CSE3201 Operating Systems
Second Semester, Third Year 2018
Assignment 0 Part 2

Version 1.2 post date: July 26, 2018
last modified 26/07/2018 at about 13:20:00

This assignment is worth a possible 15% marks of the lab class mark component of your assessment.
No bonuses apply for this assignment.
The assignment is due on Thursday, 09 of August 2018, at 23:59:59.

Introduction

The aim of Assignment 0, Part 2 is to have you familiarise yourself with the environment you will be using for the remaining assignments. The assignment consists of two assessable parts.

The first assessable component consists of a set of directed questions to guide you through the code. The answers to this code reading Q & A component of the assignment will be given. You will produce a report in latex format composed of question and answer accordingly. The only accepted report is a latex script which will produce a .dvi file after compilation.

The second part of this assignment consists of you making a very minor change to the existing OS. The change is conceptually trivial, so you can view this assignment as us giving away marks as an incentive for you to get the assignment environment up and running early in the session. This assignment is worth 15% of the LAB mark component of your final assessment.

Note that code reading component is assessable, we view it as compulsory. You will really struggle with the assignments if you fail to get an understanding of the code base. The code reading component is there to guide you towards acquiring that understanding.

Also note that this assignment is not indicative of the level of difficulty of the later assignments. The later assignments will be much more challenging.

This assignment will introduce you to the following components of the environment you will use during the semester.

1. * OS/161, the educational operating system you will modify to implement the assignments.
2. * System/161, the machine simulator that OS/161 runs on.
3. * GDB, a debugger that will make your life much easier.

OS/161

OS/161 is an educational operating system. It aims to strike a balance between giving students experience working on a real operating system, and potentially overwhelming students with the complexity that exists in a fully fledged operating system, such as Linux. Compared to most deployed operating systems, OS/161 is quite small (approximately 20,000 lines of code and comments), and therefore it is much easier to develop an understanding of the entire code base, as you will begin to do during this assignment.

The source code distribution contains a full operating system source tree, including the kernel, libraries, various utilities (ls, cat, etc.), and some test programs. The OS/161 boots on the simulated machine in the same manner as a real system might boot on real hardware.

System/161

System/161 simulates a "real" machine to run OS/161 on. The machine features a MIPS R2000/R3000 CPU including an MMU, but no floating point unit or cache. It also features simplified hardware devices hooked up to lamebus. These devices are much simpler than real hardware, and thus make it feasible for you to get your hands dirty, without having to deal with the typical level of complexity of physical hardware.

Using a simulator has several advantages. Unlike software you have written thus far (Windows excluded :-)), buggy software may result in completely locking up the machine, making to difficult to debug and requiring a reboot. A simulator allows debuggers access to the machine below the software architecture level as if debugging was built into the CPU chip. In some senses, the simulator is similar to an in circuit emulator (ICE) that you might find in industry, only it's done in software. The other major advantage is speed of reboot, rebooting real hardware takes minutes, and hence the development cycle can be frustratingly slow on real hardware.

GDB

You should already be familiar the GDB, the GNU debugger. GDB allows you to set breakpoints to stop your program under certain conditions, inspect the state of your program when it stops, modify its state, and continue where it left off. It is a powerful aid to the debugging process that is worth investing the time needed to learn it. GDB allows you to quickly find bugs that are very difficult to find with the typical printf style debugging.

Details beyond the level you need to know can be found at http://www.gnu.org/software/gdb/gdb.html. A brief and focused introduction will be given later in this document.

The group Reading/Discussion Part 2 of Assignment 0

This is probably the first time most of you will attempt to understand, and carefully modify, a large body of code that you did not write yourself. It is imperative that you take the time to read through the code to get an understanding of the overall structure of the code, and what each part of the code does.

This assessable, code reading component of this assignment aims to guide you through the code base to help you comprehend its contents, identify what functionality is implemented where, and be able to make intelligent decisions on how to modify the code base to achieve the goals of the assignments.

You don't need to understand every line of code, but you should have a rough idea of what some files do.

Invest the time now in gaining an overall understanding of the code base. Now is probably the least busiest part of the year for you. Don't waste it and struggle later.

