Instructor: Ding Yuan
Course Number: ECE344

Operating Systems

ECE344, WINTER 2014
UNIVERSITY OF TORONTO

ASSIGNMENT 2: SYSTEM CALLS AND PROCESSES

Release date: Feb 21.
Due date: March 14th, 11:59 pm.
TA responsible for this Lab: Afshar Ganjali (a.ganjali at utoronto dot ca) and Xu Zhao (nuklly at gmail dot com).

Objectives

After this assignment, you should:

- Be able to write code that meets a specified interface definition.
- Understand how to represent processes in an operating system.
- Design and implement data structures to manage processes in an OS.
- Understand how to implement system calls.
- Understand how to use synchronization in OS code.

Introduction

Your current OS161 system has minimal support for running executables -- nothing that could be considered a true process. This assignment starts the transformation of OS161 into a true multi-tasking operating system. After this assignment, your OS will be capable of concurrently running multiple processes from actual compiled programs stored in your distribution. These programs will be loaded into OS161 and executed in user mode by System/161.

This assignment will require the synchronization primitives you have developed in the previous assignment. Make sure they are bullet proof!

The main purpose of this assignment is to implement user-level processes. So far, almost all the code you have written for OS161 has only been run within, and only been used by, the operating system kernel. For example, the synchronization problems in the previous assignment were implemented as threads running in the kernel. In a real operating system, the kernel's main function is to provide support for user-level programs. Most such support is accessed via system calls. We have provided you with the `reboot` system call. In this assignment, you will be implementing various process-related system calls.

The first step is to read and understand the parts of the system that we have written for you. Our code can run one user-level C program at a time from the kernel. Your job will be to allow a user-level program to issue the process-related system calls to create other user-level processes.

As part of this assignment, you will need to implement the subsystem that keeps track of multiple threads and user-level processes. You must decide what data structures you will need to hold the data pertinent to a thread or a process.

Our System/161 simulator can run normal programs compiled from C. The programs are compiled with a cross-compiler, `cs161-gcc`. This compiler runs on the host machine and produces MIPS executables; it is the same compiler used to compile the OS161 kernel. We have provided various programs that you can use to test features as you add them in this and future assignments. Most these programs are available in `/testbin`. Some programs are also available in `bin` and `sbin`. To create new user programs, you will need to edit the Makefile in one of these directories (depending on where you put your programs) and then create a directory similar to those that already exist. Use an existing program and its Makefile as a template.

Design and communication

You should first come up with a clear design of your solution before you start coding. While it is not required, we encourage you to write a design document for this assignment to clearly reflect the design of your solution. If you try to code first and design later, or even if you design hastily and rush into coding, you will most certainly end up in a software "tar pit". Don't do it! Work with

your partner to plan everything you will do. Don't even think about coding until you can precisely explain to each other what problems you need to solve and how the pieces relate to each other.

Since you are working with a partner, nothing replaces good, open communication between partners. The more you can direct that communication to issues of content ("How shall we design sys_execv()?") instead of procedural details ("What do you mean, you never checked in your version of foo.c?"), the more productive your group will be.

If at any time during the course of the term, you and your partner realize that you are having difficulty working together, please speak with your professor or one of the teaching assistants. We will work with you to help your partnership work more effectively. Do not suffer in silence.

Using SVN effectively

Since you and your partner will be using SVN to manage your work, you will need to decide when and how often to commit changes. (Advice: early and often.) Additionally, you should agree upon how much detail to log when committing files. Perhaps more importantly, you also need to think about how to maintain the integrity of the system, i.e., what procedures to follow to make sure you can always extract some working version from SVN, whether or not it's the latest version, what tests to run on a new version to make sure you haven't inadvertently broken something, etc.

Clear, explicit SVN logs are essential. If you are incommunicado for some reason, it is vital for your partner to be able to reconstruct your design, implementation and debugging process from your SVN logs. In general, leaving something uncommitted for a long period of time is dangerous: it means that you are undertaking too large a piece of work and have not broken it down effectively. Once some new code compiles and doesn't break the world, commit it. When you've got a small part working, commit it. When you've designed something and written a lot of comments, commit it. Commits are free. Hours spent hand-merging hundreds of lines of code wastes time you'll never get back. The combination of frequent commits and good, detailed comments will facilitate more efficient program development.

Use the features of SVN to help you. For example, you may want to use SVN tags to identify when major features are added and when they're stable.

Naming conventions

It's a good idea to select some rudimentary protocol for naming global variables and variables passed as arguments to functions. This way, you can just ask your partner to write some function and, while s/he's doing it, you can make calls to that function in your own code, confident that you have a common naming convention from which to work. Be consistent in the way you write the names of functions: given a function called "my function", one might write its name as my_function, myFunction, MyFunction, mYfUnCtIoN, ymayUnctionFay, etc. Pick one model and stick to it (although we discourage the last two examples).

