Lab 3: Multiprogramming in NachOS

CMSC 442

Due: Friday, Oct. 13th by 11:59:59pm

(Note: This lab, and all the other Nachos labs were derived from the original NachOS projects by Tom Anderson at UC Berkeley. They have been modified to fit our lab environment and changes in the compilation software since NachOS was originally published.)

Understanding processes and address spaces

The ability to run multiple programs simultaneously is called "multiprogramming". In the previous lab, you worked with test cases in which you created multiple threads by hand as part of the operating system. In this assignment, you will work on code that allows nachos to load and run programs that have been compiled separately from the operating system as independent executable files.

So far, all the code you have written for NachOS has been part of the operating system **kernel**. In a real operating system, the kernel not only uses its procedures internally, but allows user-level programs to access some of its routines via "system calls".

In this assignment we are giving you a simulated CPU that models a real CPU. We cannot just run user programs as regular UNIX processes, because we want complete control over how many instructions are executed at a time, how the address spaces work, and how interrupts and exceptions (including system calls are handled). Instead we use a simulator.

Our simulator can run normal programs compiled from C – see the Makefile in the `test' subdirectory for an example. The compiled programs must be linked with some special flags, then converted into Nachos format, using the program "coff2noff" (which we supply). The only caveat is that floating point operations are not supported. We provide several example programs in the folder "test", which also contains a Makefile that uses the appropriate commands to cross compile the nachos programs.

As in the first assignment, we give you some of the code you need; your job is to complete the system and enhance it.

The first step is to **read and understand** the part of the system we have written for you. Most of the code you will be working with in this project is in the "userprog/" directory. The existing code can run a single user-level 'C' program at a time. As a test case, we've provided you with a trivial user program, 'halt'; all halt does is to turn around and ask the operating system to shut the machine down.

This program is located in the test/ folder. The code for the program is in halt.c. To compile it, cd to that folder and type "make". If it fails to compile, you may need to first run the makefile in the bin/ folder.

To run the program, type `nachos -x ../test/halt'. As before, trace what happens as the user program gets loaded, runs, and invokes the Halt() system call.

You may find the "road map" to Nachos helpful; it is linked from the course web site. Also, it is OK to change the definition of the "machine" parameters. For example, the amount of physical memory – if that helps you design better test cases (though this is probably unnecessary unless you run into issues).

Most of your work on this lab will be done in the userprog/ folder. The files for this assignment are

progtest.cc – test routines for running user programs.

addrspace.h, addrspace.cc – create an address space in which to run a user program, and load the program from the disk.

syscall.h – the system call interface: kernel procedures that user programs can invoke.

exception.cc – the handler for system calls and other user-level exceptions, such as page faults. In the code we supply, only the 'halt' system call is supported. You will need to add quite a bit of code here to support the remaining system calls.

bitmap.h, bitmap.cc – routines for manipulating bitmaps (this might be useful for keeping track of physical page frames)

filesys/filesys.h, filesys/openfile.h – a stub defining the NachOS file system routines.

For this assignment, we have implemented the Nachos file system by directly making the corresponding calls to the UNIX file system: this is so that you need to debug only one thing at a time. In the next lab, we'll implement a NachOS file system for real on a simulated disk.

machine/translate.h, machine/translate.cc – translation table routines. In the code we supply, we currently assume that every virtual address is the same as its physical address – this restricts us to running one user program at a time. You will generalize this to allow multiple user programs to be run concurrently. We will not ask you to implement real virtual memory support (with swapping) until the next lab. For now, every page must be in physical memory.

machine/machine.h, machine/machine.cc – emulates the part of the machine that executes user programs: main memory, processor registers, etc.

machine/mipssim.cc – emulates the integer instruction set of a MIPS R2/3000 processor.

machine/console.h, machine/console.cc – emulates a terminal device using UNIX files. A terminal is (i) byte oriented, (ii) incoming bytes can be read and written at the same time, (iii) bytes arrive asynchronously (as a result of user keystrokes), without being explicitly requested.

You will also probably want to create test programs in the test/ folder.

