Part A: Setup your kernel development environment

For compatibility please compile all your code on nml-cloud-205.cs.sfu.ca. (aka kernel-builder). You can ssh into this machine using your SFU auth.

QEMU Introduction

QEMU **(Q Emulator) is a processor emulator that began as a project by Fabrice Bellard. It is also Free software, can emulate many different CPUs, and it is stable and fast (QEMU relies on dynamic binary translation to achieve reasonable performance) QEMU is a full system emulator so it can give you access to the network and emulate your peripheral devices; we will be providing you with a file, which is the virtual disk image where Debian is installed.**

The bundle of QEMU as the system emulator, Debian GNU/Linux as the system, and the latest Linux kernel will be our testbed for Linux kernel development. We will be referring to it as QDGL from now on (QEMU - Debian GNU/Linux). We will be referring to your desktop or whatever machine you run QDGL on, as the host OS, and the virtualized Debian GNU/Linux as the guest OS.

QDGL: List of files you 'll need and how to acquire them

In this assignment, you are required to use QDGL to modify and test a Linux kernel.

First you need to copy QDGL to your directory. You will need ~400MB just to grab QDGL, and another ~600M as soon as you zip and compile the kernel.

Below is the list of the contents of QDGL:

- **llinux-2.6.26.5** : The Linux kernel source tree.
- **hda.img** : Debian's virtual disk image. Your disk is seen by the linux kernel as device /dev/hda - more about this below. **Do not touch this file**.
- (Example Scripts): **build-kernel.sh, run-mykernel.sh, run-base.sh,** Please look at these for hints on how to build/run your custom kernel.

All files can be downloaded from:

http://www.sfu.ca/~rws1/cmpt-300/assignments/cmpt300-a4.zip

QDGL: Test driving your system

To take a look at the default Debian installation, move into your copy of the QDGL directory (**cd QDGL**), and on your terminal run the following instruction:
qemu-system-i386 -m 64M -hda hda.img -nographic

Explanation:

- **qemu-system-i386**: execute the emulator binary
- **-m 64M** : tell the emulator that the emulated system will have a physical memory of 64 MB. Feel free to increase this number if your host can accept a larger value
- **-hda hda.img** : tell the emulator which file will act as the virtual hard disk image - this file should appear as device /dev/hda in the guest GNU/Linux OS
- **-nographic** : tell the emulator to use the ncurses library to draw its interface on your terminal

Alternatively, if you choose to use the SDL interface, you can simply neglect the last parameter (-nographic). If you have access to the X server of your Linux host, you should also be able to see a virtual monitor in a separate window (this option uses the SDL libraries, which you should already have installed) instead of the terminal.
This should boot Debian from the virtual disk, with the distribution kernel. You will be prompted by the boot loader to choose the kernel to boot (Press any key to continue) - you can safely just wait 5sec, which will start the default kernel, or you can pick it yourself in the grub menu. After the kernel log messages, you will be prompted with the login ; the system is called cmpt300. Below are the two accounts which already exist in the system

USER: root PASS:cmpt300
USER: cmpt300 PASS: cmpt300

You should not enter as root unless you absolutely need to.

Hitting the command **uname -a** you can check the system information -- you should see Debian's pre-installed kernel version (2.6.18-6).

You should always make sure you turn off your client properly ; the virtual disk is setup as a journaled ext3 filesystem, but properly shutting down is essential to keep it "clean". Another good habit would be to be taking backups of your virtual disk image at checkpoint events. To turn off your system safely, if logged in as root just type

poweroff

otherwise, if logged in as cmpt300, type

sudo poweroff

QDGL : Compiling and testing a new kernel

For the following we are assuming that you are working on a GNU/Linux environment. Back to the QDGL directory, and as soon as you have test-driven your default Debian box please compile your own kernel.

cd linux-2.6.26.5

To compile your kernel run this command(If asked a question just press enter).

make ARCH=i386 -j16

Your newly compiled kernel image is at arch/x86/boot/bzImage inside your kernel directory (i.e., the directory where you untarred the kernel sources: linux-2.6.26.5 by default and assumed from now on). To test drive your new kernel run

qemu-system-i386 -m 64M -hda hda.img -nographic -append "root=/dev/hda1 console=ttyS0,115200n8 console=tty0" -kernel linux-2.6.26.5/arch/i386/boot/bzImage

Explanation:

- **-append "root=/dev/hda1 console=ttyS0,115200n8 console=tty0"**: passes the parameter in quotes to the kernel. The kernel needs to know where the root directory will be to mount it and boot properly. The kernel also needs to know it is displayed on a serial console along with the console parameters if you choose to use a serial display.
- **-kernel linux-2.6.26.5/arch/i386/boot/bzImage**: tells the emulator where the external (to the image) kernel to boot lies (full or relative path)

Now if you login again and type **uname -a** you should see that the new kernel version is **2.6.26.5**

Congratulations, at this point you have managed to boot a kernel you compiled yourself !

