[CPEN 331. Fall 2019](https://sites.google.com/view/cpen331-fall2019/home)

* [Home](https://sites.google.com/view/cpen331-fall2019/home)
* [Calendar](https://sites.google.com/view/cpen331-fall2019/calendar)
* [Assignments](https://sites.google.com/view/cpen331-fall2019/assignments)

**Assignment 1**

**Assignment 1**

The purpose of this assignment is to get you started with OS/161, the system we will be using throughout the semester. Like many assignments in this course, this one has been adapted from the original CS161 course at Harvard.

By the time you complete this assignment, you will have mastered the following:

* Install the environment for running OS/161
* Set up and use a git source control repository for your source code.
* Fetch and build the kernel source tree.
* Become familiar with the OS/161 source tree.
* Learn the basics of using the debugger.

**Step 1. OS/161 installation**

First, a few words about your working environment. You will be running OS/161 **not** on real hardware, but on a hardware simulator, called System/161. Running on a simulator is a lot more portable, because you can install the simulator anywhere you want. So the bulk of the work in the first step of the assignment is configuring the environment for running System/161. You will follow the link below to configure your own virtual machine that you will use later in the course to complete assignments. We advise performing the configuration on the machine that you are most likely to use for working on your assignments, e.g., your laptop. If you are likely to use more than one computer, perform the installation on each.

Follow [these instructions](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fos161ubc%2Fos161-installation&sa=D&sntz=1&usg=AFQjCNHVjtZQJAvxRSM9xiLHeDsSbebyUw) to install the OS/161 environment, download the code, build it and run System/161 with your newly built kernel.

You can complete these steps any time you'd like, however we highly recommend attempting them before your lab session, so that you can get help from TAs if you run into trouble.

**Step 2. Tag your repository**

Let's start by tagging your repository to indicate that you are beginning Assignment 1.

After you've got your git repository set up, and OS/161 code downloaded and built, open a shell, and issue the following commands:

**We will assume that your source base, tracked by git, is located in your home directory in ~/os161/src**.

git tag asst1-start

git push --tags

**Step 3. Create your submit file**

The next step is to create a file where you will put the answers to the code reading exercises below as well as some output that we will ask you to produce in your shell.

* Create a directory ~/os161/src/submit
* Then create ~/os161/src/submit/asst1

These directories must be under your working git tree, because you will use git to submit assignments. Then create a new file asst1-answers.txt, by firing it up in gedit, for example:

cd ~/os161/src/submit/asst1

gedit asst1-answers.txt

To avoid blocking your shell, you can open another tab in your terminal prior to running this command (by following File->Open Tab), or you can put an ampersand (&) after the editor command.

**Step 4. Copy some output from git commands into your submit file**

Now, to make sure that you properly completed the first step, we are asking you to run some commands in your shell and copy the output into the submit file.

1. First of all, let's make sure you can run System/161. In a shell, invoke sys161 (you should know how to do that if you have complete step 1) and copy the output of that command into your submit file.

2. Now, let's make sure that you got your git repository working. From any directory in your git-controlled source tree, type:

git log

**Copy the output of this command into your submit file.**

You should become familiar with git log, as this is a very useful command. It will show you the history of all commits in your local clone.

**Next execute the** git tag **command and copy/paste its output into your submit file**. If you correctly created a tag in Step 2, you should see it here.

**Step 5. Complete the code reading exercises**

The following is a set of code reading exercises. ***Write down the answers to all of the questions and put them into your submit file.***

OS/161 is a simplified skeleton of a modern operating system. It comes with a configurable build system, code for some useful user-level utilities that can be run from within OS/161, and of course code for the operating system itself. To complete the assignments of this course, you will need to get your hands deep in the guts of the OS/161 codebase, and the sooner you become familiar with it, the better. To that end, you should look through the files and begin to internalize how the code is structured, what goes where, and how things work. This applies both to the build system and the codebase itself.

To guide you in this process ***please write up and hand in answers to the questions found below in this section. Put them in a text file asst1-answers.txt in the submit/asst1 subdirectory of your OS/161 repository***.

The questions are designed to encourage code exploration. (We've tried to avoid questions that can be answered simply using grep.) The goal is to help you understand key parts of the system. That said, you are not yet expected to understand everything you see; that's what the rest of the course is for. But you should get the "gist," and your answers should reflect that.

Please be as detailed as possible, giving function names and full pathnames in the source tree where appropriate. You don't need to explain what every last line of a function does, but your answers should be conceptually complete. Note that some questions may require longer answers than others.

Before you begin, it may be useful to [familiarize yourself with some tools](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fos161ubc%2Fdebugging-os161%2Fbrowsing-the-source-code&sa=D&sntz=1&usg=AFQjCNF_3dv3AnHqXcy7dUM_icm9iotIYg) that will help you browse the source code more efficiently.