Step 2. Tasks

The first four tasks should be done as a group and you may need to get them working before you can work on the remaining tasks. The next three tasks should be assigned to a particular member of your group. You can all work on them, but one person in your group is responsible for THAT task and their grade on the project will largely depend on successful implementation of that task. I leave it up to you to decide who will take which task – I will not play referee in your internal group politics.

Task 1 (To be completed by the entire group):

Fix any problems with your Lab 2 implementation. In particular, make sure your Locks and Conditions work and that Thread::Join and Thread::Finish work properly. Alarm and Comm aren't used by the rest of the lab, so you should be able to continue without fixing them unless they cause deadlocks or other major issues, though I will take off points if they are still not working properly when you submit.

Task 2 (To be completed by the entire group):

Implement system call and exception handling. You must support ALL of the system calls defined in syscall.h except for thread fork and yield (which will be implemented as an individual task, below).

To do this, you will need to expand the if statement in userprog/exceptions.cc which currently only handles the "Halt" system call. You will need to add an "else if" clause for each of the system calls listed in userprog/syscall.h. I recommend that you write helper functions in a separate file and call them from your if statement (this will require adding the new file to Makefile.common).

Do not make any changes to syscall.h since it's used both by NachOS kernel code and by the test programs that will run in the O.S.

We have provided you an assembly-language routine "syscall" to provide a way of invoking a system call from a C function (UNIX has something similar – see "man syscall"). You'll need to implement Task 4 before you can code and test the "exec", "wait", "exit", and "join" system calls. The routine "StartProcess" in progtest.cc may be of use to you in implementing the "exec" system call.

If you read the comments in these files carefully, you will learn two useful things:

1. Arguments to the system calls are passed in registers 4 through 7. Any return value should be written to register 2. The Machine class has ReadRegister and WriteRegister functions for this.

Register 2 contains the "type" or of the system call. This is an integer which unique identifies which system call has been requested. These correspond to constants defined in "syscall.h". For example, the Halt() system call corresponds to the constant SC_HALT (which is 0).

Note: After processing most system calls, your code in exception.cc must also advance the program counter, which is stored in the "PCReg" register. The assembly language NachOS simulates is MIPS-based, which means every instruction is exactly 4 bytes long. In the next lab, you will need to modify this a bit to handle page faults (which simply repeat the current instruction), but you don't have to worry about that now. Halt is obviously an exception to this rule, since the operating system terminates when it is called.

2. Some of the system calls pass no arguments or pass only integer arguments. However, several of them pass pointers (usually character pointers – strings). These pointers are addresses in "user space". That is -- they are logical (virtual memory) addresses -- not "physical addresses". This means Nachos can't use them directly -- they must be translated. To do this, you will need to use the ReadMem and WriteMem functions in machine/translate.h.

These functions take one, two, or four bytes of memory and copy them from "user program" memory to "nachos" memory. Since you can only copy bytes in fixed size chunks, you will need to use loops to copy strings of arbitrary length.

For example, if the system call takes a string named "str", you might need to do something like this:

```
char* kernel_str = new char[256];
for (int i=0; i<256;++i) {
        int tmp;
        machine->ReadMem((int)&str[i], 1, &tmp);
        kernel_str[i] = (char)tmp;
}
do_something(kernel_str);
delete kernel_str;
```

This code reads one byte at a time from the “str” variable in the test program's virtual memory and puts it into “kernel_str” in NachOS kernel memory. You can pass kernel_str to a helper function that uses it to do the system call and then delete it.

To -write- to virtual memory, you will need to do something similar (but using WriteMem instead of ReadMem and changing the order of a few statements). I highly recommend that you create functions to handle this for both reading and writing that you can reuse this code from any system call. This will reduce code duplication and make it much less likely that your code encounters subtle and difficult to remove bugs.

Note that you need to “bullet-proof” the NachOS kernel from user program errors – there should be nothing a user program can do to crash the operating system (other than calling halt directly).

