Project 1: Virtual Machine and Linux

CSE 330: Operating Systems

1. Summary

The first project is a warm-up exercise which helps you become familiar with the virtual machine (VM) and Linux software that you will use extensively for your kernel development throughout this semester. In this exercise you need to create a new VM using VirtualBox, install a Linux OS on the VM, install the latest kernel in this Linux system, and add a new module and system call to the kernel.

2. Description

Note: The references included in the following description contain only *generic* instructions for completing the tasks of this project; they are *by no means* exact step-by-step instructions. To get more help, attend the lectures, ask the instructor and TAs, and use your favorite search engine (there are lots of useful materials on the web!)

Step 1: Create a new VM in VirtualBox.

We will use VMs extensively throughout the projects of this course. They make our lives much easier as kernel developers. We can conveniently test our new kernels without crashing the physical machines and take VM snapshots to save the progress of our work. In addition to completing this step as required in the project, I encourage you to play with your VM and get familiar with these useful features.

Notes:

1) You can use either the VirtualBox in the instructional labs and install it on your own computer (it is free). You can store it on a USB drive to make it portable.
2) We will work on 64bit kernel in our projects, so make your VM *64bit* too.
3) If your computer has more than one core, give your VM more than one too so it can be faster when compiling a new kernel.
4) Give you VM more than 20GB of virtual disk because building a new kernel requires a lot of space. But use dynamic allocation so that its actual storage usage grows as needed.

Reference: "VirtualBox End-user Documentation", URL: https://www.virtualbox.org/wiki/End-user_documentation

Step 2: Install Ubuntu 19 on your new VM.

Ubuntu is one of the major GNU/Linux distributions and it is quite user friendly. In this step, you will install Ubuntu on your new VMs which will be used for all the labs of this course. (LTS means that this version will be supported almost forever.) To do this you need to download the latest .iso file and provide the iso in the CD disk image property of the VM.

References: "Ubuntu Desktop Guide", URL:

https://help.ubuntu.com/lts/ubuntu-help/index.html

Step 3: Compile and install a new kernel on your new Ubuntu.

In this step, you will upgrade the kernel in your VM to the latest stable version which will be used as the basis for all your kernel development this semester.

Basic instructions:

1) Download the latest long-term stable kernel from http://www.kernel.org. (Linux gets updated on a daily basis. So don't be surprised if it is a different version for everybody.)
2) Uncompress the file into a folder.
3) Configure the modules that you want to add or remove. For this you can use a utility called *menuconfig* by typing the command *make menuconfig*. You may need to install the necessary dependencies such as *gcc* and *libncurses5-dev*. Hint: In Ubuntu, you can use *apt-get* to install a new package.
4) For grading purpose, attach a unique local version to your kernel. For example, if your name is Dora Marquez, your local version string should be "DoraMarquez". It can be set using *menuconfig*, in "General setup" -> "Local version".
5) Build the kernel by executing the command *make bzImage*.
6) Build the modules by executing the command *make modules* (this may take a long time when you do it for the first time).
7) Install the new kernel modules by executing the command *make modules_install*.
8) Install the new kernel image by executing the command *make install*.
9) Configure the GRUB boot-loader so it can load the new kernel image that you just built. By running *update-grub* command GRUB should find all the bootable kernels and add them to the GRUB menu.
10) Restart Linux and select the new kernel's entry from the GRUB menu list. By default the GRUB menu will not be shown; to see it, in */etc/default/grub*, change GRUB_TIMEOUT=-1 and run *update*-grub again; then when you reboot, you should see the menu.
11) Upon next login, check the current running kernel version using "*uname -r*" and verify that you are indeed using your new kernel.

