CS350 Operating Systems Winter 2016

Assignment 2a

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

1 Code Review

This section gives a brief overview of some parts of the kernel that you should become familiar with.

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.

`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.

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.

`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.

1.3 `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.)

1.4 `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.

1.5 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`).

2 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.

If in Assignment 1 you wrapped any new code with `#if OPT_A1`, it will also be included in your build when you compile for Assignment 2.

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

- `fork`
- `getpid`
- `waitpid`
- `_exit`

`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.

2.1 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.

2.2 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.

3 Testing

The kernel's `runprogram` command, which was described in Section 1.1, 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.

4 Configuring and Building

Before you do any coding for Assignment 2a, you will need to reconfigure your kernel for this assignment. Follow the same procedure that you used to configure for Assignment 1, but use the Assignment 2 configuration file instead:

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

This will configure, build and install your Assignment 2a kernel. Note that you build your kernel in `kern/compile/ASST2`, not `kern/compile/ASST1`.

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!

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 2a
```

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.