CS 471 Operating Systems
Fall 2020

Programming Assignment 0 (PA0): Setting up OS/161 environment and code reading

Due date: Friday, September 11, 11:59 pm

Table of contents

1. Introduction

This assignment will familiarize you with OS/161, the operating system with which you will be working throughout this semester, and System/161, the machine simulator on which OS/161 runs. We also introduce tools that will make your work this semester easier.

While this assignment does not require to write non-trivial code, it does require that you do some careful code reading and exploring of the directory organization of a kernel source code and binary package. So, you are encouraged to spend more time and performing simple trial-and-error process on your own to study the OS/161 system. As you will see, this is in fact crucial in answering the questions below.

The first part of this document briefly discusses the code on which you'll be working and the tools you'll be using. The following sections provide precise instructions on exactly what you must do for the

assignment. **Make sure** you read the entire assignment before you start working on it. Each section with **[hand me in]** at the beginning indicates a section where there is something that you must do for the assignment. Note that you are to hand in 6 files, summarized at the end. You will be handing in answers to questions as well as scripts (screen captures) of multiple activities, in a compressed file.

All the steps shown below must be completed on the Zeus Linux cluster of VSE (zeus.vse.gmu.edu).

2. What are OS/161 and System/161?

The code for this semester is divided into two main parts:

- **OS/161:** the operating system that you will augment in subsequent programming assignments.
- **System/161:** the machine simulator which emulates the physical hardware on which your operating system will run. This course is about writing operating systems, not designing or simulating hardware. Therefore, you may not change the machine simulator. If, however, you think you have found a bug in System/161, please let the course staff know as soon as possible.

The OS/161 distribution contains a full operating system source tree, including some utility programs and libraries. After you build the operating system you boot, run, and test it on the simulator.

We use a simulator in CS 471 because debugging and testing an operating system on real hardware is extremely difficult. The System/161 machine simulator has been found to be an excellent platform for rapid development of operating system code, while still retaining a high degree of realism. Apart from floating point support and certain issues relating to RAM cache management, it provides an accurate emulation of a MIPS R3000 processor.

OS161 assignments are cumulative. You will build each assignment on top of your previous submission.

3. About Git and GitLab

Git is a free and open source distributed version control system. It manages the source files of a software package so that multiple programmers may work simultaneously. Each programmer has a private copy of the source tree and makes modifications independently. GitLab offers a centralized hosting for the Git repositories. Git synchronizes with the central repository using the ‘push’ and ‘pull’ commands. Git also attempts to intelligently merge multiple people's modifications, highlighting potential conflicts when it fails. You can find several online tutorials to explain the basic features and commands of Git.

Most programming you have probably done at GMU has been in the form of 'one-off' assignments: you get an assignment, you complete it yourself, you turn it in, you get a grade, and then you never look at it again. The commercial software world uses a very different paradigm: development continues on the same code base producing releases at regular intervals. This kind of development normally requires multiple people working simultaneously within the same code base, and necessitates a system for tracking and merging changes. Therefore, it is imperative that you start becoming comfortable with Git, which is arguably the most popular distributed version control system of modern time.

Git is very powerful, but for CS471 you only need to know a subset of its functionality.

4. About GDB

GDB (GNU Debugger) allows you to examine what is happening inside a program while it is running. It lets you execute programs in a controlled manner and view and set the values of variables. In the case of OS/161, it allows you to debug the operating system you are building instead of the machine simulator on which that operating system is running.

In some ways debugging a kernel is no different from debugging an ordinary program. On real hardware, however, a kernel crash will crash the whole machine, necessitating a time-consuming reboot. The use of a machine simulator like System/161 provides several debugging benefits. First, a kernel crash will only crash the simulator, which only takes a few keystrokes to restart. Second, the simulator can sometimes provide useful information about what the kernel did to cause the crash, information that may or may not be easily available when running directly on top of real hardware.

