CS350 **Operating Systems** **Fall 2007**

Assignment Two (OS/161)

Version 1.0: Oct. 10

Note 1: we will create, maintain and modify a web page of hints for this assignment. It will be located at `http://www.student.cs.uwaterloo.ca/~cs350/F07/assignments/a2-hints.html`

Note 2: please see the man pages for descriptions of the system calls. The man pages specify the system call parameters, the expected behaviour, return values and error codes.

1 Key Requirements

This section outlines the parts of your implementation that are absolutely essential. This and future assignments rely on correctly working versions of the items listed below.

1.1 Bare Minimum Requirements

These are bare minimum requirements that absolutely must work properly in order for you to be able to do anything useful for the next assignment. We recommend that you implement and debug these functions in the order presented here and worry about widening the scope of their functionality after these and the calls described in 1.2 are working.

- `_exit`: Be sure to implement a correctly functioning `_exit` system call. If you aren't able to get this implemented and debugged properly it will be difficult to write and execute user-level programs in this and future assignments. To start with don't worry about how other processes get the exit value. Ensure that when a process calls `_exit` that the proper exit code is obtained and that the program does not continue to execute. Later you can/should ensure that it meets all of the specification for the `_exit` system call.
- `write`: Be sure that the `write` system call is able to at least correctly write data to the system console. This will permit user-level programs to call printf to print output to stdout. This will make it possible to demonstrate that programs in this and future assignments are executing as expected (or not). For now don't worry about being able to write to other files, only worry about writing to stdout. Later you can/should ensure that it meets all of the specification for the `write` system call.

1.2 Slightly more than Bare Minimum Requirements

Meeting these requirements will permit you to do more interesting and realistic testing of your virtual memory system in the next assignment. Again, we recommend that you implement and debug these functions in the order presented here and worry about widening the scope of their functionality and then implementing other system calls only after these are working properly.

- `fork`: This will allow you to create multiple user processes.
- Passing arguments using `argc` and `argv`: Modify runprogram in kern/userprog/runprogram.c so that it is passed arguments from the command line that invokes it. For example, you should be able to run the add program and get the proper result.

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

  Note that once you have this working it shouldn't take much work to implement execv.
- `waitpid`: This will allow one process to fork another, wait for its completion and check its exit status.
- `execv`: If you can pass arguments on the command line to `runprogram` it shouldn't be difficult to implement the `execv` system call.

2 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 important 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 should read these files and prepare short answers to the questions. You are expected to prepare a brief document, in PDF format, containing your answers. 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 `codeanswers.pdf` in the directory os161-1.11.

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. Note that to answer some of the questions, you will have to look in files outside this directory.

`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 running 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. Additionally, once you have designed your process system, `runprogram()` should be altered to start processes properly within this framework (including passing the started programs arguments via `argc` and `argv`). That is, it should be possible to launch a process with arguments either from the kernel menu via `runprogram()` or via an `execv()` system call from an application program.

`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, so this is a good file to read very carefully. 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 are the ELF magic numbers?

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

Question 3. Why can the struct `uio` that is used to read in a segment be allocated on the stack in `load_segment()` (i.e., where does the memory read actually go)?

Question 4. In `runprogram()`, why is it important to call `vfs_close()` before going to user mode?

Question 5. What function forces the processor to switch into user mode? Is this function machine dependent?

Question 6. In what file are `copyin` and `copyout` defined? `memmove`? Why can't `copyin` and `copyout` be implemented as simply as `memmove`?

Question 7. What (briefly) is the purpose of `userptr_t`?

In `kern/arch/mips/mips`: traps and syscalls

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. `md_usermode()` is the key function for returning control to user programs. `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. Notice that `reboot()` is the only case currently handled. You will also find a function, `md_forkentry()`, which is a stub where you will place your code to implement the `fork()` system call. It should get called from `mips_syscall()`.

Question 8. What is the numerical value of the exception code for a MIPS system call?

Question 9 Why does `mips_trap()` set `curspl` to `SPL_HIGH` "manually", instead of using `splhigh()`?

Question 10 How many bytes is an instruction in MIPS? (Answer this by reading `mips_syscall()` carefully, not by looking somewhere else and explain how you arrived at the answer.

Question 11 Why do you "probably want to change" the implementation of `kill_curthread()`?

Question 12 What would be required to implement a system call that took more than 4 arguments?

`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.

`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, although it may be instructive in the long run to do so. For present purposes you need only look at the code that implements the user-level side of system calls.

`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.

Question 13 What is the purpose of the `SYSCALL` macro?

Question 14 What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not by looking somewhere else.) Explain where you found and how your arrived at the answer.

3 Implementation

For this assignment, you are expected to implement exception handling and the following system calls:

- `open`, `read`, `write`, `lseek`, `close`

- `getpid`
- `fork`, `execv`, `waitpid`, `_exit`

Integer codes for these and other system calls are listed in `kern/include/kern/callno.h`.