The top-level Directory

The os161-ASST0 directory contains the top-level directory of the OS/161. It contains a few files, and subdirectories containing distinct parts of OS/161. The files are:

1. * Makefile this makefile builds the OS/161 distribution, including all the provided utilities. It does not build the operating system kernel.
2. * configure: this is a configuration script, similar to autoconf, but not generated by autoconf. You shouldn't need to understand or tamper with it.

3. * defs.mk: this file is generated by running ./configure. Unless something goes wrong, you shouldn't need to do anything with it.
4. * defs.mk.sample: this is a sample defs.mk file in case something does go wrong with configure. If configure does fail, you can fix def.mk using the comments in this file.

os161 contains the following directories:

1. * bin: contains the source code for the user-level utilities available on OS/161. They are a subset of the typical unix /bin tools, e.g. cat, cp, ls.
2. * include: these are the include files used to build user-level programs on OS/161, they are not the kernel include files. Among other things, they contain appropriate definitions for using the C library available on OS/161.
3. * kern: contains the sources to the OS/161 kernel itself. We will cover this in more details later.
4. * lib: the user-level library code for libc is here.
5. * sbin: contains the source code for the user-level system management utilities found in /sbin on a UNIX machine (e.g. halt, reboot, etc.)
6. * testbin: these are pieces of test code. They are most relevant to the course given at Harvard, but are included here for your perusal and potential use.

Your focus during this code walk through should be on the kernel sources. You won't need a detailed understanding of the utilities in bin and sbin, however broad understanding of how they work and where things are is useful. Likewise with the lib and include directories.

The Kern Subdirectory

This directory and its subdirectories are where most (if not all) of the action takes place. The only file in this directory is a Makefile. This Makefile only installs various header files. It does not actually build anything.

We will now examine the various subdirectories in detail. Take time to explore the code and answer the questions, include questions and answer in your report.

kern/arch

This directory contains architecture-dependent code, which means code that is dependent on the architecture OS/161 runs on. Different machine architectures have their own specific architecture-dependent directory. Currently, there is only one supported architecture, which is mips.

kern/arch/mips/conf

conf.arch: This tells the kernel config script where to find the machine-specific, low-level functions it needs (see mips/mips).

> Question 1: What is the vm system called that is configured for assignment 0?

Makefile.mips: Kernel Makefile; it copies this when you "config a kernel".

kern/arch/mips/include

These files are include files for the machine-specific constants and functions.

> Question 2. Which register number is used for the stack pointer (sp) in OS/161?
> Question 3. What bus/busses does OS/161 support?
> Question 4. What is the difference between splhigh and spl0?
> Question 5. Why do we use typedefs like u_int32_t instead of simply saying "int"?
> Question 6: What must be the first thing in the process control block?

kern/arch/mips/mips

These are the low-level functions the kernel needs that are machine-dependent.

Question 7. What does splx return?
Question 8. What is the highest interrupt level?
Question 9. What function is called when user-level code generates a fatal fault?

kern/asst1

This is the directory that contains framework code for one of the assignments at Harvard. You can safely ignore it.

kern/compile

This is where you build kernels. 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., ASST1, ASST2, etc. These directories are created when you configure a kernel (described in the next section). This directory and build organisation is typical of UNIX installations and is not necessarily universal across all operating systems.

kern/conf

config is a shell script that takes a config file, like ASST1, and creates the corresponding build directory. Later (not now), in order to build a kernel, you will do the following:

```
% cd kern/conf
% ./config ASST0
% cd ../compile/ASST0
% bmake depend
% bmake
```

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/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. Now answer the following questions.

Question 10. How frequently are hardclock interrupts generated?
Question 11. What functions comprise the standard interface to a VFS device?
Question 12. How many characters are allowed in a volume name?
Question 13. How many direct blocks does an SFS file have?
Question 14. What is the standard interface to a file system (i.e., what functions must you implement to implement a new file system)?
Question 15. What function puts a thread to sleep?
Question 16. How large are OS/161 pids?
Question 17. What operations can you do on a vnode?
Question 18. What is the maximum path length in OS/161?
Question 19. What is the system call number for a reboot?
Question 20. Where is STDIN_FILENO defined?

kern/main

This is where the kernel is initialised and where the kernel main function is implemented.

Question 21. What does kmain() do?

kern/thread

Threads are the fundamental abstraction on which the kernel is built.