Probably the easiest systems calls to implement are Create and Yield. The remaining file handling calls: Open, Close, Read, and Write are also fairly straightforward. Exit and Join are slightly more difficult as they involve creating data structures to track the exit status of a process. Probably the hardest of these system calls to implement is Exec, which requires changes to the address space class, but without Exec, you can’t really implement and test Exit and Join. The calls to thread fork and yield are an individual task (Task 7) and will also require significant changes to the Address Space class.

Create

Create is probably easiest system call to implement and test. To implement Create, you should use the "Create" function of the FileSystem class in filesys/filesys.h. You can test it by creating a program in the test/ folder that calls the create system call (use the code in halt.c as a model – don't forget to update the Makefile in test/ to build your test case). You can then use “ls” at the Linux command prompt to see if the file has been created in your userprog/ directory.

Until you implement Read and Write and the SynchConsole class (Task 3), you won't have an easy way to print debugging statements. If you get Create working (and tested), you can kind of hack around this by using the Create system call at different points in your test code to create files you can see with “ls”.

This is pretty hacky – but it works!

Open

To implement Open, you should use the "Open" function of the FileSystem class which returns an OpenFile pointer. You will need to store that pointer in some kind of per-process data structure. This “file table” should be created and initialized in the Addrspace class.

A vector or array of OpenFile pointers works well for this. You should return a unique integer id that can be used later to retrieve the pointer from your data structure. You may want to reserve the first two spaces in this data structure for Console input and output (i.e. cin and cout). The Read and Write functions will use that id number to access the pointer.

(Note that, unusually, “Open” is in the “.h” file not in filesys.cc where member functions usually go).

Close

There is no "Close" function in the OpenFile class. Instead, the destructor closes the file. Your close system call should delete the corresponding OpenFile object and remove it from the file table data structure. You should implement this in a way that makes it possible to open at least 254 simultaneous files, close them, and open 254 more files. That is, we should be able to re-use closed file descriptors.

Read and Write

The read and write system calls are used for two separate tasks. If passsed a file descriptor (OpenFileId) of ConsoleInput (i.e. 0) or ConsoleOutput (i.e. 1), they are used for Console I/O. Console I/O is discussed in Task 3. If passed a file descriptor greater than one they read and write to a file. To implement file read and write, you should use the file descriptor to look up the pointer “Open” placed in your file table and then use the Read and Write functions in filesys/openfile.h to get input or produce output to a file.

The remaining system calls are discussed in Tasks 4-7.

Task 3 (To be completed by the entire group):

In order to really test your code, you need to be able to print to the console. Add if statements to your Read and Write functions so that if they are passed the "ConsoleInput" or "ConsoleOutput" fds (0 and 1 respectively) they will read from or write to the console instead of a file. Iif the program tries to read from ConsoleOutput or write to ConsoleInput, the behavior is undefined, but you should ensure that NachOS does not crash.

NachOS provides a Console class which uses interrupts to handle Console I/O. It will allow you to read or write a single character at a time from the console, but it does not handle printing of complete strings. This means that access to this console device is currently "Asynchronous". If two threads print a line of text, that line can be interrupted, causing a race condition.

Also, access to the Console is a producer/consumer system -- we don't want to read from the console buffer until data is actually there (otherwise, we'll get garbage) and we don't want to write to the console while the previous character is still being output (otherwise, we might clobber it and not print everything).

To support Console Read and Write, create a “SynchConsole” class in “synchconsole.h” and “synchconsole.cc” (in the userprog/ folder) that provides the abstraction of synchronous access to the console. Then create a global SynchConsole object in threads/system.cc which your system calls can use. The file “progtest.cc” has some example code for using the Console class. You can use this as the basis for the functions in your SynchConsole. However, I highly recommend using Locks and Conditions for this instead of just semaphores.

Task 4 (To be implemented by the entire group):

Every thread in NachOS is given its own AddrSpace (address space) object which keeps track of which pages of physical memory have been allocated to it. Right now, we assume that only one process is running at a time. This means that if you try to load two programs (using “Exec”), they would be loaded into the same memory frame, overwriting each other.

