Advanced Operating Systems

Computer Science Department
Indiana University
Bloomington, IN 47405

The Tao

A manager went to his programmers and told them: "As regards to your work hours: you are going to have to come in at nine in the morning and leave at five in the afternoon." At this, all of them became angry and several resigned on the spot.

So the manager said: "All right, in that case you may set your own working hours, as long as you finish your projects on schedule." The programmers, now satisfied, began to come in at noon and work to the wee hours of the morning.

— Geoffrey James, *The Tao of Programming*

Programming Assignment 2: System Calls, Threads, and Processes

Assigned	Sep 12, 2012
Design Due	Sep 22, 2012
Due	Oct 3, 2012
Note	This problem set is to be done in groups of two.

Introduction

In the last assignment you started working in teams of two. In general, you may find it difficult to work in a team if you are accustomed to working alone, but it is essential for the completion of the remaining assignments and is a worthwhile skill to develop in any case. These assignments are too complex to be done single-handedly, and you will gain valuable real-world experience from learning to work in a team effectively.

Working in Teams

There are a number of issues that you and your partner should work out now, when things are calm, so you needn't figure them out at 2:00 AM in the heat of the moment.

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).

Git Use

Since you and your partner will be using Git 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- what procedures to follow to make sure you can always extract a working version of some sort from Git, 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 Git 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 Git 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 Git to help you. For example, you may want to use tags to identify when major features are added and when they're stable. The brave of heart might want to investigate Git branches, which provide completely separate threads of development (one significant caveat-although branches make life much easier while you're developing within a branch, merging branches together later is often a major headache).

Communication

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.

In your design documents for each assignment, you should identify, at least in general terms, who was responsible for the various parts of your solutions.

If at any time during the course of the semester, you and your partner realize that you are having difficulty working together, please come speak with Professor Lumsdaine or one of the AIs. We will work with you to help your partnership work more effectively, or in extreme circumstances, we will help you find new partners. Do not suffer in silence; please come talk with us.

Assignment Organization

In this assignment you will implement system calls, processes, and synchronization primitives for OS/161 and learn how to use them to solve several synchronization problems. Once you have completed the written and programming exercises you should have a fairly solid grasp of the pitfalls of concurrent programming and, more importantly, how to avoid those pitfalls in the code you will write later this semester.

To complete this assignment you will need to be familiar with the OS/161 thread code. The thread system provides interrupts, control functions, and semaphores. You will implement locks and condition variables.

Your current OS/161 system has minimal support for running executables -- nothing that could be considered a true process. Assignment 2 starts the transformation of OS/161 into a true multi-tasking operating system. After the next assignment, it will be capable of running multiple processes at once from actual compiled programs stored in your account. These programs will be loaded into OS/161 and executed in user mode by System/161; this will occur under the control of your kernel and the command shell in `bin/sh`.

First, however, you must implement the interface between user-mode programs ("userland") and the kernel. As usual, we provide part of the code you will need. Your job is to design and build the missing pieces.

You will also be implementing the subsystem that keeps track of the multiple tasks you will have in the future. You must decide what data structures you will need to hold the data pertinent to a "process" (hint: look at kernel include files of your favorite operating system for suggestions, specifically the `proc` structure.)

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 as long as it doesn't want to do anything but shut the system down. We have provided sample user programs that do this (reboot, halt, poweroff), as well as others that make use of features you will be adding in this and future assignments.

User level programs

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 OS/161 kernel. To create new user programs, you will need to edit the Makefile in `bin`, `sbin`, or `testbin` (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

Beginning with this assignment, please note that your *design documents* become an important part of the work you submit. The design documents should clearly reflect the development of your solution. They should not merely explain what you programmed. 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.

Note that it can often be hard to write (or talk) about new software design -- you are facing problems that you have not seen before, and therefore even finding terminology to describe your ideas can be difficult. There is no magic solution to this problem; but it gets easier with practice. The important thing is to go ahead and try. Always try to describe your ideas and designs to someone else (we suggest your partner; roommates seem to have a low tolerance for this sort of thing). In order to reach an understanding, you may have to invent terminology and notation-this is fine (just be sure to explain it to your AI in your design document). If you do this, by the time you have completed your design, you will find that you have the ability to efficiently discuss problems that you have never seen before. Why do you think that CS is filled with so much jargon?