You must use the CS471 version of GDB to debug OS/161. You can run on the UNIX systems used for the course as cs161-gdb. This copy of GDB has been configured for MIPS and has been patched to be able to communicate with your kernel through System/161. An important difference between debugging a regular program and debugging an OS/161 kernel is that you need to make sure that you are debugging the operating system, not the machine simulator. Type

```
% cs161-gdb sys161
```

and you are debugging the simulator.

5. Setting up your account and getting the distribution

As the first step, you will need to download the os161 distribution you will be working on. Be sure to type the commands manually on to the command prompt, instead of doing copy/paste from the spec.

To login to Zeus, ssh to zeus.vse.gmu.edu using your GMU username and password

```
% ssh <your-username>@zeus.vse.gmu.edu
```

In your home directory, create a subdirectory named tmp:

```
% mkdir ~/tmp ; cd ~/tmp
```

Then get the os161 distribution you will be working on by entering the following on the command line

```
% wget https://cs.gmu.edu/~yuecheng/os161-1.11.tar.gz
```

If you get a certificate error, you need to use the `--no-check-certificate` option with wget.

Next, untar the file you just downloaded:

```
% tar -xzf os161-1.11.tar.gz
```

Finally issue the module load command to load the os161 toolchain:

```
% module load sys161/1.14
```

Optionally, you can add the above "module load" command to the end of your `~/.bash_profile` file. That way, you won't have to type it every time you log-in to Zeus.

6. Setting up your GitLab repository

We will be using Mason GitLab for all our OS/161 kernel hacking projects. You will use Git to keep track of your code editing history.

Step 1: Create a GitLab Repo
First, you will need to login to GitLab (URL: https://git.gmu.edu/users/sign_in) by clicking `Sign in with: GMU Login`. Username and Password never work for some reason.

When you sign on, normally you will receive a confirmation e-mail within around 10 minutes. If you don't receive the confirmation e-mail within a few hours, send a message to the Piazza forum so that the GTAs can investigate.

Important: Make sure you create a **Private** project with the name `os161-1.11`. When creating a new project, you can directly set the visibility to Private upfront. This is important because sharing your os161 source code in a public repository would be considered **honor code violation.**

Step 2: Clone Your GitLab Repo
Before cloning your GitLab repo to Zeus, you will need to first create an RSA SSH key (on Zeus) by typing:

```
% ssh-keygen -t rsa -C "your_email_addr"
```

Or you can simply follow this tutorial at the URL: https://git.gmu.edu/help/ssh/README#generating-a-new-ssh-key-pair .

Then, add an SSH key to your GitLab account by following these instructions at the URL: https://git.gmu.edu/help/ssh/README#adding-an-ssh-key-to-your-gitlab-account

Test if the SSH-based access has been successfully set following the instructions in the following URL: https://git.gmu.edu/help/ssh/README#adding-an-ssh-key-to-your-gitlab-account

Click on the **Clone** button at the right-top corner of your GitLab repo's webpage, copy the string under **Clone with SSH** to clipboard. Then, create a new directory called os161 under your $HOME directory, cd to the directory (you are supposed to put your os161 source code in this directory for all the projects), and clone your created GitLab repo on to your (Zeus) Linux box:

```
% mkdir ~/os161 ; cd ~/os161
% git clone git@git.gmu.edu:your_gid/os161-1.11.git
```

You can then copy the downloaded os161 src into this newly created git directory:

```
% cd ~/os161/os161-1.11
% cp -r ~/tmp/os161-1.11/* .
```

You can delete the "~/tmp" directory after this step.

Step 3: Check-in Your Initial Source Code

Now, check in the source code which you have already copied into the git directory:

```
% git add *
% git commit -m "init commit"
% git push
```

If you are checking in for the first time on an empty repo (which is your case here), you should run:

```
% git push -u origin master
```

Instead of

```
% git push
```

Step 4: Share Repo with Your Teammate

If you are working in a group, it is a good idea for all group members to share access to a single GitLab repo. You can enable sharing through the GitLab web interface. Hover over to **Settings** on the left sidebar, and click **Members**. Enter your team member's Patriot ID and choose a role (Developer or Maintainer) for him/her. Your partner needs to issue the "git clone" command with the "Clone with SSH" URL.

