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

**Assignment 5**

In this assignment you get to write some of the most important system calls in the operating systems: fork, execv, waitpid, getpid and \_exit. These system calls enable the creation of new user processes. After you complete this assignment you will be able to use the shell to launch various user programs, such as /bin/cat, /bin/cp and, of course, many devious test binaries in /testbin. Before you dive into design, [read the man pages](http://www.google.com/url?q=http%3A%2F%2Fece.ubc.ca%2F~os161%2Fman%2Fsyscall%2F&sa=D&sntz=1&usg=AFQjCNFbklI1jDj3jWlyJQVi6y9bpQToQw) to understand how these system calls must behave.

**Step 1. Prepare**

Tag your repository with a tag asst5-start and push the tags to your group repository, just like you did for the previous assignment.

**Step 2. Understand the requirements**

To master this assignment, you need to master the following concepts and engineer them in OS/161:

1. How to create a copy of an existing user process and make the new process return to user mode with a different return value than the original process. This is essentially the fork() system call, which we recommend to implement first. Code in runprogram.c and in the function it calls and in proc\_create\_runprogramwill be very helpful in figuring that out.

2. How to replace the address space of a process created via fork with a completely new address space and executable. This is, essentially, the execv system call. Again, runprogram.c is your friend as you try to figure it out!

3. How to copy in/out of the userspace the argument vector for the execv system call. Though conceptually simple, this code requires lots of pointer arithmetic and handling various corner cases.

3. How to manage the process IDs, so you don't run out as processes are created and retired.

4. How to synchronize the parent and child processes in waitpid() and \_exit().

5. How to manage the file table synchronization, given that starting from this assignment two or more processes might concurrently access the same file object (processes created via fork share the file objects).

Here is more details about the system calls you need to implement.

**getpid()**

A pid, or process ID, is a unique number that identifies a process. The implementation of getpid() is not terribly challenging, but pid allocation and reclamation are the important concepts that you must implement. It is not OK for your system to crash because over the lifetime of its execution you've used up all the pids. Design your pid system; implement all the tasks associated with pid maintenance, and only then implement getpid().

**fork(), execv(), waitpid(), \_exit()**

These system calls are probably the most difficult part of the assignment, but also the most rewarding. They enable multiprogramming and make OS/161 a much more useful entity.

**fork()** is the mechanism for creating new processes. It should make a copy of the invoking process and make sure that the parent and child processes each observe the correct return value (that is, 0 for the child and the newly created pid for the parent). You will want to think carefully through the design of fork() and consider it together with execv() to make sure that each system call is performing the correct functionality.

Carefully think about the state that you will need to copy as you create a new process. While the state that the process maintains depends to some extent on your implementation of system calls in Assignment 4, your process probably has an address space (which can be copied using as\_copy() in the current dumbvm system), some form of a file table (think what you need to do to correctly manage reference counts on the underlying files as you copy the file table), a pid (which will be different between the parent and the child) and a bunch of architectural state needed to return to user mode (maintained in the trapframe). How will you need to modify the child's trapframe to make sure that the child correctly returns to user mode? When do you need to copy the parent's trapframe? Remember that as soon as the parent is returns from the system call, its trapframe will be destroyed or modified!

**execv()**, although "only" a system call, is really the heart of this assignment. It is responsible for taking newly created processes and make them execute something useful (i.e., something different than what the parent is executing). Essentially, it must replace the existing address space with a brand new one for the new executable (created by calling as\_create in the current dumbvm system) and then run it. While this is similar to starting a process straight out of the kernel (as runprogram() does), it's not quite that simple. Remember that this call is coming out of userspace, into the kernel, and then returning back to userspace. You must manage the memory that travels across these boundaries very carefully. (Also, notice that runprogram() doesn't take an argument vector -- but this must of course be handled correctly in execv()). As you are replacing an old address space with a new one, you must not leak memory used by the old address space. Dispose of it properly!

Carefully read the [man page](http://www.google.com/url?q=http%3A%2F%2Fece.ubc.ca%2F~os161%2Fman%2Fsyscall%2Fexecv.html&sa=D&sntz=1&usg=AFQjCNHsMFu1UBc8OExQ8JebZ8b8HBIj5A) for execv() to understand the trickery in passing the potentially very large argument vector to/from userspace.

