kernel compilation is a time consuming process. Therefore, to save time, a prebuilt Linux kernel will be used for this lab exercise.

Setting up

1. **Note:** you will be performing this lab exercise on `osl.csi.miamioh.edu` Linux server.
2. First create a working directory for this exercise as shown below:

```
$ mkdir ex12  
$ cd ex12
```
3. Download the supplied initial `initramfs.zip` file to the above directory. The zip file that has the initial Linux directory structure setup for you for convenience. The zip file also includes very minimal configuration and `busybox` (a single executable that serves for running multiple commands).
4. Copy the zip file to your `ex12` work directory on the Linux server.
5. Unzip the supplied zip file to create the basic directory structure to be included in the initial ram disk (`initramfs`):

```
$ unzip initramfs.zip
```

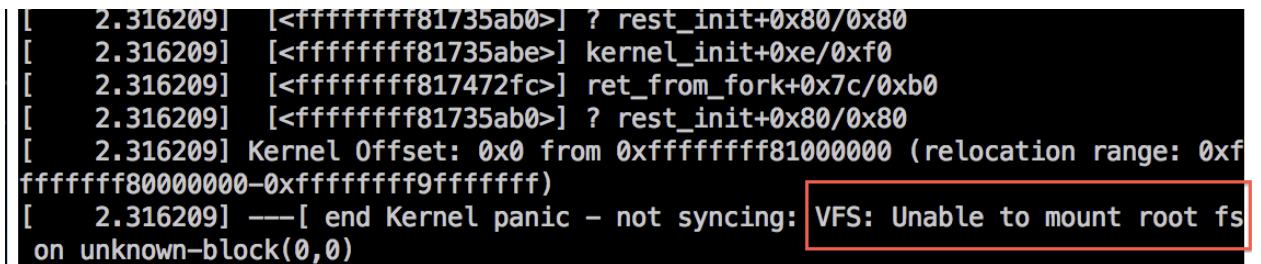
Initial run

Run the bare bones Linux kernel (it is not an operating system but just the core kernel) using the following command:

```
$ qemu-system-x86_64 -display curses -k en-us -no-reboot -no-kvm -show-cursor -kernel vmlinuz-3.17.7-300.fc21.x86_64
```

The above call to `qemu` will create a virtual machine and you should observe the following:

- i. A virtual machine will start running (using a default BIOS provided by `qemu`).
- ii. Next, the `qemu` boot loader will load the specified kernel.
- iii. The kernel will boot up.
- iv. At the end of the booting process the kernel will panic and crash (generating a "call trace") as shown below. The kernel is essentially complaining that it did not find a root file system. This is expected as we have not specified one for the kernel to mount and work with.



```
[ 2.316209] [<ffffffff81735ab0>] ? rest_init+0x80/0x80
[ 2.316209] [<ffffffff81735abe>] kernel_init+0xe/0xf0
[ 2.316209] [<ffffffff817472fc>] ret_from_fork+0x7c/0xb0
[ 2.316209] [<ffffffff81735ab0>] ? rest_init+0x80/0x80
[ 2.316209] Kernel Offset: 0x0 from 0xffffffff81000000 (relocation range: 0xffffffff80000000-0xffffffff9fffffff)
[ 2.316209] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
```

In the above screenshot what does VFS stand for? What role does it play in the Linux kernel?

Virtual File System. This acts as the “middle-man” between the kernel and the physical file system, allowing us to carry out storage and retrieval operations.

