[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 4**

In this assignment you will implement a few system calls that allow user programs to use files. This assignment will further train you how to navigate and understand a large and complex source base, and will teach you how to implement system calls and transfer control between the user mode and the kernel. You will implement part of the process management subsystem, which will be the subject of the next assignment.

**Step 1. Prepare**

**Prepare your git repository for working in a group**

This is the first assignment that requires you to work in teams of two. This may be difficult if you are accustomed to working alone, but it is essential for the completion of the remaining assignments and is a worthwhile skill to develop in any case. These assignments are too complex to be done single-handedly, and you will gain valuable real-world experience from learning to work in a team effectively. Become familiar with [this document](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fcpen331%2Fteam-work-responsibilities-expectations-and-grading&sa=D&sntz=1&usg=AFQjCNHH0urUimciedReJ4e1VM5QbjsczA) to understand your expectations and responsibilities in working with a partner and to understand how your joint work will be graded.

To begin working together, you and your partner need to decide on a code base and then need to set up a shared Git repository. Choose your code base with care. The assignments are cumulative and you will have to live with the consequences of this decision for the remainder of the semester. We suggest that you and your partner resolve conflicts about things like programming style and naming conventions now in order to avoid confusion later. Working together on a program can be much more demanding and frustrating than doing lab work together. (Imagine writing a coherent term paper with someone else!)

The code base you select should be a working OS/161 system with clean, well-commented, bullet-proof synchronization primitives. You and your partner should share your solutions to the previous assignments (it's good practice to learn to read and understand someone else's code) and decide what your code base will be. You are free to choose either partner's code, to merge your solutions, or to use the solution set.

Before working on this assignment, you and your partner must set up a Git repository that will serve as the master repository for both of you for the rest of the semester. We assume you've picked a name for your team. Names like "satanic-leaf-tailed-gecko" or "pleasing-fungus-beetle" are totally fine, but for the sake of brevity (in case you ever need to type that thing), our examples will use "blobfish" (click [here](https://www.google.com/url?q=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FBlobfish&sa=D&sntz=1&usg=AFQjCNGIaySH9QeAsrzo9PCFXwtzhf8DRg) for a picture).

First, of all, **please follow the same instructions as you did in the beginning of the semester to** [**create a new repository for you group on bitbucket**](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fos161ubc%2Fworking-with-bitbucket&sa=D&sntz=1&usg=AFQjCNHcvI9JMJA76menpIZmkOl2oe4c0A). Assign permissions as follows:

* Read/write permissions to both partners
* Read permissions to Atlassian account **ubc.cpen331.2016@gmail.com**

For the sake of convenience, we'll assume your names are ALICE and BOB, and that you've chosen to use ALICE's code. If your names happen to be ALICE and BOB, great! If not, please adjust the following directions appropriately.

You are also allowed to use our assignment one solution set instead of your or your partner's code. Please contact the course staff if you're interested in this option, but we very much recommend that you use either your or your partner's code.

Now, let's push some code to this new repository. ALICE should do the following in her repository:

$ git checkout master

$ git log

# Verify this contains all the commits you want!

$ git remote set-url origin https://alice@bitbucket.org/alice/blobfish.git

# Your repository is probably called something other than "alice/blobfish.git" and the username is something other than "alice".

$ git push origin master --tags

BOB should now do the following:

$ cd ~/os161 $ mv src oldos161-src

# This will move Bob's existing code out of the way (feel free to back it up to

# some other location if you like)

$ git clone https://alice@bitbucket.org/alice/blobfish.git src

# Again, your repository is probably not called ""alice/blobfish.git"

# You will be prompted to enter your bitbucket username and password.

# Now you need to set up the instructor remote again

$ cd src

$ git remote add instructor http://dev.ece.ubc.ca/git/OS161

Now one of you should tag the repository:

$ cd ~/os161/src

$ git tag asst4-start

$ git push --tags

Now your partner should git pull and make sure the asst4-start tag got pulled successfully.

Don't forget to configure and build the new tree you're working in (especially BOB, since this is an entirely new repository for him)

***Once you have set up the repository, you need to commit and push a new*** submit/name ***file for your team. Include student IDs and names of both partners, for example:***

123456 Alice Chen

654321 Bob Jones

And then:

git commit -m "Group name file" submit/name

git push

**Fetch a new test into your (new) repository**

To make testing your code easier, we have created a new test for the system calls that you will implement for this assignment. Fetch the code from the instructor remote and merge it into your source base as follows:

cd ~/os161/src

git fetch instructor

git merge instructor/fsyscalltest

Since you added new files to the userland, don't forget to recompile it!

bmake

bmake install

Though this test won't run perfectly until you finish the next assignment (after the test passes, it will encounter the unimplemented exit system call and crash your kernel), it gives you a chance to implement system calls in smaller chunks, making the assignments less challenging.