Begin your assignment

The very first thing you need to make sure is that you do not have any outstanding updates in your Subversion tree. Use `svn update` and `svn commit` to get your tree committed. Make sure that you do not commit generated files or there will be a penalty.

Now, tag your repository exactly as shown below. The `$SVN344_SVN` variable setup is described here.

```
% svn copy  -m "starting assignment 2" $ECE344_SVN/trunk $ECE344_SVN/tags/asst2-start
```

Next configure and build OS161 for this assignment. The configuration and build instructions, and the command line arguments to OS161 are similar to the previous assignment. Replace ASST1 with ASST2 for this assignment.

Code walk-through

To help you get started, we have provided the following questions as a guide for reading through the code. After reading the code and answering questions, get together with your partner, and you should be ready to discuss strategy for designing your code for this assignment.

kern/userprog: running user process

This directory contains the files that are responsible for the loading and running of 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 and into virtual memory space. (ELF is the name of the executable format produced by `cs161-gcc`.) Of course, at this point this virtual memory space 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. We will rectify that in Assignment 3.

`runprogram.c`: This file 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 -- when writing your design doc, you should determine what more 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; for example, a program started by `runprogram()` should be passed the correct arguments.

`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 userlevel programs, so this is a good file to read very carefully. You should also examine the code in `lib/copyinout.c`.

Questions

1. What are the ELF magic numbers?
2. What is the difference between UIO_USERISPACE and UIO_USERSPACE? When should one use UIO_SYSSPACE instead?
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)?
4. In `runprogram()`, why is it important to call `vfs_close()` before going to usermode?
5. What function forces the processor to switch into usermode? Is this function machine dependent?
6. In what file are `copyin` and `copyout` defined? Why can't `copyin` and `copyout` be implemented simply as `memmove`?
7. What is the purpose of `userptr_t`?

kern/arch/mips/mips: traps and syscalls

Exceptions/interrupts are the key to operating systems; they are the mechanisms that enable 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, which sets up a "trap frame" and calls into the operating system. Since "exception" is such an overloaded term, operating system lingo for an exception is a "trap" -- when the OS traps execution. Interrupts are exceptions, and more significantly for this assignment, so are system calls. Specifically, `syscall.c` handles traps that happen to be syscalls. Understanding at least the C code in this directory is key to being a real operating systems junkie, so we highly recommend reading through it carefully.

`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. You will also find a function, `md_forkentry()`, which is a stub where you will place your code to implement the `fork()` system call. The `fork` system call should get called from `mips_syscall()`.

Questions

1. What is the numerical value of the exception code for a MIPS system call?
2. Why does `mips_trap()` set `curspl` to `SPL_HIGH` "manually", instead of using `splhigh()`?
3. How many bytes are in an instruction in MIPS? (Answer this by reading `mips_syscall()` carefully, not by looking somewhere else.)
4. Why do you "probably want to change" the implementation of `kill_curthread()`?

4. Why do you probably want to change the implementation of `kill_curthread()`?
5. What would be required to implement a system call that took more than 4 arguments?

lib: user-level library code

`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 what values it expects to appear in what registers 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. Job interviewers have an uncanny habit of asking people to implement standard C library functions on the whiteboard. For present purposes you need only look at the code that implements the user-level side of system calls, which we detail below.

`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. OS161 puts them all together to simplify the makefiles.

Questions

1. What is the purpose of the `SYSCALL` macro?
2. What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

Design and implementation

The full range of system calls that we think you might want over the course of the semester is listed in `kern/include/kern/callno.h`. For this assignment, you need to design and implement the following system calls:

- fork, getpid, waitpid, _exit
- execv

The first set of system calls (e.g., `fork`) allow running multiple user-level processes in OS161. However, all the processes run the same program. The last system call (`execv`) will allow running different user-level programs, making OS161 a much more useful entity.

It's *crucial* that your syscalls handle all error conditions gracefully (i.e., without crashing OS161.) You should consult the OS161 man pages included in the distribution (`man/syscall`) and understand fully the system calls that you must implement. You must return the error codes as decribed 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. Adherence to the guidelines is as important as the correctness of the implementation.

The file `include/unistd.h` contains the user-level interface definition of the system calls that you will be writing for OS161 (including ones you will implement in later assignments). 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`. As you discovered (ideally) in Assignment 0, the integer codes for the calls are defined in `kern/include/kern/callno.h`.

printing system call

When you run some of the test programs, you will notice that `printf()` and other console-printing libc calls do not work. You will need to implement a system call to make `printf()` and other console-printing calls work. You should first figure out which system call is eventually used to perform the printing, and implement this system call. The implementation of this system call should be straightforward.

read system call

To get some of the testing programs to work, you will also need to implement a read system call that reads one character at a time. The implementation of this system call should be straightforward.

fork()

`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()` to make sure that other system calls are performing the correct functionality.

