Assignment 2: System Calls and Multiprogramming

Remember to do your work in accordance with USC's USC code of Academic Integrity.

The second phase of Nachos is to support multiprogramming. 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. Nachos 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. Run the program `nachos -x ../test/halt' from the userprog directory. As before, trace what happens as the user program gets loaded, runs, and invokes a system call.

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 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, the 'halt' system call is supported, as well as Read, Write, Open and Close - file handling system calls.

- ;bitmap.h, bitmap.cc - routines for manipulating bitmaps (this might be useful for keeping track of physical page frames)
- filesys.h, openfile.h (found in the filesys directory) - 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 assignment four, we'll implement the Nachos file system for real on a simulated disk.
- translate.h, translate.cc - translation table routines. In the code we supply, **we 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 virtual memory support until assignment 3; for now, every page must be in physical memory.
- machine.h, machine.cc - emulates the part of the machine that executes user programs: main memory, processor registers, etc.

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 them via **system calls**.

In this assignment we are giving you a simulated CPU that models a real CPU. In fact, the simulated CPU is the same as a real CPU (a MIPS chip), but 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.
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 - use int's and char's.

You are to compile Nachos in the userprog directory. Nachos user programs are to be compiled from the test directory. Once this is done, you can run a simple test of the file handling system calls using this command:

    nachos -x ../test/testfiles (run Nachos from the userprog directory)

You should see the following output (or something very close).

testing a write
Machine halting!

Ticks: total 1854, idle 1600, system 170, user 84
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 16
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

Assignment

1. (10 points) Implement system calls and exception handling. This consists of two parts:
   - You must support the following system calls already defined in syscall.h: Fork, Exec, Exit, and Yield. You will complete the implementation of Exec and Fork in part two, because you need multiple address spaces to be able to test their proper functionality. We have provided you an assembly-language routine, "syscall", to provide a way of invoking a system call from a C routine (UNIX has something similar - try 'man syscall'). **You are NOT to implement Join.**
   - You are to implement system calls for the Lock class Acquire() and Release() methods and the Condition class Wait(), Signal(), and Broadcast() functions. These are five NEW system calls that do not exist in Nachos currently.

You must also implement constructor and destructor system calls to create a Lock and a Condition object. These are to be called: CreateLock, DestroyLock, CreateCondition, DestroyCondition. The CreateLock and CreateCondition system calls can either take no argument, or you can provide a character array parameter to give them a name. It is your choice. When calling CreateLock or CreateCondition, you are to return an integer value. This value is an index position into a kernel structure array of actual Lock and Condition objects. You are not to return a kernel space pointer, as that would allow the user program to manipulate it directly. The user program sees it as an identifier that is somehow associated by you (in the kernel of Nachos) with the kernel-level Lock, or Condition, object that you created. This way, on an Acquire system call, you pass an int argument - ; the value returned by the CreateLock system call - so the OS can use that argument (cast appropriately to a Lock pointer in kernel mode) to get to the Lock object created in CreateLock.
For the DestroyLock and DestroyCondition system calls, they take a single integer parameter - the identifier for the Lock or Condition kernel object that is to be deleted.

We have provided code for the five system calls that are concerned with files: Create, Open, Close, Read and Write. NOTE: To output messages from within a Nachos user program, you must use the Write system call. printf will not work, as it is a Unix system call and is not implemented in Nachos user programs (you can use it in kernel mode).

The Exit system call must ensure that Thread::Finish is called, except for the very last thread running in Nachos. For the very last thread (not the last thread in a process - unless it's the last process in Nachos), you must call interrupt->Halt() to actually stop Nachos. If you don't do this, Nachos will not terminate. This assignment requires that Nachos terminates.

Yield is very simple to implement.

Exec and Fork require actual implementation (which you will do in part 2) - not just organizing calls to existing capabilities in the Nachos kernel. You will have to deal with creating address spaces, initializing them with the code of the program to be run when necessary, creating a new thread, and binding the thread to its address space and creating a new stack.

