This is an assignment in which you will integrate the components of the various assignments from the course into one cohesive system. From here, you will have a functional user-space operating system that can execute real programs! This is a challenging task, but should also be a rewarding one. You have plenty of room to be creative and, hopefully, you will have an operating system that you can interact with by the end!

Collaboration Policy

This assignment is intended to be implemented individually, but builds on the components that you have implemented throughout the semester! It may be to your benefit to reach back out to your partners from various exercises as you work through the various tasks of this assignment.

Unlike in previous assignments, there is no code review for this assignment! Do your best to put the pieces together, and you will submit a write-up alongside that describes your strategies you used for putting the implementation together.

Submission

You are asked to submit two deliverables for this assignment: ① the programming component, and ② your write-up describing your solution. Note, the write-up is not a formal document. Instead, use this as a log in which you document the things that you are trying, how you are testing each piece, and what you've seen that makes you feel confident you can move on. You can find more details in the "`Write Up`" section of this document.

Getting Started

This assignment is designed to run on the course virtual machine (e.g., `ssh stum2025@itbdcv-lnx04p.campus.pomona.edu`). The stencil for assignments in this class assume particular compiler versions and processor features, so using the VM will make your life easier.

I recommend that you work in the base repository from the `procs` assignment for this assignment. If you need to refer back to the components of this assignment, you can do so by accessing it here.

Deadlines

Section	Deadline	Submission URL
Programming Component	Dec 4 (EOD)	HW5 Programming
Resubmission Deadline	Dec 9 (EOD)	HW5 Programming (late due date)

1 High Level Description

The goal of this assignment is to build a singular system that utilizes the various subsystems that you have implemented over the course of the semester. At a high level, the goal is to have the binary file from your `shell` implementation stored on a file system disk image (which you can write to using a test from the `fs` assignment). Then, you can boot up a process from your `proc` assignment and, using memory allocated by your `mm` assignment, run the shell binary file.

Note that the integration of these components is a non-trivial task! At each step, there are challenges associated with putting the pieces together, and this document serves as a basis to help point out what some of those challenges might be in advance to help you foresee some of them. In addition, this assignment is very amenable to the *(Optional) Extensions* at the end of this document. Think about applying some of these extensions into your integration so that you can have a fully functional user-space operating system!

2 Tests

In this assignment, there are no pre-determined tests. However, you should test the various components of your integration as you go. If you put multiple pieces together, it would be to your benefit to test that they work together before moving on. As you do so, document these tests in your write-up.

3 Write-Up

As you integrate the various components of the system, document your progress! Note the goals of the session, the things that you've tried, what worked/didn't work, and the progress that you've made along the way. This does not need to be a formal document, but a way to track what you've done – which will be helpful as things come together! For example, the following may be a piece of the log that you submit:

```
November 7: Loading Shell onto File System Disk
- revisited FS test where files could be added to the file system
- created file on disk for / path
- tested that / existed on disk by creating a test that opens the file system, reads
    the first inode, finds the data blocks associated with the file, prints their
    contents, and exits
- test crashes... exploring why
- error with test: when reading first inode, attempted to consult superblock which gave
     first _free_ inode... re-writing test...
- test passed, moving onto creating file for shell binary in the path
- adding entry for shell binary in the / directory
```

4 Memory Management and Processes/File System Subsystems

As a first step, think about what it would mean to use your binary buddy allocator in place of the slab allocator in the `proc` and `fs` assignments. Instances of allocating and freeing objects as slab objects should be instead replaced with a request to the buddy allocation system. Unlike the slab allocator, the buddy allocator does not think of allocations in terms of "per-object allocators." Instead, you must modify these allocation requests to consider the *size* of the objects being allocated and freed.

Implementation Challenges

When we create the processes subsystem in the "`boot_test`" and "`fork_test`," we call the initialization functions for each of the subsystems. Think about what your subsystems your buddy allocator and processes needs for their execution. Does the order in which the various subsystems are initialized matter?

5 Loading Shell onto File System Disk

As a next step, you write the binary of your shell assignment onto your disk. During your file system assignment, you were tasked with implementing a test in which a file could be written onto the file system disk and accessed from the file system. You can use the same methodology to load your shell binary onto your disk. When you do so, make sure that the binary is accessible in the file system. This likely means implementing a directory entry in the root directory for the file so that it can be accessed by following a path through your file system interface.

Implementation Challenges

Ultimately, we are going to need to access the shell file from the disk to create a process that will run from this file. Given this, you may want to make a note of where in the file system certain important details associated with the file are maintained. In particular, it will be useful to know where the start of your `main` function is maintained on your disk so that you can call it from your process creation function.

In addition, this means that your binary needs to be compiled in such a way that it is runnable on in your OS kernel. Think about what this means in terms of the platform that you use to compile the binary. If your OS kernel needs a particular set of assembly instructions and has an intended platform to run on, then the source code that runs needs to match that platform.

Hint: We need to avoid any potential security features that the compiler OS might use to toggle where certain addresses are to be stored in our shell binary. To do so, add the following flags to your compiler before loading your shell onto the disk: `-fno-pie -no-pie`
Hint: If we want to know the address of a particular location in a binary, then we can find where it is in advance by using binary analysis tools, such as `objdump -d <binary name>`. This value would typically be found by the processes subsystem by scanning for a particular symbol, but for our purposes it is sufficient to hard-code the address into a special "`proc_shell_start`" function that you will implement in the next part.

6 Executing the Shell Binary

Thinking about your `init` process, which can boot and exit, let's create a new task for this process. Rather than a simple boot and exit, let's have the `init` process load the shell binary from the file system. This means that your process needs to have a mechanism to access the file system and read files from it. Fortunately, this simply requires linking your file system to your processes subsystem, so long as you know where your shell is stored in your file system.

Implementation Challenges

When creating a process to start executing a function, we have generally needed a pointer to that function that will have a particular signature. This was done by keeping a static copy of the function to execute in the global space of the main executable file of the "`boot_test`," "`fork_test`," etc... In this case, we need to have a variable of the *same type* as the executable functions so that our processes subsystem can refer to it.

Hint: We generally need to know where the important functionality of your binary is going to live in your simulated disk so that, when you load the file from the disk file. This means that we need to tell the compiler to turn off certain security features that will allow the program to be executable in user space (e.g., the stack). To do so, add the following flags to your compiler recompilation before loading your shell onto the disk: `-fno-stack-protector -z execstack`
Hint: Note that, when defining a function, what we are really doing is declaring a pointer with a definition attached to it. In the same way, we can declare a pointer and set its value to be a location in memory that we will then interpret as the data associated with the function body. More concretely, if we know where the "`main`" function lives in our binary, we can create a variable that refers to this function that we can generally execute.
Hint: If we want to execute the `main` function from the shell, then we need to have that value in memory (from the perspective of your OS kernel) somewhere in memory. This means reading the data from disk and saving it somewhere that your processes subsystem can access. At what point would we need this to happen if we are setting the main function of the `init` process to be the `main` function of the shell?

7 (Optional) Extensions

7.1 System Call Libraries

In addition to keeping your shell on the disk, similarly upload libraries that refer to implementations of the various tests that emulated system calls to your disk!

7.2 Custom Programs

Extend your shell implementation to refer to the environment in your emulated operating system rather than on your host. If you make a call to `ls`, it should create a process that calls the main function of this binary file *on your disk* and execute it!

7.3 Modifying Your Shell

As currently constructed, your shell takes a command from the user and creates a process on the *host* device when performing the inputted command. Modify your shell implementation to refer to processes on your OS rather than the host system!