CS350 Operating Systems Winter 2021

Assignment 2

In this assignment, you are asked to implement several OS/161 process-related system calls along with locks and condition variables. Before you start implementing system calls, you should review and understand those parts of the OS/161 kernel that you will be modifying.

Important: before you start working on this assignment, you should reconfigure and rebuild your OS/161 kernel:

```
cd kern/conf
./config ASST2
cd ../compile/ASST2
bmake depend
bmake
bmake install
```

All kernel builds for this assignment should occur in the `kern/compile/ASST2` directory.

To build the OS/161 user-level applications, you need to run `bmake` in the top-level directory of the OS/161 source tree:

```
% cd cs350-os161/os161-1.99
% bmake
% bmake install
```

Generally, you should not have to rebuild those applications every time you build a new kernel. However, there are certain header files, e.g, in `kern/include/kern` that are used by the kernel and by the user-level application programs. In the unlikely event that you make changes to these files, you must rebuild the user-level code.

It is always OK to rebuild the user-level applications. If you are getting any weird, unexpected behaviour from those applications, it is a good idea to rebuild them just to be on the safe side.

More importantly, **make sure 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!

1 Implement Kernel Synchronization Primitives

The OS/161 kernel includes four types of synchronization primitives: spinlocks, semaphores, locks, and condition variables. Spinlocks and semaphores are already implemented. Locks and condition variables are not – it is your task to implement them.

1.1 Implement Locks

Your first task is to implement locks. The interface for the lock structure is defined in `kern/include/synch.h`. Stub code is provided in `kern/threads/synch.c`. You can use the implementation of semaphores as a model, but **do not build your lock implementation on top of semaphores** or you will be penalized. In other words, your lock implementation should not use `sem_create(), P(), V()` or any of the other functions from the semaphore interface.

Locks are used throughout the OS/161 kernel. You will need properly functioning locks for this and future assignments to ensure that the kernel's threads are properly synchronized. Because of this, implementing locks correctly - though not difficult - is the most important part of this assignment. **Make sure that you get locks working before moving on to the other parts of the assignment.**

1.2 Implement Condition Variables

Use lock_do_i_hold

The second task is to implement condition variables for OS/161. The interface for the `cv` structure is defined in `kern/include/synch.h` and stub code is provided in `kern/thread/synch.c`. Each condition variable is intended to work with a lock: condition variables are only used *from within the critical section that is protected by the lock*

1.3 Testing Locks and Condition Variables

Cannot run test back to back until A3,
use command sys161 kernel-ASSTX "sy2;q", run sy1, sy2, uw1

The file `kern/test/synchtest.c` implements a simple test case for locks, and another for condition variables. You can run the lock test from the kernel menu by issuing the `sy2` command, e.g,:

```
% sys161 kernel "sy2;q"
```

AFTER testing with 1 CPU, increase the number of CPUs in cs350-os161/root/sys161.conf
to increase the change of race conditions, line 134. Remember to change back to 1 CPU

Similarly, the `sy3` command will run the condition variable test. If the lock test reports "Lock test done" without reporting any failure messages, it has succeeded. The output from the condition variable test should be self-explanatory.

Testing synchronization primitives like locks and condition variables is difficult. Both `sy2` and `sy3` are subject to false positives. In other words, an incorrect lock or condition variable implementation may pass these tests. However, if your implementation fails a test, there is definitely a problem. Since the synchronization tests are not perfect, we may use code inspection - in addition to testing - to evaluate your lock and condition variable implementations.

Although you are free to implement locks however you want, **you should not modify any of the kernel's test programs**, i.e., do not modify any of the files in `kern/test`. Furthermore, you should not make any changes to the way that the tests are invoked, e.g., do not change "`sy2`" to "`sy2a`".

2 System Calls

This section gives a brief overview of some parts of the kernel that you should become familiar with in order to implement the required system calls.

2.1 `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.

`proc_syscalls.c`: This file is intended to hold the handlers for process-related system calls, including the calls that you are implementing for this assignment. Currently, it contains a partial implementation of a handler for `_exit()` and stub handlers for `getpid()` and `waitpid()`.

`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.

2.2 `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.

`locore/trap.c`: This file contains the function `mips_trap()`, which is the first kernel C function that is called after an exception, system call, or interrupt returns control to the kernel. (`mips_trap()` gets called by the assembly language exception handler.)

`syscall/syscall.c`: This file contains the system call dispatcher function, called `syscall()`. This function, which is invoked by `mips_trap()` determines which kind of system call has occured, and calls the appropriate handler for that type of system call. As provided to you, `syscall()` will properly invoke the handlers for a few system calls. However, you will need to modify this function to invoke your handler for `fork()`. In this file, you will also find a stub function called `enter_forked_process()`. This is intended to be the function that is used to cause a newly-forked process to switch to user-mode for the first time. When you implement `enter_forked_process()`, you will want to call `mips_usermode()` (from `locore/trap.c`) to actually cause the switch from kernel mode to user mode.