**Prepare your source tree**

For this assignment you no longer need the files that you used to solve synchronization problems. Reconfigure your kernel using the DUMBVM configuration, the way you did for Assignment 2:

cd kern/conf

./config DUMBVM

cd ../compile/DUMBVM

bmake depend

bmake

bmake install

Tag your repository:

$ cd ~/cs161/os161

$ git tag asst4-start

$ git push --tags

**Step 2. Complete the code reading exercises.**

1. What are the ELF magic numbers?
2. What is the difference between UIO\_USERISPACE and UIO\_USERSPACE? When should one use UIO\_SYSSPACE instead?
3. Why can the struct uio that is used to read in a segment be allocated on the stack in load\_segment() (i.e., where does the memory read actually go)?
4. In runprogram(), why is it important to call vfs\_close() before going to usermode?
5. What function forces the processor to switch into usermode? Is this function machine dependent?
6. In what file are copyin and copyout defined? memmove? Why can't copyin and copyout be implemented as simply as memmove?
7. What (briefly) is the purpose of userptr\_t?
8. What is the numerical value of the exception code for a MIPS system call?
9. How many bytes is an instruction in MIPS? (Answer this by reading syscall() carefully, not by looking somewhere else.)
10. Why do you "probably want to change" the implementation of kill\_curthread()?
11. What would be required to implement a system call that took more than 4 arguments?
12. What is the purpose of the SYSCALL macro?
13. What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)
14. After reading syscalls-mips.S and syscall.c, you should be prepared to answer the following question: OS/161 supports 64-bit values; lseek() takes and returns a 64-bit offset value. Thus, lseek() takes a 32-bit file handle (arg0), a 64-bit offset (arg1), a 32-bit whence (arg2), and needs to return a 64-bit offset value. In void syscall(struct trapframe \*tf) where will you find each of the three arguments (in which registers) and how will you return the 64-bit offset?
15. As you were reading the code in runprogram.c and loadelf.c, you probably noticed how the kernel manipulates the files. Which kernel function is called to open a file? Which macro is called to read the file? What about to write a file? Which data structure is used in the kernel to represent an open file?
16. What is the purpose of VOP\_INCREF and VOP\_DECREF?

Save your code reading exercises:

mkdir ~/os161/src/submit/asst4

And put your answers into ~/os161/src/submit/asst4/asst4-answers.txt

Now tell git about your new file:

cd ~/os161/src

git add submit/asst4/asst4-answers.txt

**Step 3. Design and implementation**

The system calls you need to implement in this assignment are:

open(), read(), write(), lseek(), close(), dup2(), chdir(), and \_\_getcwd()

Although these system calls may seem to be tied to the filesystem, in fact, these system calls are really about manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the current working directory) is specific only to the process, but others (such as file offset) is specific to the process and file descriptor.

Begin by understanding what these system calls need to do, what arguments they take and what values they return by reading the man pages. The man pages are in your os161 source tree, and for your convenience we also placed them [here](http://www.google.com/url?q=http%3A%2F%2Fece.ubc.ca%2F~os161%2Fman%2F&sa=D&sntz=1&usg=AFQjCNG3oXHjfgid9tz3DFWLPAotgT1wmQ).

Your implementation of these system calls will probably consist of two parts: the actual *implementation* of the system call (e.g., the code that invokes the I/O to read or write a file, or advances the seek pointer, etc.) and the code that serves as the *transfer point* between the entry to the system call and your implementation. After doing the code reading exercises you already know that this transfer point code is located in kern/arch/mips/syscall/syscall.c. As evident from the code, the goal of the transfer point is to correctly transfer the arguments from the user program, invoke the function containing the implementation of the right system call, and transfer back the results. You can either work on the implementation first and on the transfer point code later or vice versa. You might want to write some unit tests that will test the implementation separately from the "transfer point" code.

**How to begin:**

1. Take a look at kern/test/fstest.c to learn about the functions available in the kernel for manipulating files. Another place to look for useful function is in the kern/vfs directory. These functions will be very helpful to you when implementing the solution to this assignment.
2. Carefully study kern/arch/mips/syscall/syscall.c. Look at the existing system calls as well as the comments at the top of this file. They will tell you exactly how to pass the arguments in and out of the system calls.

Besides correctly shuffling the arguments between user and kernel space and relaying the calls to kernel functions, a large part of this assignment involves tracking various process-related state needed for correct implementation of the system calls. For example, if a process opens a file and gets back a file descriptor, it then must be able to write into that file by passing the file descriptor into the kernel. So you need to keep track of the correspondence between the file descriptors and the underlying kernel objects used for representing files (what are they?). Similarly, if a processes changes the file offset by invoking the lseek system call, you need to remember that offset.