You will have to maintain a process table, in order to maintain the mapping from the SpaceIds that you pass up to the user level, and the actual address space data structures. When Exiting a thread, if it is the last thread in the process, you will need to ensure that you delete the address space (and all other resources). You may think that you don't need a process table for this project, but trust me, you will definitely need it for project 3 - besides, every OS has a process table.

Note that you will 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 (with the exception of explicitly asking the system to halt). Bullet-proofing includes, but is not limited to making sure that any pointers passed to the nachos kernel point to valid memory in the user's address space, and that any parameters are reasonable. You may set limits on parameters (i.e. maximum buffer sizes or other values) but they must be documented in your writeup.

Your answer to this part must include a test suite of user programs that demonstrate the correctness of your system calls (other than Exec and Fork which are covered in part 2). You must demonstrate not only that your system calls work when used correctly, but that they do not crash Nachos when used incorrectly. You should exercise your creativity in finding many different possible types of abuse and showing that your system is safe from them.

Document your test suite. Tell what error each program or subroutine is checking for and make the output as clear as possible. Comment your test code. The test suite should be user programs like the halt example. Note that your test case should also demonstrate the line buffering behavior of your synchronized console.

(10 points) Implement multiprogramming. The code we have given you is restricted to running one user program at a time. You will need to:
Come up with a way of allocating physical memory frames so that multiple programs can be loaded into memory at once (cf. bitmap.h). Some of the work has already been done - see the AddrSpace constructor.

Provide a way of copying data to/from the kernel 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). For big hints on how this is done, see the file handling system calls that we've provided.

Complete the implementation of the Exec and Fork system calls.

Note that scheduler.cc now saves and restores user machine state on context switches - when you compile from the userprog directory.

Now that you have independent address spaces, and a way of switching between processes in a controlled manner, you must complete the implementation of Exec. The routine 'StartProcess' in progtest.cc may be of use to you in implementing the 'Exec' system call.

Nachos Exec creates a new address space and starts a single thread running that code. You can think of it as a combination of Unix fork and exec.

Design and demonstrate a test suite that shows that Exec and Fork perform correctly, when used in any combination and cannot be used to break the OS. Your tests must show not only that proper input to Fork and Exec work properly, but that improper input is captured and dealt with appropriately. In no case should Nachos terminate abnormally.

2. (20 points) Implement part 2 of project 1 as a set of multithreaded Nachos user programs. You must use the new Lock and Condition system calls that you implemented in part 1 for synchronization.

You will have to use the Fork system call to create all your threads.

If your Exec system call is working properly, then it should be possible to have multiple "Airports", each running independently, executing by different processes simultaneously.

You are to provide a test suite that proves your implementation is correct. You must fully explain what Locks and Condition Variables you use in your solution and how it guarantees proper synchronization between the threads in your tests.

Adding New System Calls

You will need to make the following changes to add the new system calls - so the trap interrupt will work.

Change start.s in the test subdirectory. You will need to copy and paste the lines from one of the other system calls for your new system calls. I've included the lines below from Halt. Each system call will need the same set of lines, changing 'halt' to your new system calls: Acquire, Release, Wait, Signal, and Broadcast, etc.
.globl Halt
.ent Halt
Halt:
  addiu $2,$0,SC_Halt
  syscall
  j   $31
.end Halt

Make two changes to syscall.h.

1. Add the new syscall codes. The existing codes in syscall.h are below. You must start with a number greater than these numbers.

/* system call codes -- used by the stubs to tell the kernel which system call * is being asked for */
#define SC_Halt  0
#define SC_Exit  1
#define SC_Exec  2
#define SC_Join  3
#define SC_Create 4
#define SC_Open  5
#define SC_Read  6
#define SC_Write 7
#define SC_Close 8
#define SC_Fork  9
#define SC_Yield 10

2. Add the new function prototypes for the new system calls. The function prototype for Halt() in syscall.h is shown below.

/* Stop Nachos, and print out performance stats */
void Halt();