To help you get started, we have provided the following questions as a guide for reading through the code. We recommend that you divide up the code and have each partner answer questions for different modules. After reading the code and answering questions, get together and exchange summations of the code you each reviewed. Once you have done this, you should be ready to discuss strategy for designing your code for this assignment.

Begin Your Assignment

Before you start work on this assignment, tag your Git repository. The purpose of tagging your repository is to make sure that you have something against which to compare your final tree. Make sure that you do not have any outstanding updates in your tree. Use `git commit` and `git push` to get your tree commited in the state from which you want to begin this assignment.

Then, tag your repository exactly as shown below.

```
% git tag -a asst2-begin -m "Begin Assignment 2."
% git push origin master --tags
```

Configure OS/161 for ASST2

The procedure for configuring a kernel is the same as in ASST0, except you will use the ASST2 configuration file:

```
% cd kern/conf
% ./config ASST2
```

You should now see an ASST2 directory in the `compile` directory.

Building for ASST2 When you built OS/161 for ASST0, you ran make from `compile/ASST0`. In ASST2, you run make from (you guessed it) `compile/ASST2`.

```
% cd compile/ASST2
% make depend
% make
```

If you are told that the compile/ASST2 directory does not exist, make sure you ran config for ASST2.

"Physical" Memory

In order to execute the tests in this assignment, you will need more than the 512 KB of memory configured into System/161 by default. We suggest that you allocate at least 2MB of RAM to System/161. This configuration option is passed to the `busctl` device with the `ramsize` parameter in your `sys161.conf` file. Make sure the `busctl` device line looks like the following:

```
31 busctl ramsize=2097152
```

Note: 2097152 bytes is 2MB.

Programming with OS/161

If your code is properly synchronized, the timing of context switches and the order in which threads run should not change the behavior of your solution. Of course, your threads may print messages in different orders, but you should be able to easily verify that they follow all of the constraints applied to them and that they do not deadlock.

Built-in thread tests

When you booted OS/161 in ASST0, you may have seen the options to run the thread tests. The thread test code uses the semaphore synchronization primitive. You should trace the execution of one of these thread tests in GDB to see how the scheduler acts, how threads are created, and what exactly happens in a context switch. You should be able to step through a call to mi_switch() and see exactly where the current thread changes.

Thread test 1 ("tt1" at the prompt or `tt1` on the kernel command line) prints the numbers 0 through 7 each time each thread loops. Thread test 2 ("tt2") prints only when each thread starts and exits. The latter is intended to show that the scheduler doesn't cause starvation -- the threads should all start together, spin for awhile, and then end together.

Debugging concurrent programs

`thread_yield()` is automatically called for you at intervals that vary randomly. While this randomness is fairly close to reality, it complicates the process of debugging your concurrent programs.

The random number generator used to vary the time between these `thread_yield()` calls uses the same seed as the random device in System/161. This means that you can reproduce a specific execution sequence by using a fixed seed for the random number generator. You can pass an explicit seed into random device by editing the "random" line in your `sys161.conf` file. For example, to set the seed to 1 , you would edit the line to look like:

```
28 random seed=1
```

We recommend that while you are writing and debugging your solutions you pick a seed and use it consistently. Once you are confident that your threads do what they are supposed to do, set the random device to `autoseed`. This should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated.

General Questions

Include the answers to these questions in a file named "questions.txt" with your final assignment.

To implement synchronization primitives, you will have to understand the operation of the threading system in OS/161. It may also help you to look at the provided implementation of semaphores. When you are writing solution code for the synchronization problems it will help if you also understand exactly what the OS/161 scheduler does when it dispatches among threads.

Thread Questions

1. What happens to a thread when it exits (i.e., calls `thread_exit()`)? What about when it sleeps?
2. What function(s) handle(s) a context switch?
3. How many thread states are there? What are they?
4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?
5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

Scheduler Questions

6. What function is responsible for choosing the next thread to run?
7. How does that function pick the next thread?
8. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

Synchronization Questions

9. Describe how `thread_sleep()` and `thread_wakeup()` are used to implement semaphores. What is the purpose of the argument passed to `thread_sleep()`?
10. Why does the lock API in OS/161 provide `lock_do_i_hold()`, but not `lock_get_holder()`?