Though keeping track of this state seems simple at first, this becomes more complicated once you read the requirements for dup2(), which duplicates a file descriptor, mandating that the underlying file object could be accessed via two different descriptors. It's a good idea to read the [requirements for all system calls in the man page](http://www.google.com/url?q=http%3A%2F%2Fece.ubc.ca%2F~os161%2Fman%2Fsyscall%2F&sa=D&sntz=1&usg=AFQjCNFbklI1jDj3jWlyJQVi6y9bpQToQw)s before finalizing your design. Things become even more complicated in Assignment 5, where you'll need to implement fork(), which dictates that the file objects be shared between the parent and child processes; since these can run concurrently, you will need some careful synchronization! Though you don't need to worry about this for the current assignment, designing your file-tracking system with Assignment 5 in mind might make your life easier later.

**Important:** Before sitting down to write code, get together with your partner and write down the following **for every system call**:

* the arguments it takes
* the return values it might return
* the errors it must check for
* the information it needs to access and update inside the file table
* the functions and macros available in os161 that you can use for its implementation
* the potential race conditions and how they must be prevented (assume no user-level threads for this assignment).

After going through that exercise, you will have a pretty good idea of how to structure your implementation. ***We will require that you bring this document to the lab or to any meeting with a TA or the instructor before you can get any help on this assignment.***

It is important to know that for any given process, the first file descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr). These file descriptors should start out attached to the console device ("con:"), but your implementation must allow programs to use dup2() to change them to point elsewhere. Unix treats devices as special files. You can manipulate them in the same way as regular files, except not all operations that you can do on regular files will make sense for all devices (for example, it makes no sense to create a directory on a system console). For example, to open a console for reading and writing, you would pass "con:" to the function that opens regular files. The flags that you pass to the file opening function will depend on the desired access mode for the file (i.e., reading for stdin and writing for stdout and stderr).

You may want to add new files to the kernel during this assignment. Refresh your memory on how to do this by going over the solution to the code reading exercises for Assignment 1.

**Tips on testing your code:**

**Beware of the menu thread!** In this assignment you do not yet have a clean mechanism to prevent the kernel menu thread from competing with user program threads for console I/O. As a result, the kernel menu thread and any user programs you run will try to read from the console at the same time. Just imagine the mess that this is going to create! You are trying to test you "read" system call by having a user program read characters from STDIN, but the input just disappears! In the next assignment you will implement a clean solution to this problem by having the menu thread use waitpid to wait for the program that it runs. For now. make the menu thread go away by having it block on some semaphore or run an infinite while loop after it launches the user program. This dirty hack is actually okay for this assignment, because once your user program finishes running it will crash your system anyway due to an unimplemented \_exit system call (see below), which you will implement only in the next assignment. So you will only be able to run one program before your kernel crashes. That is okay. We allow for this type of hackery in this assignment.

To begin testing your code for this assignment, use the user-level program fsyscalltest, which you have fetched into your repository earlier. Recall that you can invoke a user program directly from the menu by typing "p" followed by the name of the program, for instance: p fsyscalltest. If you read the code in fsyscalltest.c, you will see that it consists of several functions with varying levels of test sophistication. Feel free to modify this code to test your implementation piecemeal: e.g., to test only the open() and close() system call, comment out everything except the code calling open() and close().

Note, that fsyscalltest (or any other user program) will not exit cleanly until you complete the next assignment. After it successfully completes the tests (assuming that your kernel correctly implemented the tested functionality), the C runtime will attempt to call an \_exit() system call, at which point your kernel will complain about unimplemented system call and then panic due to very minimal implementation of kill\_curthread(). Don't worry about this for now. You will deal with this in the next assignment. For now, just make sure that your kernel passes all the tests included in /testbin/fsyscalltest (it's okay to crash due to an unimplemented \_exit() call).

Next, test your kernel using /testbin/filetest (again, it's okay to crash due to an unimplemented remove() call).

Finally, test your code by running /testbin/badcall. It tests your code's ability to handle incorrect invocation of system calls. Though your code is not yet ready to pass many of the bad call tests, because they require the functionality of the next assignment, you should be able to pass many of them, such as:

open

read

write

close

lseek

chdir

dup2

\_\_getcwd

Again, for now you are allowed to crash after your kernel prints the Unknown syscall 68 message, but not before then! (That system call number refers to remove, which you have not implemented.)

**How we will mark your assignment**

Code reading (32 points total):

* 0 - no answer
* 1 - so-so answer
* 2 - correct coherent answer.

16\*2 = 32 maximum

Tests (100 points total):

* fsyscall test: max 7 points for each of the 5 subtests, 35 total
* filetest: 35 points max
* bigseek: 14 points max
* badcall: 16 points max (max 2 points each for each of the 8 badcall tests)

Code quality (66 points total):

* 0 - non-existent or incomprehensible
* 1 - kind-of sort-of
* 2 - basically ok
* 3 - dholland kind code

Take the final score and multiply by 22.

**Final Score:** Add up all the points for code reading, tests, and code quality.

**Step 4. Submit your assignment**

If you added any new files to your tree, add them using the git add command. For example, if you have added kern/include/filetable.h, do:

git add kern/include/filetable.h

Then submit as usual.

git commit -a -m "Solution to Assignment 4"

git push

git tag asst4-submit

git push --tags