7. Code reading [hand me in]

One of the challenges of os161 is that you are going to be working with a large body of code that was written by someone else. When doing so, it is important that you grasp the overall organization of the entire code base, understand where different pieces of functionality are implemented, and learn how to augment it in a natural and correct fashion. As you and your partner develop code, although you needn't understand every detail of your partner's implementation, you still need to understand its overall structure, how it fits into the greater whole, and how it works.

In order to become familiar with a code base, there is no substitute for actually sitting down and reading the code. Admittedly, most code makes poor bedtime reading (except perhaps as a soporific), but it is essential that you read the code.

You should use the code reading questions included below to help guide you through reviewing the existing code. While you needn't review every line of code in the system in order to answer all the questions, we strongly recommend that you look over every file in the system.

The key part of this exercise is understanding the base system. Your goal is to understand how it all fits together so that you can make intelligent design decisions when you approach future assignments. This may seem tedious, but if you understand how the system fits together now, you will have much less

difficulty completing future assignments. Also, it may not be apparent yet, but you have much more time to do so now than you will at any other point in the semester.

The file system, I/O, and network sections may seem confusing since we have not discussed how these components work. However, it is still useful to review the code now and get a high-level idea of what is happening in each subsystem. If you do not understand the low-level details now, that is OK.

These questions are not meant to be tricky - most of the answers can be found in comments in the OS/161 source, though you may have to look a textbook for some background information. Place the answers to the following questions in a file called ~/os161/asst0/code-reading.txt.

Top Level Directory

The top level directory of many software packages is called `src` or `source`. The top of the OS/161 source tree is also called os161-1.11. In this directory, you will find the following files:

Makefile: top-level makefile; builds the OS/161 distribution, including all the provided utilities, but does not build the operating system kernel.
configure: this is an autoconf-like script. It sets up things like `How to run the compiler.' You needn't understand this file, although we'll ask you to specify certain pathnames and options when you build your own tree.
defs.mk: this file is generated when you run `./configure`. You needn't do anything to this file.
defs.mk.sample: this is a sample defs.mk file. Ideally, you won't be needing it either, but if configure fails, use the comments in this file to fix defs.mk.

You will also find the following directories:
bin: this is where the source code lives for all the utilities that are typically found in /bin, e.g., cat, cp, ls, etc. The things in bin are considered "fundamental" utilities that the system needs to run.
include: these are the include files that you would typically find in /usr/include (in our case, a subset of them). These are user level include files; not kernel include files.
kern: here is where the kernel source code lives.
lib: library code lives here. We have only two libraries: libc, the C standard library, and hostcompat, which is for recompiling OS/161 programs for the host UNIX system. There is also a crt0 directory, which contains the startup code for user programs.
man: the OS/161 manual ("man pages") appear here. The man pages document (or specify) every program, every function in the C library, and every system call. You will use the system call man pages for reference in the course of assignment 2. The man pages are HTML and can be read with any browser.
mk: this directory contains pieces of makefile that are used for building the system. You don't need to worry about these, although in the long run we do recommend that anyone working on large software systems learn to use make effectively.
sbin: this is the source code for the utilities typically found in /sbin on a typical UNIX installation. In our case, there are some utilities that let you halt the machine, power it off and reboot it, among other things.
testbin: these are pieces of test code.

You needn't understand the files in bin, sbin, and testbin now, but you certainly will later on. Eventually, you will want to modify these and/or write your own utilities and these are good models. Similarly, you need not read and understand everything in lib and include, but you should know enough about what's there to be able to get around the source tree easily. The rest of this code walk-through is going to concern itself with the kern subtree.