- v. After a kernel panic, the kernel essentially hangs. The only thing you can do now is to turn off the VM. Since there is no power button for you to push, you need to do this via `qemu`'s command. console Switch to `qemu`'s command console by pressing **Escape** and then **2** (number key). Type the command `quit` to exit the hypervisor (thereby stopping the VM)

Creating a root file system aka `initrd`

Estimated time to complete: 15 minutes

As observed in the previous part of this exercise, the Linux kernel requires a valid root file system to operate with. In this part of the exercise a minimal initial file system called `initrd` a.k.a `initramfs` will be created. In order to aid creating the basic set of directories that the Linux kernel requires in `initramfs`, a zip file containing the directory structure is supplied for this exercise. Use the supplied zip file using the following commands:

- i. View the directory structure in `initramfs` and observe how the file named `busybox` has been symbolically linked to various commands including the `init` program. Busybox is a versatile static binary that provides simple implementations for many standard commands such as: `ls`, `ps`, `cat`, `less`, `grep`, etc. The supplied `initramfs` uses `busybox` as the `init` executable required by Linux kernel using a symbolic link (`busybox` uses the file name to detect how it should behave). Recollect that `init` is the first process that is ever run by the Linux kernel.
- ii. Now, create your own initial ram file system (`initramfs`) using the following command where `MUId` is your unique ID (**ensure you are in the `initramfs` directory**):

```
$ cd initramfs
$ find . -print0 | cpio --null -ov --format=newc > ../initramfs.cpio
$ cd ..
```

- iii. Finally run the kernel with the newly created `initramfs` using the command below from the directory where the `initramfs.cpio` file is present. The kernel should start up and drop you to the root-shell prompt `#`.

```
$ qemu-system-x86_64 -display curses -k en-us -no-reboot -no-kvm -show-cursor -kernel vmlinuz-3.17.7-300.fc21.x86_64 -initrd initramfs.cpio
```
- iv. Note that you are root (or super user) on the virtual machine. You can try a few of the standard shell commands at the `#` prompt. In order to quit your virtual machine type `reboot` at the `#` prompt.

Customizing your `initrd`

Estimated time to complete: 15 minutes

The `initramfs` supplied to you is a barebones distribution that has very few commands. However, you can create additional commands by creating symbolic links to `busybox` as suggested below:

- i. Ensure your Type-2 hypervisor (aka `qemu` process) is not running.
- ii. Change current working directory to the appropriate **bin directory in your `initramfs`** directory.

- iii. Run `ls -l (ell)` to see how the symbolic links (aka symlinks) are displayed by `ls`. After you create additional symlinks they should be listed in a similar manner by `ls`.

- iv. Additional symlinks (or soft links) can be created using the command (ensure you are in your `initramfs/bin` directory) where *CmdName* is some command's name:

```
$ ln -s busybox CmdName
```

Using the above command, create symlinks for the following commands: `clear`, `grep`, `ps`, `top`, `adduser`, `addgroup`, `cal`, `echo`, `mkfs.ext2`, `rpm`, `wc`, `who`, `vi`.

Use `ls -l (ell)` to ensure that the symbolic links have been successfully created.

- v. Setup a custom port to be used later on in this exercise. **You will have to find a port that is unused via trial-and-error**, in case the command below generates any errors:

```
$ export PORT_NUM=4000
```

- vi. To save you some time and running `qemu` use supplied `run_qemu.sh` script to create the cpio archive and run `qemu`

```
$ ./run_qemu.sh
```

- vii. The VM should start up and drop you to the root-shell prompt `#`. Type some of the commands that you created (such as: `wc`, `who`, `cal`, etc.) in your OS distribution and ensure it works correctly. Use the `reboot` command to terminate your virtual machine.

Customizing your OS Startup

Estimated time to complete: 15 minutes

Now it is time to add a bit of flair to your OS distribution to print a startup banner for your distribution in the following manner:

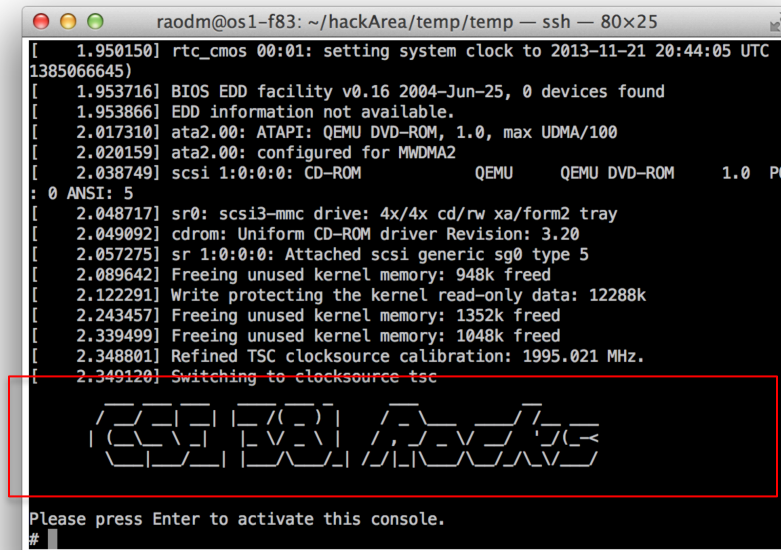
- i. Ensure your virtual machine (aka `qemu` process) **is not running**.
- ii. In the `initramfs` directory, locate the `etc/init.d/rcS` file. This file is essentially a shell script that is automatically run by `busybox's init` program present in the root directory of your custom file system.
- iii. Edit the `etc/init.d/rcS` using `nano` and add code to print messages to the end of this file as shown below (watch out for messages for single quotes in them – they will cause problems if you don't replace single quotes with some other character):

```
echo 'your message goes here'
```

- iv. Use an `echo` statement to print your MU login id. **If you have time at the end of the**

lab: Instead of a simple echo statement, use a series of echo statements to print your MU login id as ASCII art banner that can be generated online at: <http://patorjk.com/software/taag/>. Here is an example banner (If you are creative share your creativity with your instructor):

- v. You can also add some color to the banner being printed. See examples at: https://misc.flogisoft.com/bash/tip_colors_and_formatting



The screenshot shows a terminal window titled 'raodm@os1-f83: ~/hackArea/temp/temp — ssh — 80x25'. The terminal displays various system boot messages, including BIOS EDD facility information, ATA and SCSI device detection, and kernel memory freeing. A red rectangle highlights the bottom of the terminal output, which contains an ASCII art banner for 'raodm' and the prompt '#'. Below the banner, it says 'Please press Enter to activate this console.'

- vi. Now, recreate run your custom OS distribution using `run_qemu.sh` script and verify that your banner is correctly generated.
- vii. Next make a screenshot of your banner and paste it in the space below:

[illegible]

Part #2: Running a custom web-server inside the VM

Estimated time to complete: 15 minutes

Background: In this course **we developed our own custom multithreaded web-server in C++ in a lab exercise**. C++ programs are standalone in that they can be compiled to run standalone without any additional libraries. The solution for Exercise #8 has been compiled and already included in the zip file for this exercise. Recollect that this web-server can run commands (specified as part of the URL) and return the output of the command back to the web-browser

Exercise:

- i. Ensure your virtual machine (aka qemu process) is running (via `./run_qemu.sh` script). Remember the value you have set for `PORT_NUM`.
- ii. In your VM, run the web-server developed as part of this course via the following command (yes, do use port 80):

```
# /bin/WebServer 80
```
- iii. From your web-browser, try running different Linux commands. You will need to change the port number to match the value you set for `PORT_NUM` (instead of 4000). A few are suggested below. You should observe the web-server running the commands.
 - `http://osl.csi.miamioh.edu:4000/ls`
 - `http://osl.csi.miamioh.edu:4000/uname -a`
 - `http://osl.csi.miamioh.edu:4000/reboot`

Part #3: Submit files to Canvas

Upload just the following files:

1. This document saved as a PDF file with the convention `MUId_Exercise12.pdf`.
2. Your modified version of the startup script in `etc/init.d/rcS` **renamed as `rcS.txt`**