Question 22. Is it OK to initialise the thread system before the scheduler? Why (not)?
Question 23. What is a zombie?
Question 24. How large is the initial run queue?

kern/lib

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

kern/userprog

This is where to 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.

kern/vm

This directory is also fairly vacant. Virtual memory would be mostly implemented in here.

kern/fs

The file system implementation has two subdirectories. We'll talk about each in turn.

kern/fs/vfs

This 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 review vfs.h and vnode.h before looking at this directory.

Question 25. What does a device name in OS/161 look like?
Question 26. What does a raw device name in OS/161 look like?
Question 27. What lock protects the vnode reference count?
Question 28. What device types are currently supported?

kern/fs/sfs

This is the simple file system that OS/161 contains by default. You may augment this file system as part of a future assignment, so we'll ask you questions about it then.

kern/dev

This is where all the low level device management code is stored. You can safely ignore most of this directory.

This concludes the assessable group reading/discussion report writing component of the assignment 0. Feel free to discuss your answers with fellow group member, and your teacher. You can discuss with any student to understand but your report should be different from other groups. You have to prepare a report groupwise as assigned previously. You are not allowed to copy all or any part of othe report, if you thus so, you will be given 0. Submit the final report as Latex source code. See near the end of this document which will discuss about format, name and other specification of your report which must be followed.

Building a Kernel

First download os161-ASST0.zip from course website and unzip. Now to the business end of this assignment. You will now build and install a kernel.

* You first have to configure your source tree.

% cd os161-ASST0
% ./configure
* Now you must configure the kernel itself.
% cd os161-ASST0/kern/conf
% ./config ASST0
* The next task is to build the kernel.
% cd ../compile/ASST0
% bmake depend
% bmake
* Now install the kernel
% bmake install

Running your Kernel

If you have made it this far, your have built and installed the entire OS. Now it is time to run it.

* cp the sample sys161.conf.sample to sys161.conf in ~/cs161/root/ directory
* Change to the root directory of your OS.

% cd ~/os161/root * Now run system/161 (the machine simulator) on your kernel.
% sys161 kernel
* Power off the machine by typing q at the menu prompt.

Using GDB

I cannot stress strongly enough to you the need to learn to use GDB. You can find directions and a short tutorial on using GDB with os161 here. Note: the version of gdb used for these assignments is cs161-gdb.

Modifying your Kernel

We will now go through the steps required to modify and rebuild your kernel. We will add a new file to the sources. The file contains a function we will call from existing code. We need to add the file to the kernel configuration, re-config the kernel, and the rebuild again.

* Begin by downloading **hello.c** from course website and place it in kern/main/.
* Find an appropriate place the in the kernel code, and add a call to complex_hello() (defined in hello.c) to print out a greeting (Hint: one of the files in kern/main is very appropriate). It should appear immediately before the prompt.
* Since we added new file to the kernel code, we need to add it to the kernel configuration in order to build it. Edit kern/conf/conf.kern appropriately to include hello.c.
* When we change the kernel config, we need to re-configure the kernel again.

% cd ~/os161-ASST0/kern/conf
% ./config ASST0

* Now we can rebuild the kernel.

% cd ../compile/ASST0
% bmake depend
% bmake
% bmake install

* bmake treats warning as error
* If you find error (warning) using 'bmake', then try with bmake WERROR=
* Run your kernel as before. Note that the kernel will panic with an error message.

* Use GDB to find the bug (Hint: the display, break, and step commands will be very useful).
* Edit the file containing the bug, recompile as before and re-run to see the welcome message.

Note: If you simply modify a file in an already configed source tree, you can simply run make again to rebuild, followed by make install. You only need to reconfig if you add or remove a file from the config, and you only need to make depend if you add (or modify) a #include directive.

The Assessable part 2 of Assignment 0

This assignment is worth a possible 18 marks out of 25 for this assignment 0 in which total 25% of the lab mark component of your assessment

The task

Follow the above instructions to add the given file to the operating system. Once you have found (using GDB) and fixed the bugs, you have completed the assignment. Make sure you see the Hello World!!! output just prior to the menu prompt.

```
sys161: System/161 release 2.0.8, compiled Jul 9 2018 11:30:58

OS/161 base system version 2.0.3

Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014

President and Fellows of Harvard College. All rights reserved.

Put-your-group-name-here's system version 0 (ASST0 #2)

356k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
Hello World!!!
OS/161 kernel [? for menu]:
```

Check your assignment works with the menu choice supplied on the command line as follows.

```
% sys161 kernel q
```

This is generally the way we test your submission.