Let's begin with some discussion questions for the in-class part of this assignment.

**Question 1:** In the book chapters and in class you were introduced to the mechanisms used to transfer control between user processes and the operating system. Tell us where we can find the first line of OS/161 code that is executed when a trap occurs. Then tell us where control gets transferred to from that point. What about an interrupt? How does that differ?

**Question 2:** Making a system call, such as write, ultimately leads to a trap. Find the code in OS/161 that invokes system calls from user programs and causes traps. In which file and on which lines did you find this code?

**Question 3:** Why do you suppose there are libc functions in the "common" part of the source tree (common/libc) as well as in userland/lib/libc?

Below is a brief overview of the organization of the source tree, and a description of what goes where.

* configure -- top-level configuration script; configures the OS/161 distribution and build system. It does not configure the operating system kernel, which is a slightly different kind of configuration.

**Question 4:** Name two things that configure configures. What might invalidate that configuration and make you need/want to rerun it?

* Makefile -- top-level makefile; builds the OS/161 distribution, including all the provided utilities, but does not build the operating system kernel.
* common/ -- code used both by the kernel and user-level programs, mostly standard C library functions.
* kern/ -- the kernel source code.
* kern/Makefile -- Once again, there is a Makefile. This Makefile installs header files but does not build anything.
* 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. There are two directories here: mips which contains code specific to the MIPS processor and sys161 which contains code specific to the System/161 simulator.
* kern/arch/mips/conf/conf.arch -- This tells the kernel config script where to find the machine-specific, low-level functions it needs (throughout kern/arch/mips/\*).
* kern/arch/mips/include/ -- This folder and its subdirectories include files for the machine-specific constants and functions.

**Question 5:** 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/\* -- The other directories contain source files for the machine-dependent code that the kernel needs to run. A lot of this code is in assembler and will seem very low level, but understanding how it all fits together will help you immensely on Assignment 2.

**Question 6:** How large is a trapframe? Why?

* kern/arch/sys161/conf/conf.arch -- Similar to mips/conf/conf.arch.
* kern/arch/sys161/include -- These files are include files for the System161-specific hardware, constants, and functions. machine-specific constants and functions.
* kern/compile -- This is where you build kernels. See below.

**Question 7:** Under what circumstances should you re-run the kern/conf/config script?

**Question 8:** Under what circumstances should you run bmake depend in kern/compile/DUMBVM?

**Question 9:** Under what circumstances should you run bmake or bmake install in kern/compile/DUMBVM?

* 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/fs -- This is where the actual file system implementations go. The subdirectory sfs contains a simple default file system. You will augment this file system as part of Assignment 4, so we'll ask you more questions about it then. The subdirectory semfs contains a special-purpose file system that provides semaphores to user-level programs. We may ask you more questions about this later on, after we discuss in class what semaphores are.
* 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.)
* kern/lib -- These are library routines used throughout the kernel, e.g., arrays, kernel printf, etc. Note: You can use these data structures as you implement your assignments in CS161. We strongly encourage you to look around and see what we've provided for you.
* kern/main -- This is where the kernel is initialized and where the kernel main function is implemented.

**Question 10:** When you booted your kernel, you found that there were several commands that you could issue to experiment with it. Explain exactly where and what you would have to do to add a command that printed out, "Hello world!"

* kern/proc -- This is where process support lives. You will write most of the code that goes here during Assignments 4 and 5.
* kern/synchprobs -- This is the directory that contains/will contain the framework code that you will need to complete assignment 1. You can safely ignore it for now.
* kern/syscall -- 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. In Assignments 4 and 5, you'll implement this support.
* kern/thread -- Threads are the fundamental abstraction on which the kernel is built (do not forget to look back at header files!)
* kern/vfs -- The vfs is the "virtual file system." It is an abstraction for a file system and its existence in the kernel allows you to implement multiple file systems, while sharing as much code as possible. The VFS layer is the file-system independent layer. You will want to go look atvfs.h and vnode.h before looking at this directory.
* kern/vm -- This directory is fairly vacant. In Assignments 6 and 7, you'll implement virtual memory and most of your code will go in here.
* 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/ -- fragments of makefiles used to build the system.
* userland/ -- user-level libraries and program code
* userland/bin/ -- 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.

**Question 11:** Why do we need to include these in your OS/161 distribution? Why can't you just use the standard utilities that are present on the machine on which you're working?

* userland/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.
* userland/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.

**Question 12:** When a user program exits, what is done with the program's return value?

* userland/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.
* userland/testbin/ -- this is the source code for the test programs found in /testbin in the installed OS/161 tree. You will want to examine this directory closely and be aware of what's available here, as many of these test programs will be useful during the course of the semester.

