Programming Assignment 1 - Adding A System Call

CSCI 3753 - Operating Systems
University of Colorado at Boulder
Due Date: Friday, January 24, 2014 11:55pm

Spring 2014

1 Introduction

Welcome to the first programming assignment for CSCI 3753 - Operating Systems. In this assignment we will go over how to compile and install a modern Linux kernel, as well as how to add a custom system call. Most importantly you will gain skills and set up an environment that you will use in future assignments.

You will need the following software/files:

- VirtualBox.
- CSCI 3753 Spring 2014 Virtual Machine v1.1 or newer, along with the packages for the *cu-cs-csci3753* course. See `http://foundation.cs.colorado.edu/sde/` for Virtual Machine installation instructions.

All other software and files should be installed on the virtual machine image. It is important that you do not update the virtual machine image, except where noted in this assignment, as all the assignments are designed and written using the software versions that the VM is distributed with. Also make note that this assignment will require you to re-compile the kernel at least twice, and you can expect each compilation to take at least half an hour and up to 5 hours on older machines.

2 About the Virtual Machine

The virtual machine for this class is an Ubuntu 12.04 for the x86 architecture (64-bits). It has one user (*user*) with a password of *user*. In the home directory you will create a directory called *kernels* for storing kernel source code, and change the current directory to *kernels*, by running the command

```
mkdir kernels; cd kernels
```

To obtain a copy of the source code for the kernel we will be using, run the command

```
apt-get source linux-image-$(uname -r)
```

It is recommended that for each assignment you make a new copy of the source inside the *kernels* directory. This can be accomplished by running

```
cp -R linux-lts-raring-3.8.0 linux-lts-raring-3.8.0-CULogin-pa\#
```

To make things easier on everyone, please name your source trees `linux-lts-raring-3.8.0-CULogin-pa#` where *CULogin* is replaced with your CULogin and *pa#* is replaced with the current programming assignment number (as is done in the cp example above). Note that your CULogin must be in all lowercase characters. Someone named John Doe might name their source tree for Programming Assignment 1 `linux-lts-raring-3.8.0-doej-pa1`.

3 Assignment Steps

What follows are the steps you will need to go through to complete this assignment. They include:

- Configuring Grub
- Compiling a kernel
- Writing a system call
- Writing a test program that uses the system call

3.1 Configuring Grub

Grub is the boot loader installed with Ubuntu 12.04. It provides configuration options to boot from a list of different kernels available on the machine. By default Ubuntu 12.04 suppresses much of the boot process from users, as a result, we will need to update our Grub configuration to allow booting from multiple kernel versions and to recover from a corrupt installation. Perform the following:

1. From the command line, load the grub configuration file:

   ```
   sudo emacs /etc/default/grub
   ```

   (feel free to replace emacs with the editor of your choice).

2. Make the following changes to the configuration file:

   - Comment out:

     `GRUB_HIDDEN_TIMEOUT=0`

   - Comment out:

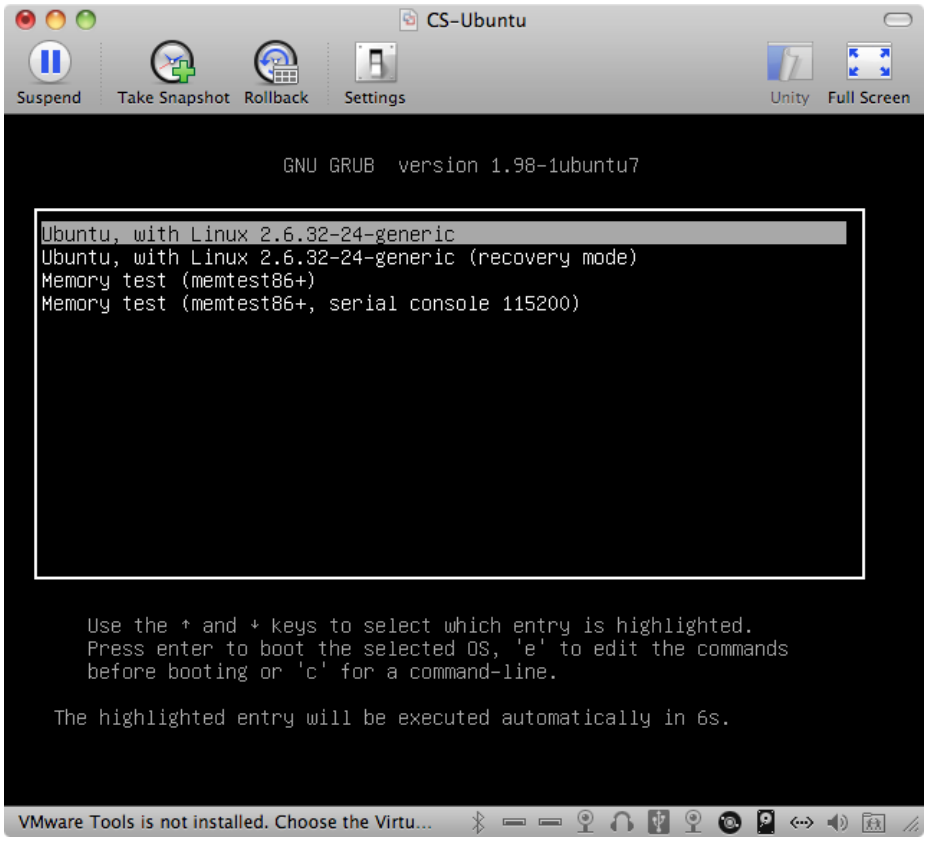

Figure 1: Linux Grub Boot Menu

`GRUB_HIDDEN_TIMEOUT_QUIET=true`

- Comment out:

  `GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"`

  and add the following new line directly below it:

  `GRUB_CMDLINE_LINUX_DEFAULT=""`

- Save updates

3. From the command line, update Grub: `sudo update-grub`.
4. Reboot your virtual machine and verify you see a boot menu similar to the one shown in Figure 1.

3.2 Compiling Your Kernel

Now that our boot loader has been updated, we are ready to compile a new kernel. Compiling a modern kernel is a complex issue that often involves many steps. Luckily for us people have been doing this for a while and have created build systems that automate many of these

tasks. We will be using the Debian build system that is employed by the Ubuntu community to simplify the entire process. Some of things that it does are:

- Compile the code
- Link the code
- Create an *initrd* (initial ramdisk) image for use when booting the system
- Pack the files into *.deb* packages

One you have a fresh copy of the kernel source use the `cd` command to enter the directory containing the source. The Linux kernel has many options for how it should be built. To see these options type `make menuconfig` and take a look around. Don't change anything, but explore some of the options.

Next we will issue the command that starts the Debian build system. Make sure to insert your *CULogin* and programming assignment numbers, as well as updating the revision number (included in the `--append-to-version` string) as you compile newer versions of your code. *Note: The following is a single command. Ignore the line breaks when entering it in the terminal.*

```
fakeroot make-kpkg --initrd
--append-to-version=-CULogin-pa1 --revision=1 kernel-image kernel-headers
```

You can now go and get a snack or something as this is going to take a while. Two to four hour builds are not uncommon. When it is finally done you will find two *.deb* files in the *~/kernels* directory. These contain the files that need to be installed for you to be able to use your kernel. To install them use the following commands.

```
sudo dpkg -i linux-image-<your version string>.deb
```

and

```
sudo dpkg -i linux-headers-<your version string>.deb
```

Those version strings will be different for every kernel that you compile (if you remember to update your programming assignment and revision numbers) so be careful with them.

If you are installing a kernel image for which a revision is already installed (i.e. when you install the modified kernel or any subsequent builds), you may receive a few warning messages alerting you to the fact that you will overwrite the currently installed kernel. This is normal. Bypass the warnings and reboot.

Now that your new kernel is installed you can reboot the VM and it will boot into your new kernel by selecting it from the GRUB menu. To make sure you are in the correct kernel use the '`uname -a`' command.

```
1  /* This file defines the HelloWorld system call */
2
3  #include <linux/kernel.h>
4  #include <linux/linkage.h>
5
6  asmlinkage int sys_helloworld(){
7    printk(KERN_EMERG "YOUR_CULOGIN says Hello World!\n");
8    return 0;
9  }
```

Listing 1: Code for HelloWorld system call

3.3 Adding A System Call

The first step in this process is to actually define the system call. For the purposes of this assignment we will only add the call for the x86 architecture. As such, the file containing our system call will reside in *arch/x86/kernel* and we will call it *helloworld.c*. Create this file and add the code shown in Listing 1.

What is going on here? Well, for starters we include two important files. The first file defines many constants and functions that are used in kernel hacking, including the function `printk`. The second file defines macros that are used to keep the stack safe and ordered during system calls. Next, we have the function prototype. Here we see the `asmlinkage` macro used, which is defined in *linux/linkage.h*. We have named the function `sys_helloworld` because all system calls begin with the `sys_` prefix. The function `printk` is used to print out kernel messages, and here we are using it with the `KERN_EMERG` macro to print out "Hello World!" as if it were a kernel emergency. Not that depending on your system settings, `printk` message may not print to your terminal. Most of the time they only get printed to the kernel syslog system. If the severity level is high enough, they will also print to the terminal. Look up `printk` online or in the kernel docs for more information.