The kern subdirectory

Once again, there is a Makefile. This Makefile installs header files but does not build anything. In addition, we have more subdirectories for each component of the kernel as well as some utility directories.
kern/arch: This is where architecture-specific code goes. By architecture-specific, we mean the code that differs depending on the hardware platform on which you're running.
For our purposes, you need only concern yourself with the mips subdirectory.
kern/arch/mips/conf:
conf.arch: This tells the kernel config script where to find the machine-specific, low-level functions it needs (see kern/arch/mips/mips).
Makefile.mips: Kernel Makefile; this is copied when you "config a kernel".
kern/arch/mips/include: These files are include files for the machine-specific constants and functions.

- **Question 1.** Which register number is used for the frame pointer (fp) in OS/161? How do you know? Explain your answer.
- **Question 2.** What bus/buses does OS/161 support? How do you know? Explain your answer.
- **Question 3.** What is the difference between splhigh and spl0? Explain.
- **Question 4.** What are some of the details which would make a function “machine dependent”? Why might it be important to maintain this separation, instead of just putting all of the code in one function?

kern/arch/mips/mips: These are the source files containing the machine-dependent code that the kernel needs to run. Most of this code is quite low level.

- **Question 5.** What does splx return? How did you get your answer? Explain.
- **Question 6.** How many hardware interrupts lines does MIPS have? How many of them are we actually using in OS/161?

kern/asst0: You can safely ignore this directory for now.
kern/compile: This is where you build kernels.

8. Building kernel [hand me in]

In order to build kernel, you have two options. You can use the php file provided by the TA or use the alternative method. Before starting any of the methods, make sure to issue the module load command

```
% module load sys161/1.14
```

IMPORTANT: You should remember to issue the module load command above everytime you logon to zeus when you intend to work on os161 !

First method: This is based on running a script provided by the TA. Give the following command to get the script file

```
% wget http://mason.gmu.edu/~aroy6/build-asst0.php
```

Then run this php file by typing

```
% php build-asst0.php
```

Alternative method: In the compile directory, you will find one subdirectory for each kernel you want to build. In a real installation, these will often correspond to things like a debug build, a profiling build, etc. In our world, each build directory will correspond to a programming assignment, e.g., ASST2, ASST3, etc. These directories are created when you configure a kernel. This directory and build organization is typical of UNIX installations and is not universal across all operating systems.

kern/conf: config is the script that takes a config file, like ASST0, and creates the corresponding build directory. So, in order to build a kernel, you should enter the following:

```
% cd os161-1.11
% ./configure --ostree=$HOME/os161/root
% cd kern/conf
% ./config ASST0
% cd ../compile/ASST0
% make depend
% make
% make install
```

This will create the ASST0 build directory and then actually build a kernel in it. Note that you should specify the complete pathname ./config when you configure OS/161. If you omit the ./, you may end up running the configuration command for the system on which you are building OS/161, and that is almost guaranteed to produce rather strange results.

kern/dev: This is where all the low level device management code is stored. Unless you are really interested, you can safely ignore most of this directory.

kern/include: These are the include files that the kernel needs. The kern subdirectory contains include files that are visible not only to the operating system itself, but also to user-level programs. (Think about why it's named "kern" and where the files end up when installed.)

- **Question 7.** How many interrupt levels we *actually* use in OS/161?
- **Question 8.** How large are OS/161 pids (process identifiers)? How many processes do you think OS/161 could support as you have it now? A sentence or two for justification is fine.
- **Question 9.** What is the system call number for a reboot? Is this value available to userspace programs? Why or why not?

kern/lib: These are library routines used throughout the kernel, e.g., managing sleep queues, run queues, kernel malloc, etc.

- **Question 10.** What is the purpose of functions like copyin and copyout in copyinout.c? What do they protect against? Where might you want to use these functions?

kern/main: This is where the kernel is initialized and where the kernel main function is implemented.
kern/thread: Threads are the fundamental abstraction on which the kernel is built.