Now that you've perused the source tree, here is the last question.

**Question 13:** Imagine that you wanted to add a new system call. List all the places that you would need to modify/add code. Then review your answers to questions 7-9 and note which of those actions you need to take in order to test the new system call.

**Step 6. Learn how to use the debugger**

First, make sure you can launch the debugger from the command line by following [these steps](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fos161ubc%2Fdebugging-os161&sa=D&sntz=1&usg=AFQjCNFPmNtC8UZ-Qj8wLw68AiUcA8-4Aw).

Next, walk through [this GDB tutorial](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fos161ubc%2Fdebugging-os161%2Fhow-to-use-gdb&sa=D&sntz=1&usg=AFQjCNH_JxUuJHAMsVrSNeMMWp5EKOnf7g) and try the commands explained there in your debug session. We highly recommend that you learn how to use the graphical interface to GDB as described at the end of that document.

**Step 7. Trace the execution from start to menu()**

Next, you will demonstrate that you know how to use GDB by answering a concrete question about OS161 execution and putting your answers in the file asst1-answers.txt in the submit/asst1 directory in the clone of your git repository. Rather than finding answers by reading through the source code, obtain them by running the debugger. ***When possible, copy the output of the debugger that helped you find the answer to each question into your submission file.***

**Question 14:** What is the name of the very first function that executes when OS161 starts up?

**Question 15:** What is the very first assembly instruction that executes?

**Question 16:** Set the breakpoints in the kernel function that shows the menu and in the kernel main function. Now tell GDB to display all the breakpoints that were set and copy the output to your submit file.

**Question 17:** Briefly describe what happens between the beginning of the execution and the invocation of the kernel main function.

**Question 18:** What is the assembly language instruction that calls the kernel main function?

**Question 19:** Step through the boot() code to find out what functions are called during early initialization. Paste the gdb output that shows you what these functions are.

**Question 20:** Set a breakpoint in thread\_bootstrap(). Once you hit that breakpoint, at the very first line of that function, attempt to print the contents of the \*bootcpu variable. Copy the output into the submit file.

**Question 21:** Now, step through that function until after the line that says 'bootcpu = cpu\_create(0)'. Now print the content of \*bootcpu and paste the output.

Now let's learn how to use some of the very useful gdb macros provided with OS161. Copy the contents of kern/gdbscripts/array into your .gdbinit file. If you don't know where .gdbinit is and how to make sure that gdb accepts its commands, make sure that you mind the answer in the [GDB tutorial](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fos161ubc%2Fdebugging-os161%2Fhow-to-use-gdb&sa=D&sntz=1&usg=AFQjCNH_JxUuJHAMsVrSNeMMWp5EKOnf7g). Read through the macros that you just copied and find the one that helps you print the contents of the array containing all the CPUs.

**Question 22:** Print the allcpus array before the boot() function is executed. Paste the output.

**Question 23:** Print again the same array after the boot() function is executed. Paste the output.

**Step 8. Submit your assignment**

Now save your asst1-answers.txt file. Next, you will submit it using git by following the instructions below. Remember that your submit file is a new file that you created recently, so you need to make git aware of it:

git add ~/os161/src/submit/asst0/asst1-answers.txt

Next, commit your changes to your local repository using git commit:

git commit -m "Answers to asst1"

The text next to -m option is the comment for your commit. If you had used just git commit without the -m option, git would have opened a text editor suggesting to enter a comment, and you could have entered it there and proceeded by saving the file.

Note that while you have committed the changes to your local repository, you have not yet published them anywhere, and thus you have not yet submitted your completed assignment! To do so, you need to push your clone back to your 'master' git repository.

cd ~/os161/src

git push

Finally, you should tag your repository to indicate that you have finished the assignment.

git tag asst1-submit

git push --tags

Congratulations! You are done with Assignment 1!

**Marking rubric**

\*\* Git repository: 7 points MAX.

3 points for producing the output of git commands and 4 points for correctly committing all the required files into the repository.

In other words:

7 if all files have been correctly committed to git and the output from the git commands is shown correctly.

Then:

Subtract 1-3 points if the output of git commands is incomplete

Subtract 1-4 points if the files were not correctly committed into the repository: e.g., some files are missing, imcomplete, tags wrong/missing etc.

\*\* System161 output: 7 points MAX.

7 points -- if the output from running sys161 is present.

0 if the output is not present.

\*\* Code reading and GDB questions: 46 points MAX: (2 points per question, 23 questions in total).

For each question:

0 if the answer is wrong or missing

1 for a weak, partially wrong or poorly written answer

2 for a correct, well-written answer

MAXIMUM POINTS FOR THE ASSIGNMENT: 60.