Although it may seem simple at first, **waitpid()** requires a fair bit of design. Read the specification carefully to understand the semantics, and consider these semantics from the ground up in your design.

The implementation of\_exit() is intimately connected to the implementation of waitpid(). They are essentially two halves of the same mechanism. Most of the time, the code for \_exit() will be simple and the code for waitpid() relatively complicated -- but it's perfectly viable to design it the other way around as well. If you find both are becoming extremely complicated, it may be a sign that you should rethink your design.

**A note on errors and error handling of system calls:**

The man pages in the OS/161 distribution contain a description of the error return values that you must return. If there are conditions that can happen that are not listed in the man page, return the most appropriate error code from kern/include/kern/errno.h. If none seem particularly appropriate, consider adding a new one. If you're adding an error code for a condition for which Unix has a standard error code symbol, use the same symbol if possible. If not, feel free to make up your own, but note that error codes should always begin with E, should not be EOF, etc. Consult Unix man pages to learn about Unix error codes; on Linux systems man errno will do the trick.

Note that if you add an error code to src/kern/include/kern/errno.h, you need to add a corresponding error message to the file src/kern/include/kern/errmsg.h.

**kill\_curthread()**

Feel free to write kill\_curthread() in as simple a manner as possible. Just keep in mind that essentially nothing about the current thread's userspace state can be trusted if it has suffered a fatal exception -- it must be taken off the processor in as judicious a manner as possible, but without returning execution to the user level.

**Step 3. Plan your work**

It is always a good idea to complete your assignment one step at a time and test every step. For example, in this assignment it makes most sense to implement the fork() system call first. Even if you only have fork() complete, you will be able to test it using /testbin/forktest (set no wait to 1 in forktest.c if you would like to run it before waitpid is implemented), and perhaps a few others. You may first implement fork() with a very simple pid assignment scheme, and improve on it later as you are designing the rest of the system calls. The concepts outlined at the beginning of Step 2 are a good starting point for dividing the work between the partners.

**Step 4. Test your code**

**Testing using the shell**

In user /bin/sh you will find a simple shell that will allow you to test your new system call interfaces. When executed, the shell prints a prompt, and allows you to type simple commands to run other programs. Each command and its argument list (an array of character pointers) is passed to the execv() system call, after calling fork() to get a new thread for its execution. The shell also allows you to run a job in the background using &. You can exit the shell by typing "exit".

Under OS/161, once you have the system calls for this assignment, you should be able to use the shell to execute the following user programs from the bin directory: cat, cp, false, pwd, and true.

Before you run the shell, take a look at the following comment in kern/main/menu.c:

\* Note that this does not wait for the subprogram to finish, but

\* returns immediately to the menu. This is usually not what you want,

\* so you should have it call your system-calls-assignment waitpid

\* code after forking.

\*

\* Also note that because the subprogram's thread uses the "args"

\* array and strings, until you do this a race condition exists

\* between that code and the menu input code.

Essentially, you will need to put some synchronization code into common\_prog() to make sure that the menu thread is not trying to run concurrently with your shell and does not compete with it for the console input/output.

**Error handling**

Your kernel must return expected error codes from system calls. You will have plenty of opportunities to test that functionality using the devious /testbin/badcall (which you should now be able to invoke from a shell). At this point your system should not crash no matter what the user program does. While we allowed your kernel to crash when it encounters an unimplemented system call in the previous assignment, starting from this assignment invoking an unimplemented system call must not crash the kernel. Neither should your kernel crash if it runs out of resourced (e.g., memory) as it is trying to execute user programs. Even/testbin/forkbomb should not crash your kernel, though we understand that it will become pretty useless once forkbomb is invoked (by the way, the same would happen on a real system!).

Take a look at the next section for the lists of tests your system needs to pass.

**Memory leaks**

You should implement your kernel so that it does not leak memory. Your execv should properly dispose of the old address space (and any other old unneeded state) and returned the memory back to the system. If you free compound structures that include pointers to other dynamically allocated memory, remember to free that memory as well (just like when you free a lock, free the string containing the lock's name). If a system call fails in the middle of execution (e.g., because of a bad pointer coming from user space) you must free up the resources allocated up to that point, so they are not hogged forever.