Now that we have defined the system call we have to tell the build system about it. Open the file *arch/x86/kernel/Makefile*. At the end of this file, add the following line (do not place it inside any special control statements in the file):

```
obj-y += helloworld.o
```

This tells the build system that it now has to compile the object *helloworld.o* and link it with all the others.

Next, the kernel must be made aware of the new system call. The first file that needs to be modified is *arch/x86/syscalls/syscall_64.tbl*, which contains a long list of definitions that assign specific numbers to each system call. Find the end of the first list in this file (before the x-32 specific system calls) and, using the existing entries as a guide, add our new system call. Use the `common` abi for this system call, which means that this system call is for both the x32 and x86-64 Linux application binary interfaces (search the Web if you'd like more information). The *name* field (third field) in this new entry can be anything you like (such as "helloworld"). Keep track of the *system call number* assigned to our new call as it will be important later. Consult the kernel documentation or the Interwebs if you have trouble.

The next file that we need to modify is *include/linux/syscalls.h*. Near the end of this file, just before the last `#endif`, add a prototype for our system call:

```
1  /* kernel/helloworld.c */
2  asmlinkage int sys_helloworld();
```

3.4 Re-compiling Your Kernel

Now that we have modified the kernel we need to re-compile it and then install the new version. Before we do that, though, we need to clean up the previous build by running the command `make-kpkg clean`. This tells the build system to remove all the files generated during the previous build. Now you can use the same commands to build and install your kernel as before, remembering to increment the revision number by one (`--revision=2`).

3.5 Writing A Test Program

Lastly, you need to test your new system call. To do so you will write a program that uses the `syscall` function. The manual pages for `syscall` (which you can find by running the command '`man syscall`') contain all the information you need to write your test program, including the files you need to `#include`. The argument you need to pass `syscall` is the system call number that you saved from earlier. To check for the results of your `printk` call look in the file */var/log/syslog* (you may need root or sudo access), or run the `dmesg` command (see the dmesg manpage for more information).

3.6 Resources

Refer to the following list of URLs for additional information on Grub and compiling a kernel.

- Operating Systems Concepts Chapter 2
- https://help.ubuntu.com/community/Grub2
- https://help.ubuntu.com/community/Kernel/Compile
- http://kernelnewbies.org/
- *~/kernels/linux-lts-raring-3.8.0/Documentation*

4 Grading

As for all assignments for this course 60% of your grade will be assigned during the grading meeting that will take place after you turn in your work. During this meeting we will take a look at each of the files listed below, as well as booting into each of your new kernels. I will also ask you why each of the changes that we made were necessary.

The remaining 40% will be awarded on the following basis:

- 10% for compiling the first kernel
- 10% for adding the necessary components for the new system call
- 10% for compiling the second kernel with the new system call
- 10% for your test program that uses the new system call

The following files need to be submitted to the Moodle by the due date:

- *arch/x86/kernel/helloworld.c*
- *arch/x86/kernel/Makefile*
- *arch/x86/syscalls/syscall_64.tbl*
- *include/linux/syscalls.h*
- */var/log/syslog*
- Source code for your test program.
- A README with your contact information, information on what each file contains, information on where each file should be located in a standard build tree, instructions for building and running your test program, and any other pertinent info.

Combine your files into some form of archive (zip, tar, etc) and upload them to Moodle as a single file.

A Making Builds Faster

A.1 Enabling Multiple Cores

If your computer has more than one processor/core, you may configure the virtual machine to use two or more processing units. See `http://foundation.cs.colorado.edu/sde/vm-install.html` for instructions on configuring the VM to use multiple processors. You can modify the VM settings after the VM has been imported/installed by clicking on the "Settings" button in VirtualBox after the VM has been shutdown.

Now that you have made additional processing units available to the VM you need to tell the build system how to take advantage of them. To do this you must export a value to the variable `CONCURRENCY_LEVEL`. The command to do so is '`export CONCURRENCY_LEVEL=N`' where N is the number of jobs you wish the build system to execute at the same time. A good value for N is the number of processing unites + 1. For a dual core system this value would then be 3.

Now, when you use *make-kpkg* it will issue N jobs at a time. However if you close the terminal you were using and open a new one the value for 'CONCURRENCY_LEVEL' will be unset. If you want this value used for `CONCURRENCY_LEVEL` every time you will need to add the '`export CONCURRENCY_LEVEL=N`' command to the file *~/.bashrc* like so: '`echo export CONCURRENCY_LEVEL=N > ~/.bashrc`.'

A.2 Distributed Builds

Look into using *ccache* and *distcc*. This will take a fair amount of work and should only be considered by people who think themselves Linux system administrators.