- **Question 11.** What is the difference between 'thread_exit()' and 'thread_destroy()'? Can a thread call thread_exit() on itself? Can it call thread_destroy() on itself? If not, why?
- **Question 12.** What are the possible states that a thread can be in? When do "zombie" threads finally get cleaned up? How did you obtain your answer. Explain.
- **Question 13.** What function makes a thread to yield the CPU? When might you want to use this function?

kern/userprog: This is where you will add code to create and manage user level processes. As it stands now, OS/161 runs only kernel threads; there is no support for user level code. Later, you'll implement this support.
kern/vm: This directory is also fairly vacant at present.
kern/fs: The file system implementation has two subdirectories. We'll talk about each in turn. kern/fs/vfs is the file-system independent layer (vfs stands for "Virtual File System"). It establishes a framework into which you can add new file systems easily. You will want to go look at vfs.h and vnode.h before looking at this directory.
kern/fs/sfs: This is the simple file system that OS/161 contains by default.

9. Running the kernel [hand me in]

1. Script the following session using the command:
   `% script <return>`
2. Change into your root directory and get the sys161.conf file
   `% cd ~/os161/root`
   `% cp /opt/apps/sys161-1.14/sys161.conf.sample sys161.conf`
3. Run the machine simulator on your operating system.
   `% sys161 kernel`
4. At the prompt, type
   `p /sbin/poweroff <return>`
   This tells the kernel to run the "poweroff" program that shuts the system down.
   NOTE: Unless you run the 'building a kernel' script we provide, this step won't work. If you are building your own (without the script) you need to run 'make' in the top of the source tree (should be ~/os161/os161-1.11/Makefile)
5. End your script session (using the "`exit`" command.)
6. Rename your script output to run.script
   `% mv typescript ~/os161/asst0/run.script`

10. Practice modifying your kernel [hand me in]

1. In the kern/main directory, create a file called hello.c.
2. In this file, write a function called hello that uses kprintf() to print "Hello World\n".
3. Edit main/main.c and add a call (in a suitable place) to hello().
4. Make your kernel build again. You will need to edit conf/conf.kern, reconfig, and rebuild.
5. Make sure that your new kernel runs and displays the new message. Add hello.c to Git by typing
   ```
   % git add hello.c
   ```

6. Once your kernel builds, script a session demonstrating a config and build of your modified kernel. Your script should also show a run of your modified kernel. Call the output of this script session newbuild.script
   ```
   % mv typescript ~/os161/asst0/newbuild.script
   ```

11. Using GDB [hand me in]

For this part, you need to connect to zeus in two different windows. These windows are called "run window" and "debug window", respectively. When connecting to zeus, in this phase, make sure to connect to the same physical machine (i.e., you should connect to zeus-1.vse.gmu.edu OR zeus-2.vse.gmu.edu) in BOTH windows (connecting to zeus-1 in one window, and zeus-2 in the other one will not work. Note that if you simply connect to zeus.vse.gmu.edu, you will connect to zeus-1 or zeus-2 in a more or less arbitrary manner).

1. Script the following gdb session (that is, you needn't script the session in the run window, only the session in the debug window).
2. Run the kernel in gdb by first running the kernel and then attaching to it from gdb.
   ```
   ------------ (In the run window:) ------------
   % cd ~/os161/root
   % sys161 -w kernel
   ------------ (In the debug window:) ------------
   % script
   % cd ~/os161/root
   % cs161-gdb kernel
   (gdb) target remote unix:.sockets/gdb
   (gdb) break menu
   (gdb) c
   [gdb will stop at menu() ...]
   (gdb) where
   [displays a nice back trace...]
   (gdb) detach
   (gdb) quit
   ```
3. End your script session. Rename your script output to gdb.script
   ```
   % mv typescript ~/os161/asst0/gdb.script
   ```

12. Practice with Git [hand me in]

In order to build your kernel above, you already checked out a source tree. Now we'll demonstrate some of the most common features of Git. Create a script of the following session (the script should contain everything except the editing sessions; do those in a different window). Call this file git-use.script.

