Nachos Assignment #2: Multiprogramming

Tim Brecht
Computer Science 4321/5421
Due date: Tuesday – October 21, 1997, 2:30 p.m.

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. Our 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. Run the program 'nachos -x ../test/halt'. 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, only the 'halt' system call is supported.

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.

The following files are found in the directory "code/machine":

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 in 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.

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

console.h, 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, and (iii) bytes arrive asynchronously (as a result of user keystrokes), without being explicitly requested.

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. In fact, the simulated CPU is the same as the 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.

The assignment is items 1 and 2 below plus test programs to demonstrate that your implementation works and is correct (see the directory code/test for example test programs).

1. Implement system calls and exception handling. You must support all of the system calls defined in syscall.h, except for thread fork and yield, which can be implemented for extra credit. 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'll need to do part 2 of this assignment in order to test out the 'exec' and 'wait' system calls; the routine 'StartProcess' in progtest.cc may be of use to you in implementing the 'exec' system call.

   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). Also, to support the system calls that access the console device, you will probably find it helpful to implement a "SynchConsole" class, that provides the abstraction of synchronous access to the console. "progtest.cc" has the beginnings of a SynchConsole implementation; look ahead to the file system assignment for the similar example for the SynchDisk class.

   NOTE: halt.c is one test program and shell.c has already been provided. You should write test programs that demonstrate that your system works and programs that also point out where/when your system does not work.

2. Implement multiprogramming with time-slicing. 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 segments so that multiple programs can be loaded into memory at once (cf. bitmap.h), (b) 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), and (c) use timer interrupts to force threads to yield after a certain number of ticks. Note that scheduler.cc now saves and restores user machine state on context switches.

   Explain how you would instrument the operating system to keep track of average response time for executing user programs. Also describe how you would write a test case (a set of user programs to run) that performs well using your policy for scheduling the CPU among user programs, and a test case that performs poorly (has very long average response time). What type of performance would you expect from these programs. Explain why your policy is good or bad. NOTE: for this final portion a written explanation is sufficient. You are NOT required to instrument your operating system or write and run the test cases.