References: Linux boot process

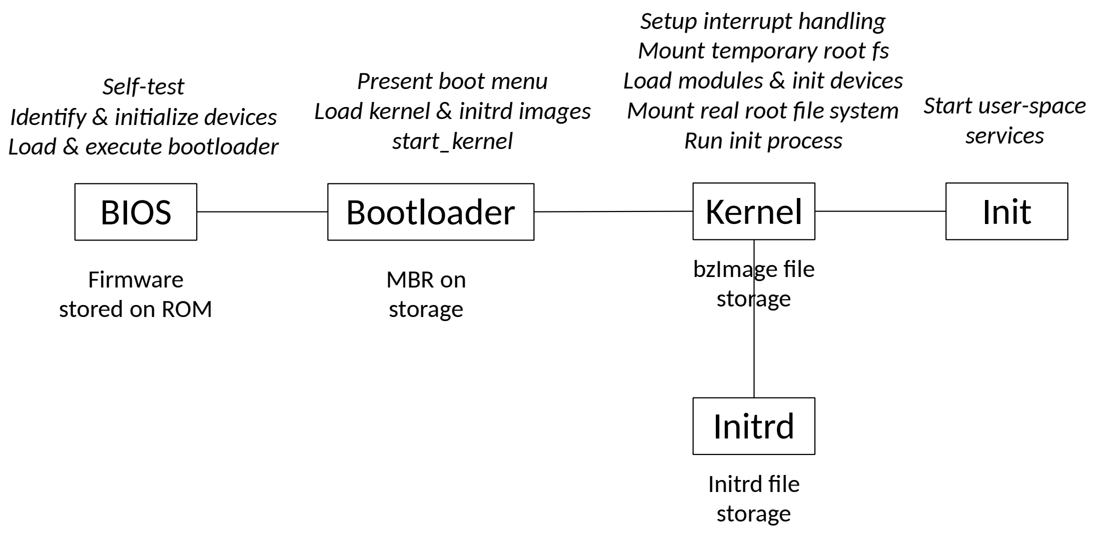

Figure 1. Linux boot process

Hints:

1) Building a kernel from scratch is quite time consuming, so do it on a good computer when you compile it for the first time.
2) You should finish the above three steps in *a week* or so. The following two steps are more challenging and should consume more time.
3) Command-line tools are important for kernel development. For example, the above Step 3 requires the use of a variety of command-line tools. To help you become familiar with the command-line interface, try to do everything from a shell terminal, instead of using the GUI. If you don't know what command to use, the quickest way to find is through a search engine. If you don't understand the usage of a command, the easiest way is to run "*man*" followed by the command name.
   There are many Linux command-line references on the Web. Here is a good one: http://portal.aauj.edu/e_books/linux_complete_command_reference.pdf
4) Some steps mentioned above require administrative privileges, e.g., using *apt-get* to install a missing package and using *make install* to install the new kernel image. When you run these commands as a regular user, you need to add "*sudo*" in front of the command, which gives you the necessary privileges to run the command (assuming you are a *sudo* user; check the man page of sudo for more information).
5) Kernel compilation is quite time consuming, so do it on a good computer and store your VM on a fast drive when you do it for the first time. You can also speed it up using parallel compilation. But first you need to change your VM's setting to use more CPUs. Then when you are running the make command, you can follow it by a "-j" to enable parallel compilation. Run "*man make*" and you can find the explanation of this option.

Step 4: Implement a new system call for your new kernel.

As we will also discuss in class, system calls are the main interface for user-space software to interact with the kernel. In this step, you will implement a new system call that simply prints out one message in kernel log and a user-space program that tests this system call.

Notes:

1) Name your new system call as *my_syscall*. When it is called, it should print to kernel log "This is the new system call [FULLNAME] implemented." Replace [FULLNAME] with your full name. Remember to reboot after recompiling the kernel with *make* and *make install* commands.
2) To test your system call, write a user-space program to invoke your *my_syscall* system call so that you can check whether the required message indeed appears in the kernel log.