Implement multiprogramming with time-slicing. Most of the code for this will probably go either in the StartProcess function in “progtest.cc” or in the Addrspace constructor. The code we have given you is restricted to running one user program at a time. You will need to:

(a) come up with a way of allocating physical memory frames so that multiple programs can be loaded into memory at once (The data structure in bitmap.h is useful for keeping track of which blocks of memory are used and which are free),

(b) provide a way of copying data to/from the kernel and from/to the user's virtual address space (now that the addresses the user program sees are not the same as the ones the kernel sees) as discussed in Task 2, and

(c) use timer interrupts to force threads to yield after a certain number of ticks. This can be done using a Timer object as you did for the alarms in Lab 2. Note that scheduler.cc saves and restores user machine state on context switches automatically.

You also need to implement the remaining system calls (Exit, Join, and Exec). These system calls mostly involve process control. To implement them you will need a data structure for mapping "Process IDs" to threads. This “process table” will be used by at least the Exit and Join system calls. You may also need to give each AddrSpace object a way to keep track of its own Process ID.

Exit

The exit system call causes a thread to exit (calls Thread::Finish) and sets the thread's exit status. When a process calls Exit, you should store its exit status in the process table using its process id. This exit status is an integer value which is set by the Exit system call and retrieved (in another thread) by the Join system call.

Join

When a process calls Join, it will pass in a process id. You should look up that process id in the process table to obtain a thread pointer, use thread->Join to wait until that thread has finished, and then return the exit status of the joined thread.

Exec

Exec in NachOS is more like the exec syscall in Windows than the one in Linux. It both creates a new thread (with a new address space) AND loads a program from disk into the address space for that thread. In Linux terms, this is like doing both a "fork()" and and "execve()". The parent thread continues running as before.

The "StartProcess" function in userprog/progtest.cc already does this for the "main process" that you run with "nachos -x", so it's a good example for you to follow in Exec.

Exec also adds the thread to the process table and returns the process id of the new thread (so that the parent process can call "Join" on it later).

Task 5 (To be implemented by a single member of the group): Exec with arguments

The "exec" system call does not provide any way for the user program to pass parameters or arguments to the newly created address space. UNIX does allow this -- for instance, to pass in command line arguments to the new address space. Implement this feature! (Hint: there are two ways to do this. One is to copy the arguments onto the bottom of the user address space (the stack) and then pass a pointer to the arguments as a parameter to main using Register 4 to hold the pointer. This is how UNIX does this.

The other (probably easier) approach is to add new system calls, which every program calls as the first thing in main, that fetch the arguments to the new program. This does require modifying start.s in "test/".

Task 6 (To be implemented by a single member of the group): cat and cp

Nachos provides a "shell" program that provides you with a simple command prompt, but very little other functionality. Implement Nachos versions of the `cat` and `cp` commands in the "test" folder. Remember that "cat" can take several files as arguments and prints them all to the console while "cp" opens a file, reads its contents, and writes them to a different file (possibly creating it or, if it already exists, overwriting it).

Task 7 (To be implemented by a single member of the group):

Implement multi-threaded user programs. Implement the thread fork and yield system calls, to allow a user program to fork a thread to call a routine in the same address space, and then ping-pong between threads.

Yield

The Yield system call simply causes the current thread to yield the CPU. This is probably the easiest system call of all to write (even easier than Create!)

Fork

Forking is like a lightweight Exec. It creates a new thread, but doesn't load a new process into its address space.

When a process calls Fork, you must allocate a new AddrSpace for it and then call Thread::Fork. The new address space should have its own set of pages, distinct from those in the parent process, for its stack. However, it should share all other pages (including those for the code segment and the data segment) with its parent process. This will require you to write a new address space constructor.

Step 3. Documentation

In your base nachos directory, I would like you to create a file named LAB3-README.TXT that contains:

Your group submit name
The names of each member of your group
A description of the contributions of each member (in paragraph form).

Your description should be detailed, explaining which files you changed and which task they implemented. If you were unable to get part of a task working, you should be explicit about that and explain the problem in detail.

Submitting

As usual, create a tarball of the nachos directory and submit it through the course web site.