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 the pids. Design your pid system; implement all the tasks, including locking shared data structures, associated with pid maintenance, and only then implement `getpid()`.

waitpid()

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. The pid maintenance needed for `getpid()` will be helpful for `waitpid()`.

_exit()

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.

execve()

`execv()`, although "only" a system call, is really the heart of this assignment. It is responsible for taking newly created processes and making them execute a different program (i.e., a program different than 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. Notice that `runprogram()` currently doesn't take an argument vector. This must be handled correctly in `execv()`. You will also have to consider the interation of `fork()` with `execv()`. Once you have `execv()` working, modify `runprogram()` so that it can take arguments. This will allow you to run programs that take arguments from the menu.

Some design considerations

What new data structures will you need to manage multiple processes? Hint: In this and future assignments, you will need to manage several lists of different types of items. It is a good idea to implement a generic linked list and hash table for managing these different items.

After you have implemented the `execv()` system call, your system must allow user programs to receive arguments via the command line and via the `runprogram()` function. For example, you should be able to run the following "test" program:

```
char  *filename = "/testbin/add";
char  *args[4];
pid_t  pid;

args[0] = "add";
args[1] = "5";
```

```
args[2] = "12";
args[3] = NULL;

pid = fork();
if (pid == 0) execv(filename, argv);
```

This test program will load the executable file `add`, install it as a new process, and execute it. The new process will then add the two numbers passed and print the result. Furthermore, if the test program above takes arguments and `runprogram` is used to invoke it, then `runprogram` should pass parameters to this program correctly.

Passing arguments from one user program, through the kernel, into another user program, is a bit of a chore. What form does this take in C? This is 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. 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?

How will you determine: (a) the stack pointer initial value; (b) the initial register contents; (c) the return value; (d) whether you can exec the program at all?

For implementing `waitpid()` and `_exit()`, you may consider adding two more thread subsystem

calls, `thread_join` and `thread_detach` in the kernel. `thread_join` suspends execution of the calling thread until a specified thread exits unless a `thread_detach` has been performed on the specified thread so that it is no longer joinable. These calls may interact with `thread_fork()` also.

In your design document, describe your solution to the `waitpid()/exit()` synchronization problem by describing each case that you have handled.

The OS161 menu thread forks new kernel threads to handle the commands given at the prompt. Currently, the menu thread sometimes prints the prompt before the thread it has spawned finishes executing, which can make it look like the system is not working correctly. Use your thread synchronization implementation above to make the menu thread wait until the child thread has finished before printing the prompt and accepting another command.

The man pages in the OS161 distribution contain a description of the error return values that you must return. Make sure that you handle error conditions correctly. 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`. Note that if you choose to 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`.

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.

Testing your code

After you have finished implementing all the system calls, other than `execv`, you should be able to run the following programs in the `/testbin` directory: `forktest,tictac, crash, faulter, forkbomb`. Recall that you can run these programs by using the `p` menu option, e.g., `p /testbin/forktest`, or `cd /testbin`, `p forktest`.

After you finish implementing `execv`, you will be able to run additional programs in the `/testbin` directory. These include the `add, argtest, badcall?, farm?, randcall?, sty` programs. The programs shown with a "?" may work partially. Some programs in the `/bin` and the `/sbin` directory should also work.

Feel free to run the various test programs using the OS menu or you can write a simple shell program in `bin/sh` using the system calls you have implemented to run these programs.

You will need to "bullet-proof" the OS161 kernel from user program errors. There should be nothing a user program can do to crash the operating system (with the exception of explicitly asking the system to halt). In particular, the `forkbomb` program should only report that your kernel has run out of memory but should not cause any other problem.

Preparing your assignment for submission

Finally, you need to prepare your code for submitting your assignment.

1. Once you are confident that you have completely done your assignment, run `make clean` from the `os161` directory. This will clean all generated files. Then use `svn status` in the `os161` directory to find out the status of all files. Any files that have a `?` before them are not in the repository. If you have created these files by hand, then add them to the repository

using svn add. If these are generated files, use the instructions here to ignore these files.

2. Then run `svn commit` from the `os161` directory so that all modified files are checked in your repository. Use `svn status` in the `os161` directory again to make sure that all your modified source files are properly committed. Make sure that your partner's changes are also committed.
3. Tag your repository for the end of asst2. We will be using the start and the end tags to checkout your code for marking:

```
% svn copy -m "ending assignment 2" $ECE344_SVN/trunk $ECE344_SVN/tags/asst2-end
```

Testing your assignment with os161-tester

Please read the instructions for testing your code.