It's crucial that your syscalls handle all error conditions gracefully (i.e., without crashing OS/161.) You should consult the OS/161 man pages included in the distribution and understand fully the system calls that you must implement. You must return the error codes as described in the man pages.

Additionally, your syscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the man pages.

The file `include/unistd.h` contains the user-level interface definitions of OS/161 system calls. This interface is different from that of the kernel functions that you will define to implement these calls. You need to design this interface and put it in `kern/include/syscall.h`. You need to think about a variety of issues associated with implementing the system calls required for this assignment.

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/errno.h`. If none seem particularly appropriate, consider adding a new one. (Do not, however, redefine or delete any of the existing error codes.) 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/lib/libc/strerror.c`.

3.1 Exception Handling

Your operating system should properly deal with programs that generate exceptions.

3.2 `open`, `read`, `write`, `lseek`, `close`

For any given process, the first three file descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr). These file descriptors should start out attached to the console device ("con:").

These system calls are largely about manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the current working directory) is specific only to the process, but other information (such as file offset) is specific to a particular file descriptor within a process. Think carefully about the state you need to maintain, how to organize it, and when and how it has to change.

3.3 `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 of the pids. Design your pid system; implement all the tasks associated with pid maintenance, and only then implement `getpid()`.

3.4 `fork`, `execv`, `waitpid`, `_exit`

These system calls are probably the most difficult part of the assignment. 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.

`execv()` is responsible for taking newly created processes and make them execute something useful (i.e., something different from 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 – you must modify it and calls to it so that it correctly handles arguments.

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. You may also wish to consult the UNIX man page, though keep in mind that you are not required to implement all the things UNIX `waitpid()` supports – nor is the UNIX parent/child model of waiting the only valid or viable possibility.

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.

3.5 `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.

4 Design Document

You are expected to prepare a short design document describing your implementation of these system calls. Your document should describe the major issues that you encountered in designing and implementing the mechanisms that support the required system calls in your kernel, and how you resolved those issues in your design. Be sure to justify your design decisions. Please prepare your design document as a PDF file called `design.pdf`. Your design document should be *at most* three pages long using 1 inch margins and a minimum font size of 12 points.

5 Testing

Test programs will largely be used to determine whether your implementation for various system calls is correct. They will be used to help guide the markers in determining what to look for in your source code.

5.1 OS/161 Test Programs

A fair number of relatively simple test programs are included with OS/161. The source for these programs can be found in `os161-1.11/testbin`. After they are compiled and installed they will be found in `cs350-os161/root/testbin` and they can be invoked from the shell prompt as:

```
% cd cs350-os161/root
% sys161 kernel "p testbin/add 2 4;q"
Answer: 6
```

The marking guideline will list several tests that you should try to run. It will also provide an idea of how many marks you could expect if you have a correct implementation that is able to run the specified programs properly.

5.2 Your Test Programs

Where the tests already provided with OS/161 are insufficient you should design and implement some of your own tests to demonstrate that your implementations of the system calls you have added work correctly. Be sure that these tests are sufficiently different from the test programs that are included with OS/161 that their inclusion is useful and/or necessary. You should create a document of **no more than 3 pages** called `testing.pdf` in which you describe your test programs and submit it with the rest of your assignment.

Be sure to include tests that show how you protect the operating system against malicious programmers who might try to crash your kernel by passing invalid arguments to the system calls. Place all of your test programs in a new directory you will create named `os161-1.11/ourtests`. You should be able to start by mimicking the structure of the `os161-1.11/testbin` directory, including its `Makefile`. Then start by using one of the programs from the `testbin` directory as an example of how to create, compile and install a new test program. You should ensure that your tests are installed in the directory `ourtests` by setting the BINDIR variable in the Makefile to `/ourtests`. For example the Makefile for a new program called exittest1 would look like:

```
# Makefile for exittest1.
SRCS = exittest1.c
PROG = exittest1
BINDIR = /ourtests

include ../../defs.mk
include ../../mk/prog.mk
```

Once this program is compiled and installed you should be able to run it as follows (assuming it does not require any arguments):

```
% cd cs350-os161/root
% sys161 kernel "p ourtests/exittest1;q"
```

6 What to Submit

You should submit the following items using the `submit` command:

- A copy of your OS/161 source code incorporation all of your implementation work, including any testing programs that you many have developed. Please submit the top of the OS/161 source tree (e.g., `$HOME/cs350-os161/os161-1.11`) and everything below it.
- Your file `codeanswers.pdf`, described in Section 2 and placed into the os161-1.11 directory.
- Your file `design.pdf`, described in Section 3 and placed into the os161-1.11 directory.
- Your file `testing.pdf`, described in Section 5 and placed into the os161-1.11 directory.

When using the `submit` command to submit this assignment, please use assignment number 2, like this:

```
% cd $HOME/cs350-os161/os161-1.11
% submit cs350 2 .
```