Every time you make modifications to your linux kernel, you should rebuild it (**make**)

and test it (**./qemu/qemu ... -kernel ...**).

Part B: Adding a new system call to the Linux kernel

Introduction

Though adding system calls to the kernel should never be your first choice to expand the kernel's functionality in the real world (applications and developers expect a certain set of calls), for the purpose of this class it will be a rather common habit. Below we describe how to add a new system call to the Linux kernel v2.6.26.5 from the kernel repository. For information on the underlying mechanisms, refer here, as well as in the suggested class resources and of course, any internet search engine.

We will be adding a system call named cmpt300_test which takes an integer as an argument and prints it out along with a greeting.

Adding a new system call - the Linux-kernel side

- **cd linux-2.6.26.5**
  - Change your current working directory to the kernel directory. From now on we assume you are in the kernel directory
- **mkdir cmpt300**
  - Make a new directory to host your new system call. We will call this directory cmpt300/
- **nano cmpt300/cmpt300_test.c**
  - Open a new file and add your system call there. As described, ours will be called **cmpt300_test**, it will take a single argument (int) and print it along with a simple greeting:

```
#include <linux/kernel.h>

asmlinkage long sys_cmpt300_test(int arg0)
{
  printk(" Hello World !");
  printk("--syscall arg %d", arg0);

  return((long) arg0);
}
```

Save your file under cmpt300/

- Function **printk** is the kernel's analogue of printf

- **nano cmpt300/Makefile**
  - Open a new Makefile to inform the kernel how to compile your new file(s). You only need to define how the single file you gave is turned into an object file, so in this simple case

    #Makefile start

    obj-y := cmpt300_test.o

    #Makefile end

    Save your file under cmpt300/

- **nano Makefile**
  - Open the topmost Makefile and inform it of the new directory. The Makefile will descend into it and recursively build whatever it can. To do so, find the line where **core-y** is defined (~610) and add your new directory to it:

    core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ ***cmpt300/***

    Save the Makefile where it was.

- **nano arch/x86/kernel/syscall_table_32.S**
  - Edit (with vi or any editor of your preference) syscall_table_32.S to add a new system call function pointer. Go to the last line and add the line

    .long sys_cmpt300_test / 327 */*

    By this line you add the pointer to your function in the system-calls table; this is system call number 327 for i386 processors.

- **nano include/asm-x86/unistd_32.h**
  - Edit unistd_32.h to add your new system call number. Find where the rest of the system call numbers are, go to the last one (326) and add the line

    #define __NR_cmpt300_test 327

    Now you defined how your system call function pointer can be referenced from the system calls table.

***Now recompile your kernel. See QDGL : Compiling and testing a new kernel or see the**

script build-kernel.sh

Adding a new system call - the user-level side

Since all the libraries we have are unaware of our new system call, in order to test it we need to provide the user with a library call to trigger the syscall. Boot your GNU/Linux distribution (Debian in QDGL's case), login as a simple user and follow the procedure below.

- **mkdir cmpt300**
  - Make a new directory to host your new library call. We will call this directory cmpt300/
- **cd cmpt300**
  - **Move to the new directory**
- **nano cmpt300_test.c**
  - Open a new file and add your library call declaration there.

```
#include <stdio.h>
#include <linux/unistd.h>
#include <sys/syscall.h>

#define _cmpt300_TEST_ 327

int main(int argc, char *argv[])
{
  printf("\nDiving to kernel level\n\n");
  syscall(_cmpt300_TEST_ , 300);
  printf("\nRising to user level\n\n");

   return 0;
}
```

Save your file.

- **gcc cmpt300_test.c - o cmpt300_test**
  - Compile your user level syscall interface

Testing your new system call

- Boot your new kernel
  - **qemu-system-i386 -m 64M -hda hda.img -nographic -append "root=/dev/hda1" -kernel linux-2.6.26.5/arch/i386/boot/bzImage**
  - You should be able to verify that this is a new kernel by checking its revision (ex. uname -a)
- Go the directory where you stored your library interface to the syscall and execute the test you wrote
- Your system call should now work if you try it :
  - The example above printed the following output to our virtual machine:

    Diving to kernel level

    [76.703723] Hello World !--syscall arg 300
    Rising to user level
- If you try the same system call in the kernel that does not support it (like the default Debian kernel), you would just see the user level printouts:
  - *Diving to kernel level*

    Rising to user level