CS350 Operating Systems Winter 2011

Assignment Two

This assignment has three parts. The first part, described in Section 1, requires you to read the OS/161 code and to answer a few questions based on your reading. The second part, described in Sections 2-5, requires you to add functionality to the OS/161 kernel. The third part, described in Section 6, asks you to prepare a document describing the design decisions you made when implementing processes and system calls in your kernel.

1 Code Review

As was the case for the first OS/161 assignment, you should begin with a careful review of the existing OS/161 code with which you will be working. The rest of this section of the assignment identifies some specific files for you to consider. There are also a number of questions that you should be able to answer after you have read and understood the code. You are expected to prepare a brief document, in PDF format, containing your answers to these questions. This document should be at most two pages long using an 11 point (or larger) font and 1 inch (or larger) top, bottom and side margins. In your document, please number each of your answers and ensure that your answer numbers correspond to the question numbers. Your PDF document should be placed in a file called `codeanswers2.pdf`.

In `kern/userprog`

This directory contains the files that are responsible for loading and running user-level programs. Currently, the only files in the directory are `loadelf.c`, `runprogram.c`, and `uio.c`, although you may want to add more of your own during this assignment. Understanding these files is the key to getting started with the implementation of multiprogramming.

`loadelf.c`: This file contains the functions responsible for loading an ELF executable from the filesystem into virtual memory space. (ELF is the name of the executable format produced by cs350-gcc.) As provided to you, OS/161 has a very limited virtual memory implementation. Virtual addresses are translated to physical addresses using a very simple translation scheme and the total size of the virtual address spaces of all running processes cannot be any larger than the size of physical memory.

`runprogram.c`: This file contains only one function, `runprogram()`, which is responsible for launching an application program from the kernel menu. It is a good base for writing the `execv()` system call, but only a base. You will need to determine what is required for `execv()` that `runprogram()` does not concern itself with. You will also be expected to modify the `runprogram` command as part of this assignment, as described in Section 2.3.

`uio.c`: This file contains functions for moving data between kernel and user space. Knowing when and how to cross this boundary is critical to properly implementing user-level programs. You should also examine the code in `kern/lib/copyinout.c` and figure out when these functions should be used in implementing your system calls.

Question 1. What is the purpose of `copyin()` and `copyout()`?

Question 2. What is the difference between `UIO_USERISPACE` and `UIO_USERSPACE`? When should one use `UIO_SYSSPACE` instead?

Question 3. In `runprogram()`, why is it important to call `vfs_close()` before going to user mode?

Question 4. Which kernel function is used to make a thread switch to executing user-level code?

Question 5. What (briefly) is the purpose of `userptr_t`?

In `kern/arch/mips/mips`

Exceptions are the key to operating systems; they are the mechanism that enables the OS to regain control of execution and therefore do its job. When the OS boots, it installs an "exception handler" (carefully crafted assembly code) at a specific address in memory. When the processor raises an exception, it invokes this handler, which sets up a "trap frame" and calls into the operating system. Since "exception" is such an overloaded term in computer science, operating system lingo for an exception is a "trap" – when the OS traps execution. System calls are implemented as one type of exception, and `syscall.c` handles traps that happen to be system calls.

`trap.c`: `mips_trap()` is the key function for returning control to the operating system. This is the C function that gets called by the assembly exception handler. `kill_curthread()` is the function for handling broken user programs; when the processor is in usermode and hits something it can't handle (say, a bad instruction), it raises an exception. There's no way to recover from this, so the OS needs to kill off the process. Part of this assignment will be to write a useful version of this function.

`syscall.c`: `mips_syscall()` is the function that delegates the actual work of a system call to the kernel function that implements it. You will need to write those functions, and ensure that they are invoked from `mips_syscall()`. Notice that `reboot()` is the only case currently handled. You will also find a function, `md_forkentry()`, which is intended to be the function that is run by the new thread (implementing the new process) that your kernel will need to create in response to a `fork()` system call.

Question 6. Why do you "probably want to change" the implementation of `kill_curthread()`?

Question 7. At the time that `mips_syscall()` is invoked, are interrupts enabled or disabled? How about when `kill_curthread()` is invoked?

In `kern/include`

These are 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.)

In `kern/lib`

These are library routines that are used throughout the kernel, e.g., for managing queues and for kernel memory allocation.