You can debug memory leaks by changing #undef LABELS to #define labels in vm/kmalloc.c and then invoking khdump from the kernel menu to examine the heap. For example:

OS/161 kernel [? for menu]: khdump

Remaining allocations from generation 3:

16 bytes at 0x80042e70, allocated at 0x80012d80

64 bytes at 0x80040140, allocated at 0x80006080

64 bytes at 0x800401c0, allocated at 0x8001162c

64 bytes at 0x80040200, allocated at 0x80006080

64 bytes at 0x80040340, allocated at 0x80006080

64 bytes at 0x800403c0, allocated at 0x80006080

64 bytes at 0x80040480, allocated at 0x80006080

64 bytes at 0x80040540, allocated at 0x80006080

64 bytes at 0x80040680, allocated at 0x80006080

128 bytes at 0x8003f780, allocated at 0x80023cfc

256 bytes at 0x80041b00, allocated at 0x8001162c

512 bytes at 0x80043c00, allocated at 0x8001162c

Operation took 1.195788680 seconds

You see that this command outputs a bunch of virtual addresses that were allocated via kmalloc and the corresponding addresses of code locations, where those items were allocated. To find out what these code locations are, you can use os161-addr2line (the OS/161 version of this standard Unix tool). For example, using an address from the example above:

os161-addr2line -e ~/Work/os161/root/kernel 0x80023cfc

/Users/UBC-home/Work/os161/src/kern/compile/DUMBVM/../../thread/thread.c:132

We can see that one of the structures still hanging around in the kernel was allocated at line 132 in thread.c, which in the codebase used in this example corresponds to:

thread = kmalloc(sizeof(\*thread));

in thread\_create().

That being said, before you implement virtual memory, your kernel will eventually run out of memory, because dumbvm does not properly return kernel pages as they are freed. So for this assignment feel free to increase the physical memory amount in sys161 by editing sys161.conf, as in:

31 mainboard ramsize=4194304 cpus=4

This does not mean that you are free to leak memory. Every kmalloc must be followed by a corresponding kfree when the memory is no longer needed. Just be prepared that if you execute many memory-intensive tests your system will eventually run out of memory as long as you are running under dumbvm.

**How we will mark your code**

We will run a subset of the following tests on your kernel (all from /testbin)

forktest

argtest

bigfile

crash

farm

faulter

forkbomb

multiexec

badcall

bigexec

Forktest and argtest are the simplest test programs, so first make sure that your kernel runs those. Badcall and bigexec are the most challenging.

**Tests** (60 points total):

* forktest 15 points
* argtest 15 points
* badcall 10 points
* bigfile 2 points
* crash 2 points
* farm 2 points
* faulter 2 points
* multiexec 2 points
* bigexec 10 points

If a test consists of multiple subtests (such as badcall), we award partial marks for passing some of the subtests. For example, if the code passes half of badcall tests, we award 5 points out of ten.

**Code quality** (39 points total):

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

Take the final score and mutiply by 13

**Final Score:** Add up all the points. 99 is the maximum

**Step 5. Submit your assignment**

Once you have completed your assignment commit and push everything to your group's repository on bitbucket. Then, as usual, run the following commands:

$ git status

You should see the output that is something like this:

On branch master

Your branch is up-to-date with 'bitbucket/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

# possibly a bunch of untracked files here

nothing added to commit but untracked files present (use "git add" to track)

You may or may not see the untracked files (and related message), but at least make sure that there are no uncommitted changes.

Then run:

**$ git tag asst5-submit**

**$ git diff asst5-start asst5-submit**

# This will list all of your asst2 changes. Please review it to make sure

# everything is there!

**$ git diff --stat asst5-start asst5-submit**

# This will list all the files you've modified, along with a rough indication of

# how much work has been done on each file. Make sure this looks right too!

**$ git push --tags**

If you ever need to update your submission tag, you might get the warning "fatal: tag 'asst5-submit' already exists". If this happens, just run:

**$ git tag -f asst5-submit**

Updated tag 'foo' (was d6a0fb9)

**$ git push --tags -f**