Process Questions

11. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?
12. In a block/wakeup mechanism, a process blocks itself to wait for an event to occur. Another process must detect that the event has occurred, and wake up the blocked process. It is possible for a process to block itself and wait for an event that will never occur.
    1. Can the operating system detect that a blocked process is waiting for an event that will never occur?
    2. What reasonable safeguards might be built into an operating system to prevent processes from waiting indefinitely for an event?
13. Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume no threads in any other processes have access to the semaphore. Discuss your answers.
14. In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.
15. Five batch jobs, A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.
    1. Round robin
    2. Priority scheduling
    3. First-come, first-served (run in order 10, 6, 4, 2, 8)
    4. Shortest job first

16. For each of the following scheduling algorithms, discribe a major failing of the algorithm and if appropriate provide a pathological example that illustrates this failing.
    1. First come first served
    2. Shortest job first
    3. Priority scheduling

Synchronization Problem (Hungry Birds)

Given `n` baby birds and one parent bird. The baby birds eat out of a common dish that initially contains `F` portions of food. Each baby repeatedly eats one portion of food at a time, sleeps for a while, and then comes back to eat. When the dish becomes empty, the baby bird who empties the dish awakens the parent bird. The parent refills the dish with `F` portions, then waits for the dish to become empty again. This pattern repeats forever.

Represent the birds as threads and develop code that simulates their actions. Use synchronization primitives to ensure that only one bird (either baby or parent) is using eating or filling the bowl at any one time, that no baby bird starves, and that the bowl is refilled in a timely manner. Implement this problem in OS/161 as a menu item in your kernel. You may want to construct this program in Linux first to isolate it from the implementation of the needed features in your instance of OS/161.

Design and Implementation

Locks

Implement locks for OS/161. The interface for the lock structure is defined in `kern/include/synch.h`. Stub code is provided in `kern/threads/synch.c`. You can use the implementation of semaphores as a model, but **do not** build your lock implementation on top of semaphores or you will be penalized.

Condition variables

Implement condition variables for OS/161. The interface for the cv structure is also defined in `synch.h` and stub code is provided in `synch.c`.

System calls and exceptions

Implement system calls and exception handling. 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 should implement:

- getpid
- fork
- execv
- waitpid
- _exit (properly for this assignment)
- write (improperly for this assignment)

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 decribed in the man pages.

Additionally, your miscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the man pages. Some of the grading scripts rely on the return of appropriate error codes; 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 OS/161 (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`. You need to think about a variety of issues associated with implementing system calls. Perhaps the most obvious one is: can two different user-level processes (or user-level threads, if you choose to implement them) find themselves running a system call at the same time? Be sure to argue for or against this, and explain your final decision in the design document.

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 associated with pid maintenance, and only then implement `getpid()`.

fork(), execv(), waitpid(), _exit()

These system calls are probably the most difficult part of the assignment, but also the most rewarding. 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()`, although "only" a system call, is really the heart of this assignment. It is responsible for taking newly created processes and make theme execute something useful (i.e., something 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. (Also, notice that `runprogram()` doesn't take an argument vector -- but this 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, 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.

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. 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`.

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.

write()

At the moment, our user programs are severely limited in that they cannot output anything other than an exit code. In order to be able to perform testing, debugging, and validating the behavior of our system calls, we need to, at the minimum, support some form of `write()`. For this assignment, we will rig all calls to `write()` to output to the console.

A temporary implementation of `write()`:

```
int sys_write(int fd, userptr_t buf, size_t size, int *retval) {
        int result;
        char tty[] = "con:";
        struct vnode *vn;
        struct uio u;

        (void) fd; // All writes go to "con:" for now

        result = vfs_open(tty, O_WRONLY, &vn);
        if(result != 0)
                return result;

        u.uio_iovec.iov_ubase = buf;
        u.uio_iovec.iov_len = size;
        u.uio_resid = size;
        u.uio_offset = 0;
        u.uio_segflg = UIO_USERSPACE;
        u.uio_rw = UIO_WRITE;
        u.uio_space = curthread->t_vmspace;
        result = VOP_WRITE(vn, &u);

        *retval = size - u.uio_resid;

        vfs_close(vn);

        return result;
}
```

Scheduling

Right now, the OS/161 scheduler implements a simple round-robin queue. As we learned in class, this is probably not the best method for achieving optimal processor throughput. For this assignment, you should implement two more scheduling algorithms.

You should select two fairly different scheduling algorithms and in your design document, explain why you selected the ones you did, and under what conditions you expect each to perform better. We suggest selecting one extraordinarily simple scheduling algorithm (e.g., pick a random process to run) and one more complex algorithm. For each, think carefully about what information you will need to maintain in order to completely implement the algorithm.

