Assignment 3

System Calls & Processes

Wiki Components Due: 1:00 PM, Thursday, February 19, 2004
Final Patch Due: 11:00 PM, Friday, February 27, 2004

In this assignment you will be implementing a set of file- and process-related system calls. When you have completed the required portion of the assignment, your operating system will be able to run multiple copies of a single user-space program, and allow that program to perform basic file I/O. You will make OS/161able to run *real programs*! If you successfully complete the (optional) advanced part of this assignment, you will also add the ability to run a variety of programs concurrently.

A substantial part of this assignment is understanding how OS/161 works and determining what code is required to implement the required functionality. Expect to spend at least as long browsing and digesting OS/161 code as actually writing and debugging your own code.

Preliminaries

Recall that you *must* have your path correctly set to undertake all CS 182 assignments. If you haven't done so already, put the appropriate line in your `.login` or `.bashrc`.

You no longer need your code from Assignment 2. If you wish to save what you have done, you could run something along the lines of

```
cd ~/cs182
mkdir assign2-changes
mv assign2/handin assign2-changes/
mv assign2/src/*/{thread,synch}.{c,h} assign2-changes/
```

before removing the `assign2` directory.

For this assignment, check out and `setup` your `assign3` directory in the usual way, then configure and build the kernel using the `HW3` configuration file (see previous assignments for details).

Recall that the `setup` script builds all of the "userland" programs and libraries (e.g., `/sbin/reboot`). This assignment includes an additional program, `/testbin/hw3test`. After you've run `setup` and built the kernel, running this program (using the `p` command from the menu) will yield the following results:

```
Unknown syscall 6
Unknown syscall 6
Unknown syscall 6
        ⋮
Unknown syscall 0
```

When your assignment is complete, the results will be much more satisfying.

Overview

In the previous assignment, you may have thought of your solutions to the cats and mice problem as separate programs, but they were really just functions that were linked into the kernel itself and thus ran *inside* the kernel, in kernel mode. We used that approach because the kernel wasn't yet able to run meaningful user-space code. In this assignment, you'll be making OS/161 deal with true user-space programs for the first time.

At present, OS/161's support for user-space programs is limited and incomplete. The only working system call available to user-space code is reboot. Moreover, the kernel itself doesn't currently understand what a process is or maintain any per-process state. In this assignment, you will remedy this issue by

- Tracking "per-process" information
- Providing system calls that support program I/O (e.g., read, write)
- Providing system calls for managing processes (e.g., fork)

The first step is to read and understand the existing parts of OS/161. At present, OS/161 can run one user-level C program at a time—of course, the only programs that work are `reboot`, `halt`, and `poweroff`, which all (only) use the reboot system call. Understanding how the kernel runs these simple programs should help you in planning what parts of the code you will need to modify.

Code Reading

The code-reading component of this assignment is meant to direct your attention to those parts of OS/161 that are particularly relevant to this assignment.

`kern/userprog/`
: Contains the files that are responsible for the loading and running of user-level programs. Currently, this directory only contains three source files (described below), 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.

`kern/userprog/loadelf.c`
: Contains the functions responsible for loading an ELF executable from the filesystem and into virtual memory space. (Of course, at this point OS/161's dumbvm code does not provide what is normally meant by virtual memory—although there is translation between the addresses that executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically.)

`kern/userprog/runprogram.c`

Contains only one function, runprogram, which is responsible for running a 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 more is required for execv that runprogram does not need to worry about. Additionally, once you have designed your process system, runprogram should be altered to start processes properly within this framework; for example, a program started by runprogram should have the standard file descriptors available while it's running.

`kern/userprog/uio.c`

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 be sure to read this file very carefully. You should also examine the code in `lib/copyinout.c`.

Traps & System Calls

`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. You can think of exceptions as the interface between the processor and the operating system. 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 code, which sets up a "trap frame" and calls into the operating system (a "trap" is a machine exception that the operating system catches and handles). Interrupts are exceptions, and, more significantly for this assignment, so are system calls. Specifically, `syscall.c` handles traps that happen to be syscalls. Understanding the C code in this directory is key to being a real operating-systems junkie.