Once your assignment works. Now you ready to submit your assignment. Only one member of the group needs to submit. I suggest you do the following together for this assignment.

What to submit?

Assesable Part	Type	Format	File Name(s)	Mark Distribution
Assignment 0 part 2	Report all questions 1-28 with answers Add suitable header (title,groupname,member name and roll no etc)	Latex text	asst0_report.tex	7
Assignment 0 part 2	Code	C source file	hello.c,hello.h,main.c,menu.c,conf.kern location:os161-ASST0/kern/main, os161-ASST0/kern/conf	8

Using GDB

First create a symbolic link using the following command

```
% cd ~/os161/tools/bin
% ln -s mips-harvard-os161-gdb cs161-gdb
%
```

Luckily we have a debugger at our disposal. We can use GDB to help with the tracking down of bugs in the code. Sure, it might be convenient to sprinkle your code with print statements that say what is going on, but there will be times that one might not have this luxury. Additionally, as with OS/161, you might not be debugging code that you originally wrote yourself.

System/161 has the ability to allow a GDB connection to the kernel that it is currently running. This is done with the '-w' argument. Essentially, it tells System/161 to wait for a debugger connection.

Now, the version of GDB we are using has been patched to allow connections through UNIX sockets. You can use any debugger you want, but the cs161-gdb version that is supplied will definitely connect to System161. The reason that we can't directly connect to the kernel is because it is running in the System/161 emulator. You will want to open another console, switch to the location of the kernel-ASST0 and sys/161 (~/cs161/root if you used the default directory structure.) Now type the following command to get the debugger going:

```
cs161-gdb kernel-ASST0
```

Note: You have to have PATH set properly for this to work.

This tells gdb to load the symbols found in the binary that you specified. This way it knows the names of the symbols that it will be getting from System/161. Now, type the following command to connect to System/161 and start debugging the kernel:

```
target remote unix:.sockets/gdb
```

You should see something like the following after doing this:

Notes:
You need to restart the kernel and reconnect to System/161 to run another debugging session. To get OS/161 rebooted and reconnected again:

cd ~/root and
./sys161 kernel-ASST0

This means that the debugger is now talking to System/161. Some commands that you will need to know to find the current bug in the kernel. Also, not that many of the commands can be abbreviated to save typing. The abbreviations are also included.

---

next (n):	Step to the next line of code. This will "Step Over" any subroutine calls.
step (s) :	Step to the next line of code. This will "Step Into" any subroutine calls.
list (l):	List lines of code. Type "help list" for more information.
break (b):	Set a breakpoint. You can provide line numbers, function names, etc. For example, to break at our new function, type: "b complex_hello". Or if you wanted to do it by line number you would do: "b hello.c:33".
info (i):	Get information. For example, do "info b" for info on breakpoints.
continue (c):	Continue running the code until a signal or breakpoint causes it to stop.
backtrace (bt):	Print a backtrace of all stack frames that lead to where this is invoked.
quit (q):	Quit the debugger.

So, you will want to switch back and forth between OS/161 and gdb. For example, you might want to start testing the code by setting a breakpoint at complex_hello and see if you can figure out where the kernel is crashing. After setting the breakpoint, you would want to continue, at which time you will see that OS/161 is responding on its terminal again. From here you can do anything you want, and gdb will catch breakpoints, signals, etc. Basically, you will have to juggle between the control of OS/161, and the debugging of it between two terminals. This now leaves you with the first assignment. It is your task to find out why the kernel is crashing. Once you find the problem, you are to correct it, recompile the kernel, and run it. If you are successful you will see a statement similar to the following displayed by OS/161.

Plagiarism
We take cheating seriously!!!
Penalties include
-- Copying of code: 0
-- Help with coding: negative half the assignment's max marks
-- Originator of a plagiarised solution: 0 for the particular assignment
-- Team work outside group: 0 for the particular assignments