Steps to define system call

1. Create a C source file (for you, `my_syscall.c` in the `kernel` sub-directory of the `linux` source tree) so:
   - It has near the top `#include <linux/syscalls.h>` and other `#include`s for needed headers so you can call kernel functions like `printk`

- It uses one of the **SYSCALL_DEFINE**... macros to generate the declaration of your system call service routine

  The definitions of these **SYSCALL_DEFINE**... macros are in **include/linux/syscalls.h**. Hence, the `.c` file in which you code the body of your syscall's service routine must **#include <linux/syscalls.h>**

- It has within **{ ... }** (after your **SYSCALL_DEFINE...(...)**) the code (you will write!) of the body of to be run when your syscall is called
- In **kernel/Makefile**, you add the name of one more object file (**csi500.o**) in the list of object files that this **Makefile** directs the kbuild to build and link into the fixed kernel.

  Study the contents of **kernel/Makefile** and figure out how you need to modify it so **csi500.c** is compiled into **csi500.o** during the kernel build. Hint: Just add one entry to the end of one line!

Here is an expecially simplistic system call:

```
#include <linux/syscalls.h>
#include <linux/printk.h>
SYSCALL_DEFINE(my_syscall)
{
   printk(KERN_EMERG "Hello from John Doe \n");

   return 0;
}
```

1. Make sure to use **SYSCALL_DEFINE** properly. Tip: Look for examples within the kernel code, such as in kernel/groups.c

   (It is not what you would think of first! There are ***extra commas***.).

2. Here is the command to make kbuild just try to compile your my_syscall.c file, so you will quickly find out about syntax and other compile time errors:

   If necessary, **cd ~/kernel/your linux version** You MUST be at the top of the kernel source tree, even though the Makefile and my_syscall.c you edited were in the ~/kernel/your kernel version/kernel subdirectory.

   Command: **make kernel/my_syscall.o**

   FIX AND TRY AGAIN if there are any error messages! Don't go on.

3. When building `my_syscall.o` was successful, spark off a full kernel build with: `time make -j 3`

   (`make -j 3` with the concurrent jobs option -j may speed things up.
   The `time` will tell you how long it took..)

4. You must of course, after (1) **building** your modified kernel (as done in the previous lab) you must

   (2) **copy** into `/boot` your modified kernel from the file
   named `bzImage` within `arch/x86/boot` and (3) **reboot** the virtual machine so it runs the your revised kernel.

   The source for the copying is `arch/x86/boot/bzImage` (of course, `arch` is under `~/kernel/your linux version`

   The destination for the copying is `/boot/linux-3.10.12csi500`

   Long-winded command that works from anywhere: `cp ~/kernel/your linux version/arch/x86/boot/bzImage /boot/yourlinux version`

   Write and compile an application program in into your virtual machine and run it to test your system call.

3. Submission requirements

You need to work on this project *individually*, i.e., every student needs to work on the project independently and turn in the assignment separately. Your submission should be a *single zip file* including only the following:

- The source code of your i) kernel module, ii) system call, and iii) user-space test program
- The screenshots of
  - the output of "*uname -r*" showing your name as part of the kernel version;
  - the kernel log showing the output of your system call

Notes:

- Do not submit any other source code
- Do not submit any binary
- Put everything in a single zip file named by your full name

4. Policies

1) Every student needs to *work independently* on this exercise. We encourage high-level discussions among students to help each other understand the concepts and principles. However, code-level discussion is prohibited and plagiarism will directly lead to failure of this course. We will use anti-plagiarism tools to detect violations of this policy.

Ubuntu 19

Updated instructions

Figure out the way to add syscall in syscall_64/32.tbl

```
No    64      mycall     sys_mycall
```

Do not skip unistd.h step

1. HOME/arch/x86/entry/syscalls/syscall_64.tbl

2. HOME/arch/x86/include/asm/unistd.h

3. HOME/arch/x86/include/asm/syscalls.h

4. HOME/Makefile

5. Kernel log file

/var/log/syslog