`kern/arch/mips/mips/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 user mode 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.

`kern/arch/mips/mips/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.

Userland

`lib/crt0/`
: There's only one file in here, `mips-crt0.S`, which contains the MIPS assembly code that receives control when a user-level program is started. It calls main. Your execv implementation will be interface with this code, so be sure to check what values it expects to appear in what registers and so forth.

`lib/libc/`
: There's obviously a lot of code in the OS/161 C library (but dramatically less than, say, the GNU C library on a Linux system). 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.

`lib/libc/errno.c`
: Defines the global variable errno.

`lib/libc/syscalls-mips.S`
: Contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls.

`lib/libc/syscalls.S`
: This file is created from `syscalls-mips.S` at compile time and is the actual file assembled to put into the C library. The actual names of the system calls are placed in this file using a script, `callno-parse.sh`, that reads them from the kernel's header files. This approach avoids having to make a second list of the system calls.

Wiki Component

This component is not graded in the conventional sense—completion of this part of the assignment is covered under the course's *participation* requirement. Every member of the class should should post the answer to one of the questions below on the course's Wiki and understand the answers to *all* of the questions. Some questions are easier than others, so if you have answered an easy one, feel free to answer another. You may also correct or amplify someone else's answer. If all of the questions have been adequately answered, you should think of a closely related question and both ask and answer it.

In class, I may ask you about the answers to these questions, *or* ask you closely related questions whose answers you should know from answering the questions below.

W1. What is the difference between UIO_USERISPACE and UIO_USERSPACE?
When should one use UIO_SYSSPACE instead?

W2. 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?)

W3. In runprogram, why is it important to call vfs_close before going to user mode?

W4. What function forces the processor to switch into user mode?
Is this function machine dependent?

W5. In what file are copyin and copyout defined? What about memmove?
Why can't copyin and copyout be implemented as simply as memmove?

W6. What (briefly) is the purpose of *userptr_t*?

W7. What is the numerical value of the exception code for a MIPS system call?

W8. Why does mips_trap set curspl to SPL_HIGH "manually", instead of using splhigh?

W9. How many bytes is an instruction in MIPS?
(Answer by reading mips_syscall carefully, not by looking somewhere else.)

W10. What is stored in **struct** *trapframe*?
Where is the *trapframe* that is passed into mips_syscall stored?

W11. What would be required to implement a system call that took more than four arguments?

W12. What is the purpose of userptr_t?

W13. What is the purpose of the SYSCALL macro?

W14. What is the MIPS instruction that actually triggers a system call?
(Answer this by reading the source in this directory, not looking elsewhere.)

W15. How are vfs_open, vfs_close used? What other vfs_... calls are relevant?

W16. What are VOP_READ and VOP_WRITE? How are they used?
What does VOP_TRYSEEK do?

W17. Where is the **struct** *thread* defined? What does this structure contain?

Design and Implementation Requirements

The first week of this assignment is devoted to *design*, the second to coding. Your first goal is to work out *how* you are going to achieve the objectives of this assignment, so that both members of the team will have a clear idea of what it is you are trying to achieve when you begin coding. At the end of the design phase, you will post a design document on the Wiki site. After the design documents have been posted, you may use your own, *or* steal someone else's to use in implementing your design.

Basic File and Process System Calls

For the required part of this assignment, you will need to design and implement support for the following system calls:

C1. open, read, write, lseek, close, dup2

C2. fork, _exit

Your implementation of these system calls must

- Use the system call numbers defined in `kern/include/kern/callno.h`
- Conform to the user-level API given in `include/unistd.h`
- Reflect the (POSIX-style) semantics given in the OS/161 documentation, especially with regard to error codes and return values
- *Never* crash the OS/161 kernel

The man pages in the OS/161 distribution (available on-line at http//www.cs.hmc.edu/cs182/os161/) contain a description of correct return values for various errors. 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. 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. Consult Unix man pages to learn about Unix error codes (see the `intro` page in Section 2 of the manual, use `man man` to find out how to get to Section 2). Note that if you add an error code to `kern/include/kern/errno.h`, you need to add a corresponding error message to the file `src/lib/libc/strerror.c`.