In `kern/fs`

This directory contains two subdirectories. `kern/fs/vfs` contains the file-system independent layer (`VFS` stands for "Virtual File System"). It establishes a framework into which one can add new file system types easily. The kernel accesses both files and I/O devices, such as the console, through *vnodes*, which are implemented by the VFS layer. You can see examples of vnodes in use in `runprogram.c` and `loadelf.c`. You will want to look at `vfs.h` and `vnode.h` before looking at this directory.

`kern/fs/sfs` contains the implementation of the simple file system that OS/161 uses by default. This file system is not used in this assignment.

Question 8. Which kernel function is used to open a file or device and obtain a vnode?

Question 9. What operations can you do on a vnode? If two different processes open the same file, do we need to create two vnodes?

In `lib/crt0`

This is the user program startup code. There's only one file in here, mips-crt0.S, which contains the MIPS assembly code that receives control first when a user-level program is started. It calls the user program's `main()`. This is the code that your `execv()` implementation will be interfacing to, so be sure to check which values it expects to appear in each register, and so forth.

In `lib/libc`

This is the user-level C library. There's obviously a lot of code here. We don't expect you to read it all.

`errno.c`: This is where the global variable errno is defined.

`syscalls-mips.S`: This file contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls.

`syscalls.S`: This file is created from syscalls-mips.S at compile time and is the actual file assembled into the C library. The actual names of the system calls are placed in this file using a script called `callno-parse.sh` that reads them from the kernel's header files. This avoids having to make a second list of the system calls. In a real system, typically each system call stub is placed in its own source file, to allow selectively linking them in. OS/161 puts them all together to simplify the makefiles.

2 Implementation Requirements

For this assignment, you are expected to implement exception handling and several OS/161 system calls. You are also expected to enhance the kernel's `runprogram` command to support argument passing.

All code changes for this assignment should be enclosed in `#if OPT_A2` statements, in much the same way that you used `#if OPT_A1` for Assignment 1:

```
#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.

By default, any code changes that you made for Assignment 1 (which should already be wrapped with `#if OPT_A1`) will *also* be included in your build when you compile for Assignment 2.

2.1 System Calls

You are expected to implement the following OS/161 system calls:

- `open`, `close`, `read`, `write`
- `fork`, `getpid`, `waitpid`, `_exit`
- `execv`

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 `os161-1.11/man/syscall`. They are also available on-line through the course web page.

It is expected that your system calls will correctly and gracefully handle all system call error conditions, and that they will 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/callno.h`. The file `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 want to create a kernel handler function for each system call that your kernel handles, and place a prototype for that function in `kern/include/syscall.h`.

For example, as provided to you, the OS/161 kernel supports only one system call: `reboot`. The user-level function prototype for `reboot`, which is found in `include/unistd.h`, looks like this:

```
int reboot(int code);
```

The kernel's internal handler function for the reboot system call is called `sys_reboot`. Its prototype, found in `kern/include/syscall.h`, looks like this:

```
int sys_reboot(int code);
```

Implementing file system calls

Applications do their input and output using so-called *file descriptors* that are provided by the kernel. A file descriptor may refer to an open file or to a device such as the console. The `open` system call creates a new file descriptor referring to a specified file. Subsequent `read` or `write` system calls using that file descriptor cause data to be read from or written to that file.

Whenever a new process is created, the kernel should arrange that three file descriptors are available to the process when it starts running. These are standard input, standard output, and standard error (descriptors 0, 1, and 2, respectively). The kernel should arrange that `read`s from the standard input read from the console device ("con:"), and that `write`s to the standard output or standard error are sent to the console device.

To access a file, applications would first use the `open` system call to obtain a file descriptor that refers to the file, and then use `read` and `write` to access the file data.

These system calls are largely about manipulation of file descriptors, or process-specific filesystem state. You need to think about the state you need to maintain, how to organize it, and when and how it has to change.

Process IDs and `getpid`

A pid, or process ID, is a unique number that identifies a process. pid allocation and reclamation are the important concepts that you must implement. It is not OK for your kernel to crash because it has, over its lifetime, consumed all of the available pids. Thus, your kernel should be able to reuse pids when they are no longer needed for their original purpose.

You should carefully review the manual pages for `fork`, `_exit`, and `waitpid` to understand how PIDs are expected to work. These manual pages do not completely define the process model for OS/161 but they give some ground rules which you are expected to follow. You will need to more completely define a process model (consistent with the given ground rules) before you can implement PIDs. In particular, you will need to decide which processes a given process is permitted to wait on (via `waitpid`). Make these decisions before implementing `getpid`, `waitpid`, and `fork`.

Implementing `fork`, `waitpid`, and `_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. As noted above, make sure that you understand how your PIDs will work before working on these system calls.

Implementing `execv`

You will not be able to test `execv` if `fork` is not working, so get `fork` done before starting on `execv`.

There are two parts to `execv`. The first part involves replacing the currently executing application program with a new program. The second part involves properly setting up the argument strings for the new program. You should get the first part working first, and then worry about passing the argument strings.

Passing arguments involves using the `args` parameter to `execv` to set up the `argv` and `argc` parameters to the `main()` function in the application program that `execv` is launching. `argv` is an array of strings (i.e., an array of pointers to character arrays), and `argc` indicates the number of arguments. The man page for the `execv` system call describes argument passing in more detail. Read it carefully.

Adding support for argument passing to `execv` is very similar to adding support for argument passing for `runprogram`, which is described in Section 2.3. Once you can do one, you should be able to do the other.

2.2 Exception Handling

There are a number of types of exceptions that may be generated by the MIPS processor and that must be handled by the kernel. Some of these exceptions, such as those related to virtual memory, are already handled by the OS/161 kernel as it is provided to you. However, other types of "fatal" exceptions that may occur during the execution of a user-level program are not handled well. Instead of terminating the process that caused the exception, the kernel (as provided to you) simply panics and shuts down.

For this assignment, you are expected to modify the kernel's exception handling so that fatal exceptions (e.g., illegal instructions) that occur in user-level code are handled by terminating the process that caused the exception. **Under no circumstances should such an exception in an application program cause your kernel to crash or shutdown.**

2.3 Modifying `runprogram`

The kernel's `runprogram` command, which you will find implemented in `kern/userprog/runprogram.c`, can be used to launch an application program from the kernel menu. One important thing that `runprogram` cannot do is to pass arguments to the application program that is being launched. Thus, as provided to you, `runprogram` is useful for launching application programs that do not expect command line arguments, but not for launching programs that do expect arguments.

You are required to modify `runprogram` so that it is capable of passing arguments from the kernel command line to the application that `runprogram` launches. For example, you should be able to run the `add` program from the `testbin` directory and get the proper result:

```
% sys161 kernel "p testbin/add 2 4;q"
Answer: 6
```

`runprogram` should set up arguments for the new program the same way that `execv` is supposed to set them up. The man page for the `execv` system call describes this in more detail.

3 Testing

The kernel's `runprogram` command, which was described in Section 2.3, is sufficient to allow you to run one user-level application program at a time. This is handy for testing that your system calls work.

OS/161 includes a number of application programs that you can use. The `bin` and `sbin` directories contain a number of standard utility programs, such as a command shell. In addition, the `testbin` directory contains a variety of relatively simple programs that are intended primarily to test the OS/161 kernel. These are the programs that you are most likely to find useful for the assignment. Any of these programs can be launched directly from the kernel using the `runprogram` command. An example of this was given in Section 2.3.

The source for all of the OS/161 application programs is under `cs350-os161/os161-1.11` in the `bin`, `sbin`, and `testbin` subdirectories. Once they are compiled, the application binaries are installed into `cs350-os161/root`, into the `bin`, `sbin` and `testbin` subdirectories there.

4 Configuring and Building

Before you do any coding for Assignment 2, 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/os161-1.11/kern/conf
% ./config ASST2
% cd ../compile/ASST2
% make depend
% make
% make install
```

This will configure, build and install your Assignment 2 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 `make` in the top-level directory of the OS/161 source tree:

```
% cd cs350-os161/os161-1.11
% make
```

You should not make any changes to the existing user-level applications. 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. If you make changes to these files, you must rebuild the user-level code. You do so like this:

```
% cd cs350-os161/os161-1.11
% make clean
% make
```

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 issue is not noticing that an erroneous change in header files prevents the user programs from compiling. We will be testing your kernel using our copies of the user-level applications - if we cannot build those applications, then we cannot test your code!

5 Strategy

Because of the the nature of the system calls you will be implementing, and because of the way that your work will be tested, there are some dependencies among the different parts of the assignment. Because of these dependencies, you should work on the different parts of this assignment in a specific order, as follows:

1. Implement console `write` and `_exit`

   You should get both of these calls working *before* you continue with the rest of the assignment. One important reason for this is that this is the bare minimum that you will need to be able to do anything useful in Assignment 3. A second reason is that the testing of other parts of *this* assignment depends on `write` (to the console) and `_exit` working properly. If we are unable to test these other parts of the assignment, then you will not get credit for the work that you put into them! So: get these two system calls working first.