2.3 kern/include

The `kern/include` directory contains 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.)

2.4 kern/vm

The `kern/vm` directory contains the machine-independent part of the kernel's virtual memory implementation. Although you do not need to modify the virtual memory implementation for this assignment, some functions implemented here are relevant to the assignment.

`copyinout.c`: This file contains functions, such as `copyin()` and `copyout` for moving data between kernel space and user space. See the partial implementations of the handlers for the `write()` and `waitpid()` system calls for examples of how these functions can be used.

2.5 In user

The `user` directory contains all of the user level applications, which can be used to test OS/161. Don't forget that the user level applications are built and installed separately from the kernel. All of the user programs can be built by running `bmake` and then `bmake install` in the top-level diretory (`os161-1.99`).

3 Implementation Requirements

All code changes for this assignment should be enclosed in `#if OPT_A2` statements. For example:

```
#if OPT_A2
   // code you created or modified for ASST2 goes here
#else
   // old (pre-A2) version of the code goes here,
   //  and is ignored by the compiler when you compile ASST2
   // the ``else'' part is optional and can be left
   // out if you are just inserting new code for ASST2
#endif /* OPT_A2 */
```

For this to work, you must add `#include "opt-A2.h"` at the top of any file for which you make changes for this assignment.

For this assignment, you are expected to implement the following OS/161 system calls and features:

- `fork`
- `getpid`
- `waitpid`
- `_exit`
- `execv`
- Ensure that it is also possible to pass arguments to the first process by modifying the kernel's `runprogram` command to support argument passing.

`fork` enables multiprogramming and makes OS/161 much more useful. `_exit` and `waitpid` are closely related to each other, since `_exit` allows the terminating process to specify an exit status code, and `waitpid` allows another process to obtain that code. You are not required to implement the `WAIT_ANY`, `WAIT_MYPGRP`, `WNOHANG`, and `WUNTRACED` flags for `waitpid()` - see `kern/include/kern/wait.h`.

To help get you started, there is a partially-implemented handler for `_exit` already in place, as well as stub implementatations of handlers for `getpid` and `waitpid`. You will need to complete the implementations of these handlers, and also create and implement a handler for `fork`.

There is a man (manual) page for each OS/161 system call. These manual pages describe the expected behaviour of the system calls and specify the values expected to be returned by the system calls, including the error numbers that they may return. **You should consider these manual pages to be part of the specification of this assignment, since they describe the way that that system calls that you are implementing are 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.

Your system call implementations should correctly and gracefully handle error conditions, and properly return the error codes as described on the man pages. 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.**

Integer codes for system calls are listed in `kern/include/kern/syscall.h`. The file `user/include/unistd.h` contains the user-level function prototypes for OS/161 system calls. These describe how a system call is made from within a user-level application. The file `kern/include/syscall.h` contains the kernel's prototypes for its internal system call handling functions. You will find prototypes for the handlers for `waitpid`, `_exit` and `getpid` there. Don't forget to add a prototype for your new `fork()` handler function to this file.

3.1 Process IDs

A PID, or process ID, is a unique number that identifies a process. You should carefully review the manual pages for `fork`, `_exit`, and `waitpid` to understand how PIDs are expected to work.

For the purposes of this assignment, you should ensure that a process can use `waitpid` to obtain the exit status of any of its children, and that a process may not use `waitpid` to obtain the exit status of any other processes. In the terminology used on the `waitpid` manual page, you should assume that a process is "interested" in its children, but is not interested in any other processes.

3.2 `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 **??**, 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.

3.3 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`.

3.4 Silence is Golden

Your final, submitted kernel should not produce any output other than the normal boot and shutdown messages and the kernel menu prompt. We enourage 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.

4 Testing

The kernel's `runprogram` command, which was described in Section 2.1, will allow you launch a process to run a user-level application program. This is handy for testing that your system calls work. Without making any modifications to the base OS/161 code, you should be able to run the `testbin/palin` user program, which is a simple palindrome tester. `testbin/palin` uses only `write` to the console and `_exit`, both of which are partially implemented in the OS/161 base code.

OS/161 includes a number of application programs that you can use. The `user/bin` and `user/sbin` directories contain a number of standard utility programs, such as a command shell. In addition, the `user/testbin` and `user/uw-testbin` directories contain a variety of programs that can be used to conduct some simple tests of your OS/161 kernel. The A2 hints (on-line) will identify some specific programs that we will be using to test your submission. Any of these programs can be launched directly from the kernel using the `runprogram` command.

5 What to Submit

You should submit your kernel source code using `cs350_submit` command, as you did for Assignment 1. 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 2
```

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.

You can submit multiple times, and only your last submission will be used. Just be careful that your submitted version works and that you submit well before the deadline.