I/O-Related Calls

The calls in part C1 all manipulate file descriptors. From a user-code perspective, file descriptors are small integers returned by the operating system that signify open files. On a POSIX system, the convention is that for any given process, the first file

descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr). In your solution, these file descriptors should start out attached to the console device ("con:"), but your implementation must allow programs to use dup2 to change them to point elsewhere.

Although these system calls may seem to be tied to the filesystem, in fact, these system calls are really 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. You must determine what information is specific only to the process, what is specific to the file descriptor, and how the two relate. Don't rush your design. Think carefully about the state you need to maintain, how to organise it, and when and how it has to change.

In implementing these system calls, you will not actually have to write any low-level file system code in this assignment. You will use the existing VFS layer to do most of the work. Your job is to construct the subsystem that implements the interface expected by user-level programs by invoking the appropriate VFS and vnode operations.

For consistency, please place most of your implementation of file-related system calls in the following files:

`kern/include/file.h`
: Function prototypes and data types for your file subsystem

`kern/userprog/file.c`
: Function implementations and variable instantiations

Process-Related Calls

In this part of the assignment, you should implement a simplified form of the fork system call. Your implementation should be the same as that described in the fork man page, except that it should return 1 to the parent (rather than the child's PID).

The amount of code to implement fork and _exit is quite small at this stage; the main challenge is to understand what needs to be done and where. In particular, you should design and implement the file-related system calls with fork in mind.

Some hints:

- Read the comments above mips_usermode in `kern/arch/mips/mips/trap.c`
- Read the comments in `kern/include/addrspace.h`, particularly as_copy
- You will need to copy the trap frame from the parent to the child—be careful how you do this, as there is a possible race condition
- You may wish to base your implementation on the thread_fork function in `kern/thread/thread.c`

Robustness

Currently, if user-space code generates a fatal exception, the kernel panics. This placeholder solution is obviously not acceptable, and needs to be fixed now that you'll be running actual programs in user space. You'll need to replace kill_curthread with a new version that properly handles such issues more sanely. You should, of course, try to write it in as simple a manner as possible. Keep in mind that essentially nothing about the current thread's user-space 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.

Testing

You can run `/testbin/hw3test` to test your system calls. Example output (where most of the system calls except _exit are implemented) is shown below:

```
OS/161 kernel [? for menu]: p /testbin/hw3test
Operation took 0.000212160 seconds
OS/161 kernel [? for menu]:
**********
* File Tester
**********
* write() works for stdout
**********
* write() works for stderr
**********
* opening new file "test.file"
* open() got fd 3
* writing test string
* wrote 45 bytes
* writing test string again
* wrote 45 bytes
* closing file
**********
* opening old file "test.file"
* open() got fd 3
* reading entire file into buffer
* attemping read of 500 bytes
* read 90 bytes
* attemping read of 410 bytes
* read 0 bytes
* reading complete
* file content okay
**********
* testing lseek
```

```
* reading 10 bytes of file into buffer
* attemping read of 10 bytes
* read 10 bytes
* reading complete
* file lseek  okay
* closing file
**********
* testing fork
* Forked, in parent
* Forked, in child
Unknown syscall 0
```

Advanced Process-Related System Calls

This part of the assignment is *optional*. If you choose not to undertake it, your maximum possible grade will be a B+. For this part of the assignment, you will implement the following system calls:

C3. getpid

C4. fork, execv, waitpid, and _exit

Implementing Process IDs

A pid, or process ID, is a unique number that identifies a process (as such, it has much in common with a file descriptor; both are integers that let user-space programs refer to kernel data safely). The implementation of getpid is not terribly challenging, but pid allocation and reclamation are the important concepts that you must implement. It is not okay for your system to crash because over the lifetime of its execution you've used every possible pid once. Design your pid system, implement all the tasks associated with pid maintenance, and only then implement getpid.

Process Creation, Execution and Termination

In this part, you must revise fork so that a newly created process is given a new PID and that pid is returned to the *parent* of the process. 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.

