# CS350 Operating Systems Fall 2019

## Assignment 2b

This assignment is a continuation of Assignment 2a. If you have not finished implementing the system calls from Assignment 2a, you should continue to work on them. Part of the testing for Assignment 2b will involve re-testing the system calls from Assignment 2a - so you can get some Assignment 2b credit for those calls even if they were not implemented properly when you submitted Assignment 2a. In addition, Assignment 2b requires you to implement one new system call, `execv`, which was not required for Assignment 2a.

# 1 Code Review

This section gives a brief overview of some parts of the kernel that are relevant to the new Assignment 2b requirements.

## 1.1 `kern/syscall`

This directory contains the files that are responsible for loading and running user-level programs, as well as basic and stub implementations of a few system call handlers.

`loadelf.c`: This file contains the functions responsible for loading an ELF executable from the filesystem into an address space. (ELF is the name of the executable format produced by cs350-gcc.)

`proc_syscalls.c`: This file is intended to hold the handlers for process-related system calls, including the handler for `execv`, which you are implementing for this assignment.

`runprogram.c`: This file contains the implementation of the kernel's `runprogram` command, which can be invoked from the kernel menu. The `runprogram` command is used to launch the first process run by the kernel. Typically, this process will be the ancestor of all other processes in the system. Studying the `runprogram` function should give you some ideas on how to implement `execv`. Think about how `runprogram`'s task is similar to `execv`'s, and how it is different.

## 1.2 `kern/arch/mips/`

This directory contains machine-specific code for basic kernel functions, such as handling system calls, exceptions and interrupts, context switches, and virtual memory.

`syscall/syscall.c`: This file contains the system call dispatcher function, called `syscall()`. As was described in Assignment 2a, you will need to modify this function to invoke your handler for `execv()`.

`locore/trap.c`: In this file, in addition to the kernel exception handler, you will find the function `enter_new_process`, which should be useful for your implementation of `execv`.

`vm/dumbvm.c`: This file contains the machine-specific part of OS/161's very simple implementation of virtual address spaces.

## 1.3 `kern/vm`

The `kern/vm` directory contains the machine-independent part of the kernel's virtual memory implementation.

`copyinout.c`: This file contains functions, such as `copyin` and `copyout` for moving data between kernel space and user space.

### 1.4 kern/include

**vfs.h:** This file describes the VFS interface, which the kernel uses to open and close files, e.g., program executable files. See the `runprogram` function for an example of how to use the VFS interface.

**vnode.h:** Opening a file using the VFS interface results in a `vnode` object representing the open file. The `vnode` object can be used to read data from and write data to the open file. `vnode.h` describes the `vnode` interface. See `loadelf.c` for an example of code that uses `vnode` operations to read data from a file.

**uio.h:** This file describes the `uio` interface. The kernel uses `uio` structures to describe a data transfer between a file and memory, between memory and a file, or between two locations in memory. `vnode` operations, such as `VOP_READ`, expect `uio` structures as parameters.

## 2 Implementation Requirements

For this assignment, you are expected to do two things:

1. Implement the OS/161 `execv` system call, including argument passing.
2. Ensure that it is also possible to pass arguments to the first process by modifying the kernel's `runprogram` command to support argument passing.

Your implementation should correctly and gracefully handle error conditions, and properly return the error codes as described on the man page. This is because application programs, including those used to test your kernel for this assignment, depend on the behaviour of the system calls as specified in the man pages. **Under no circumstances should an incorrect system call parameter cause your kernel to crash.**

All code changes for this assignment should be enclosed in `#if OPT_A2` statements, as was the case for Assignment 2a. Don't forget to add `#include "opt-A2.h"` at the top of any file for which you make changes for this assignment.

As was the case for Assignment 2a, your final, submitted kernel should not produce any output other than the normal boot and shutdown messages and the kernel menu prompt. We encourage you to use the `DEBUG` mechanism to generate kernel debugging output while you are testing your work, but make sure that all such debugging messages are turned off in the version of the kernel that you submit.

If your kernel produces lots of spurious output, it is more difficult for us to review the output produced by the user-level programs that we test with. If your kernel produces output other than the normal boot and shutdown messages, your assignment may be penalized.

### 2.1 execv

The `execv` system call replaces the address space of the calling process with a new address space containing a new program. After the `execv` system call, the process starts executing the new program, starting with its `main` function.

Be sure to review the manual page for the `execv` system call, which describes how `execv` is expected to behave. The system call man pages are located in the OS/161 source tree under `os161-1.99/man/syscall`. They are also available on-line through the course web page.