1. Edit the file kern/main/main.c. Add a comment with your name in it.
2. Execute
   ```
   % git diff kern/main/main.c
   ```
   to display the differences in your version of this file.
3. Now commit your changes using
   ```
   % git commit -am "First comment"
   ```
4. Remove the first 100 lines of main.c.
5. Try to build your kernel (this ought to fail).
6. Realize the error of your ways and get back a good copy of the file.
   ```
   % rm main.c
   % git checkout main.c
   ```
7. Try to build your tree again.
8. Now, examine the DEBUG macro in lib.h. Based on your earlier reading of the operating system, add ten useful debugging messages to your operating system.
9. Now, show us where you inserted these DEBUG statements by doing a diff.
   ```
   % cd ~/os161/os161-1.11
   % git diff > ../asst0/asst0.diff
   % git commit -am "Initial 10 DEBUG statements"
   ```
10. Finally, you should create a release
    ```
    % cd ~/os161/os161-1.11
    % git tag asst0-end
    % git push
    % git archive --prefix=os161-1.11/ asst0-end | gzip -c >
../asst0/os161-1.11.tar.gz
    % cd ..
    % tar cf - asst0 | gzip -c >  mygroup-asst0.tar.gz
    ```

Above, "mygroup" should be replaced by the GMU ids of the students who worked together in the assignment (e.g., msmith-jwatson-asst0.tar.gz)

13. Working with a partner

Starting with this project (and in all subsequent OS161 projects), you can - and you are **ENCOURAGED** to – work with a partner: another CS 471 student of your own section.

- A team can have no more than two members
- It is fine if you choose to work alone, but there will be no bonus points for it
- Projects are developed assuming that two students will actively interact and cooperate in exploring OS/161 and augmenting it with several features in the subsequent assignments
- When choosing a partner, pay careful attention to each other's time constraints

- You are *not* allowed to change your partner in the subsequent assignments, although you can choose to work alone in the subsequent assignments.

14. Submission and grading

You will submit your assignment via your Blackboard. The submission will consist of a properly tar'd and compressed (gzipped) file of your asst0 directory. The tar.gz file must be named to reflect the GMU ids of the students who worked together in the assignment. For example, mygroup-asst0.tar.gz where "mygroup" should be replaced by the GMU ids of the students who worked together in the assignment (e.g., msmith-jwatson-asst0.tar.gz).

Before generating the tar.gz file, your os161/asst0 directory should contain everything you need to submit, specifically:

- code-reading.txt
- run.script
- newbuild.script
- gdb.script
- git-use.script
- os161-1.11.tar.gz

Make sure to double check that you are submitting the correct file -- if you submit wrong, corrupt, or empty file you are likely to get zero.
All members of a group must submit separately the same compressed file.
This programming assignment accounts for **6%** of your final grade. Late penalty for this assignment will be 15% for each day. Submissions that are late by 3 days or more will not be accepted. Please plan in advance. To avoid the late penalty, your submission must be timestamped by the Blackboard system indicating that it was submitted before 11:59 PM on 9/11.

Questions - Programming – and system-related questions about the project should be directed Fall 2020 CS 471 OS/161 Project Piazza Page (https://piazza.com/gmu/fall2020/cs471), which is monitored by the GTAs of all CS 471 sections from 10 AM to 6 PM on weekdays. Your posts to the Piazza are by default *private*, meaning that it is received only by the instructors and GTAs. You should NOT post public inquiries by changing those settings (occasionally, the GTAs can decide to make some questions and answers *public,* if they think they are relevant for many students).

Please note that you are expected to do the code reading and exploring yourself - so you should not expect the GTAs to provide a key information that will immediately reveal the answer to some of the (short) questions.

You should consider that re-building kernel (even after minor changes) is a CPU- and I/O-intensive task for the zeus server. It is not uncommon for zeus to take several minutes to re-build your modified kernel, especially when there are many users logged on (as common on the assignment due days!) Since you may need to make some changes, iteratively re-building kernel may require significant processor time. Consider this and try to avoid postponing the project to the very last day.