Although execv is "only" a system call, it is perhaps the most challenging part of this assignment. It is responsible for taking a process and making it execute a new program. 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 user

space, into the kernel, and then returning back to user space. You must manage the memory that travels across these boundaries very carefully. (Also, notice that runprogram doesn't take an argument vector—but these must of course be handled correctly in execv).

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, although you should keep in mind that you are not required to implement all the things that Unix's waitpid supports—nor is the Unix parent/child model of waiting the only valid or viable possibility.

Your revised implementation of _exit is also intimately connected to the implementation of waitpid. They are essentially two halves of the same mechanism. In all likelihood, your code for _exit will be simple and your code for waitpid more complex—but it's perfectly viable to design it the other way around. If you find both are becoming extremely complicated, it may be a sign that you should rethink your design.

Shell

The system calls you have in OS/161 do not allow as full-featured a shell, but are nevertheless sufficient to write a simple and useful command-line interface. This part of the assignment is recommended if you undertake the advanced part, but is not assessed. You may adapt code you wrote for 105, or collaborate more widely than just your partner. Skeleton code for your shell is in `bin/sh/sh.c`.

Remember that because OS/161 implements a subset of the standard POSIX API, you can develop your shell under BSD, Linux, MacOS X, or Solaris and then compile it for OS/161. But keep in mind that OS/161 is *really* basic right now. No signals. No malloc. And no, you're not allowed to write those missing parts to make your shell better/easer. Yet.

Wiki Posting of Design Documents

Your design documents should include

- A detailed description of all new data structures (diagrams may help and are allowed[1])
- A detailed description of all changes to existing data structures
- Pseudocode for all non-trivial required system calls (e.g., open, close, fork, read, fork, exec, waitpid, and _exit)

1. GIF and PNG images are easiest to attach to your Wiki page and view, but feel free to attach printable EPS, PDF, or PS files as alternatives to your low-res raster images.

The exact method for posting your design on the Wiki site will be discussed in class on Tuesday—do not post until *after* Tuesday's class.

Creating a Patch

The final form of your solution will be a kernel patch, generated with

```
cd ~/cs182/assign3
cs182patch src/kern > handin/syscalls.patch
```

Once you have created the basic patch file, you should edit it and add any additional comments. You *may* assume that the person reading your patch can read your Wiki design document.

Hints & Tips

You need to think about a variety of issues associated with implementing system calls. In this section, we raise some questions and offer a few tips.

- Can two different user-level processes find themselves running a system call at the same time? (Whichever conclusion you reach, your conclusions in matters such as this belong in your design document.)
- If you undertake the advanced part of this assignment, your system must allow user programs to receive arguments from the command line. For example, your shell should be capable of executing lines (in user programs) such as:

  ```
  char *filename = "/bin/cp";
  char *args[4];
  pid_t pid;
  args[0] = "cp";
  args[1] = "file1";
  args[2] = "file2";
  args[3] = NULL;
  pid = fork();
  if (pid == 0)
        execv(filename, argv);
  ```

  which will load the executable file cp from, install it as a new process, and execute it. The new process will then find `file1` on the disk and copy it to `file2`.

Some questions to think about:

1. Passing arguments from one user program, through the kernel, and into another user program, is a bit of a chore. It is also rather tricky, and there are many ways to be led astray. You will probably find that very detailed pictures and several walk-throughs will be most helpful.
2. What primitive operations exist to support the transfer of data to and from kernel space? Do you want to implement more on top of these?
3. How will you determine
   (a) The stack pointer's initial value
   (b) The initial register contents
   (c) The return value
   (d) Whether you can exec the program at all?

- You will need to bullet-proof the OS/161 kernel from user-program errors. There should be nothing that a user program can do to crash the operating system (with the exception of explicitly asking the system to halt).
- What new data structures will you need to manage multiple processes?
- What relationships do these new structures have with the rest of the system?
- How will you manage file accesses? When your shell invokes the `cat` command, and the `cat` command starts to read from file descriptor 0, what will happen if the shell also tries to read from the same file descriptor? What would you like to happen?

Finally, remember to *keep it simple*. Both `open` and `fork` are allowed to return errors saying that you've opened too many files at once or forked too many processes at once, which, in turn, means that you are allowed to use fixed-sized tables.