The second parameter to `execv` is an array of pointers to arguments (parameters). The idea is that `execv` passes these parameters to the new application program, which can access them using the `argc` and `argv` parameters to its `main` function. As discussed in Section 3, you should first get `execv` working without worrying about passing these arguments properly. Once `execv` is working without argument passing, you can then focus on getting argument passing working.

### 2.2 Argument Passing

The `execv` manual page specifies how the second parameter (the argument array) must be set up. Make sure you understand this before proceeding.

Argument passing means taking the arguments that are passed to `execv` and making them available to the new program that will start running in the process that does the `execv`. To do this, your kernel will need to retrieve these arguments from the address space of the original program (before destroying its address spaces) and then set up a properly structured argument array in the address space of the new program before it starts running. You will need to decide where in the new address space to place the arguments.

In addition to passing arguments to new programs through `execv`, your kernel is also expected to be able to pass arguments to the very first program that runs, i.e., to the program that is launched in response to the "p" (runprogram) kernel command. This is similar to passing arguments through `execv`, except for the fact that the arguments are coming directly from the kernel (which reads them from its command line) rather than from the program that is making the `execv` call. Therefore, once you have argument passing working for `execv`, it should be relatively simple to get argument passing working for `runprogram`.

## 3 Strategy

We cannot test your `execv` implementation unless the system calls from Assignment 2a (`fork`, `waitpid`, `_exit`) are implemented and working properly. Therefore, there is little point in working on `execv` until the Assignment 2a system calls are done. If you did not finish the Assignment 2a system calls, get them finished first. When we test your Assignment 2b, we will also re-test the Assignment 2a system calls, and part of the Assignment 2b marks will be awarded based on that re-testing.

When you are ready to start working on `execv`, you should start by getting `execv` to work without worrying about argument passing. There are some `execv` tests that do not require argument passing - make sure that your `execv` will pass those tests before you start on argument passing, which is the most challenging part of the assignment. Once you have an `execv` that works without argument passing, submit your kernel. If you get argument passing working, you can submit a revised version. If you do not, you will at least be able to get the marks for the partially working version of `execv` that you submitted before you started on argument passing.

Finally, you should work on argument passing. Argument passing for `runprogram` is very similar to argument passing for `execv`, so if you can get argument passing working for one, you should be able to get it working for the other.

## 4 Testing

Our testing of your Assignment 2b submission will be broken down into three categories:

1. re-running the Assignment 2a tests
2. running tests of `execv` without argument passing
3. running tests of `execv` and `runprogram` with argument passing

Although argument passing is the most challenging part of this assignment, the majority of the testing marks will be assigned based on the first two categories. Thus, it is best to ensure that you pass those tests before working on argument passing.

## 5 Configuring and Building

If you have already configured your kernel for Assignment 2a, you will not need to reconfigure it for Assignment 2b unless you add new source files to the kernel. If you wish to reconfigure your kernel, do it the same way it was done for Assignment 2a.

```
% cd cs350-os161/os161-1.99/kern/conf
% ./config ASST2
% cd ../compile/ASST2
% bmake depend
% bmake
% bmake install
```

For Assignment 2b, you build your kernel in `kern/compile/ASST2`, as you did for Assignment 2a.

Remember to completely recompile your kernel and user-level programs just before you submit the assignment. A common problem is not noticing that an erroneous change in header files that are shared between the kernel and user programs prevents the user programs from compiling. If we cannot compile the user-level applications, we cannot test your code!

# 6 What to Submit

You should submit your kernel source code using `cs350_submit` command, as you did for previous assignments. It is important that you use the `cs350_submit` command - do not use the regular `submit` command directly.

Assuming that you followed the standard OS/161 setup instructions, your OS/161 source code will be located in `$HOME/cs350-os161/os161-1.99`. To submit your work, you should run

```
/u/cs350/bin/cs350_submit 2b
```

in the directory `$HOME/cs350-os161/` This will package up your OS/161 kernel code and submit it to the course account.

**Important:** The `cs350_submit` script packages and submits everything under the `os161-1.99/kern` directory, except for the subtree `os161-1.99/kern/compile`. You are permitted to make changes to the OS/161 source code outside the `kern` subdirectory. For example, you might create a new test program under `user`. However, such changes will not be submitted when you run `cs350_submit`. Only your kernel code, under `os161-1.99/kern`, will be submitted.