You may want your schedulers to be configurable. For example, for a round robin scheduler, it should be possible to specify the time slice. For a multi-level feedback queuing system, you might want to specify the number of queues and/or the time slice for the highest priority queue.

It is OK to have to recompile to change these settings, as with the HZ parameter of the default scheduler. And it is OK to require a recompile to switch schedulers. But it shouldn't require editing more than a couple `#defines` or the kernel config file to make these changes.

In any event, OS/161 should display at bootup which scheduler is in use.

Test your scheduler by running several of the test programs from `testbin` (e.g., add.c, hog.c, farm.c, sink.c, kitchen.c, ps.c) using the default time slice and scheduling algorithm. Experiment with the different scheduling algorithms that you implemented. Write down what you expect to happen with each algorithm. Then compare what actually happened to what you predicted and explain the difference.

Design Considerations

Here are some additional questions and thoughts to aid in writing your design document. They are not, by any means, meant to be a comprehensive list of all the issues you will want to consider. You do not need to explicit answer or discuss these questions in your design document, but you should at least think about them.

Your system must allow user programs to receive arguments from the command line. For example, you should be able to run the following program:

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

        args[0] = "add";
        args[1] = "2";
        args[2] = "3";
        args[3] = NULL;

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

which will load the executable file `add`, install it as a new process, and execute it. The new process will then perform the addition of "2" and "3" and output it.

Some questions to think about:

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?

You will need to "bullet-proof" the OS/161 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).

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 the shell invokes the `cat` command, and the `cat` command starts to read `file1`, what will happen if another program also tries to read file1? What would you like to happen?

Design Questions

In a directory called "asst2", you should draft a design document in some well-standardized format (e.g., .txt or .pdf -- **No MS Word or Open Office documents**). Your design document is worth 30% of the grade for this assignment. It should contain:

- Answers to the written questions above (except for "hungrybird").
- A high level description of how you are approaching the problem.
- A detailed description of the implementation (e.g., new structures, why they were created, what they are encapsulating, what problems they solve).
- A discussion of the pros and cons of your approach.
- Alternatives you considered and why you discarded them.

How to Proceed

Before you begin:

Tag your Git repository

```
% git tag -a asst2-begin -m "Begin Assignment 2."
% git push origin master --tags
```

Before the design doc is due:

1. Meet with your partner. Review the files described above. Separately answer the questions provided. Compare your solutions.
2. Discuss high level implementations of your solutions. Do detailed design in parallel with your partner.
3. Decide which functions you need to change and which structures you may need to create to implement the system calls. Discuss how you will pass arguments from user space to kernel space (and vice versa). Discuss how you will keep track of running processes. For which system call is this useful?
4. Discuss how you will implement the `execv()` system call. How is the argument passing in this function different from that of other system calls?

On the design due date:

1. Commit your final design documents and tag your repository.

```
% git tag -a asst2-design -m "Assignment 2 design."
% git push origin master --tags
```

Before the assignment is due:

1. Carefully divide up the work. `execv()` might be the single most demanding part of the assignment, but `waitpid()` is non-trivial as well.
2. For the parts you're assigned, verify that the collaborated design will really work. If something needs to be redesigned, do it now, and run it by your partner.
3. Implement.
4. Test, test, test. Test your partner's code especially.
5. Fix. Perhaps you won't need this step. (We all need to dream, right?)

On the assignment due date:

1. Commit all your final changes to the system. Make sure your partner has committed everything as well. Make sure you have the most up-to-date version of everything. Re-run make on both the kernel and userland to make sure all the last-minute changes get incorporated.
2. Run the tests and generate scripts. If anything breaks, curse (politely of course) and repeat step 1.
3. Tag your Git repository
4. Make sure everything is commited and your assignment directory has all the required files (on the server). Finally, after finalizing all your commits tag your repository.

   ```
   % git tag -a asst2-end -m "End Assignment 2."
   % git push origin master --tags
   ```

   We will be checking out and grading this version of your repository. Be sure to verify that the tagged version is *exactly* the version you'd like to have graded. If you are unsure or paranoid, it would be wise to checkout this version and reverify everything.

If you have comments or suggestions, email *p536@cs.indiana.edu*

Author: Andrew Lumsdaine
E-Mail: p536@cs.indiana.edu

*Created:*Sat Aug 25, 2001
Modified: Mon 29-Oct-2012