- `_exit`: To get started, don't worry about storing the exit code and making it available to other processes. Focus instead on ensuring that `_exit` cleanly terminates the process the calls it. Later you can ensure that `_exit` meets its full specification.
- `write`: Be sure that the `write` system call is able to correctly write data to standard output, which should be connected to the console device. This will permit user-level programs to call printf to print output to stdout. This, in turn, will make it possible to demonstrate that user-level programs in this and future assignments are executing as expected (or not). For starters, don't worry about being able to write to other file descriptors, only worry about writing to stdout. Later, you can ensure that `write` meets its full specification.

2. Implement argument passing

   The second thing that you should do is to modify `runprogram` so that it is capable of passing arguments (see Section 2.3). Some of the test programs that we will use to test other parts of this assignment expect to be passed command line arguments.

3. Other system calls, except for `execv`

   Once argument passing, console write, and `_exit` are finished, you should begin working on the remaining system calls, except for `execv`. These calls come in two groups. One group consists of the file-related system calls, i.e., `open`, `close`, `read`, and `write`. For the most part, these will be tested as a group. For example, we simply cannot test reads and writes from a file, if the file cannot be opened and closed. The second group includes `fork`, `waitpid` and `_exit`. Again, there are strong interdependencies among the calls in this group, e.g., we can't test `waitpid` if `fork` and `_exit` do not also work.

   You can tackle these two groups of calls in either order. However, we strongly recommend that you plan to finish the system calls of at least one group, rather than attempting everything and leaving a lot of things half-finished.

4. Implement exception handling and `execv`

   Nothing depends on these, so you can do them last.

6 Design Document

You are expected to prepare a short design document describing your implementation of these system calls. You should arrange your design document as answers to the following questions:

Question 1 (file descriptors): How are file descriptors implemented? What kernel data structures were created to manage file descriptors? Briefly describe the implementation of `open`, `close`, `read`, and `write`.

Question 2 (process identifiers): Briefly explain how you implemented PIDs. How does your kernel generate a PID for each new process? How does your kernel determines that a PID is no longer needed by the process to which it was assigned (and is therefore available for re-use)? Briefly describe the implementation of `fork`, `getpid`, and `_exit`.

Question 3 (waiting for processes): Briefly explain how your kernel implements the waiting required by `waitpid`. Did you use synchronization primitives? If so, which ones and how are they used? What restrictions, if any, have you imposed on which processes a process is permitted to wait for?

Question 4 (argument passing): How did you implement `argc` and `argv` for `execv` and `runprogram`? Where are the arguments placed when they are passed to the new process? Briefly describe the implementation of `execv`.

Use diagrams when appropriate. Remember, a picture is worth a thousand words. If you created kernel data structures that are shared by multiple threads, make sure to describe how access to these data structures is synchronized.

It is OK if portions of the design are not yet implemented, and you may receive some credit for your answers even if the design is not implemented. However, if parts of the design described in your answers are not implemented, then **your answers must clearly indicate which parts of the design are not implemented as described.**

Submission of answers that do not match the submitted implementation (and which do not flag any such differences) will be considered an act of academic dishonesty.

Please prepare your design document as a PDF file called `design2.pdf`. This document should be at most three pages long using an 11 point (or larger) font and 1 inch (or larger) top, bottom and side margins. Two pages should be sufficient if your description is brief and clearly written – do not feel obligated to submit a three-page document. Also, make sure that your PDF documents are laid out in portrait mode, not landscape mode.

Your design document should be self-contained. That is, it should be possible for us to understand your design without reference to your source code or to any other external material.

7 What to Submit

You should submit your kernel source code, your code question answers, and your design question answers using the `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/os161-1.11`. To submit your work, you should

1. place `codeanswers2.pdf` in the directory `$HOME/cs350-os161/`
2. place `design2.pdf` in the directory `$HOME/cs350-os161/`
3. 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, along with the two PDF files, to the course account.

Important: The `cs350_submit` script packages and submits everything under the `os161-1.11/kern` directory, except for the subtree `os161-1.11/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 `testbin`. However, such changes will not be submitted when you run `cs350_submit`. Only your kernel code, under `os161-1.11/kern`, will be submitted.