Nachos Assignment #2: Multiprogramming

Tom Anderson
CSE 451
Due date: Tuesday, November 23, 5:00 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.
Up to now, 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".

The first step is to read and understand the part of the system we have written for you. The new code is spread over several directories. There are some new kernel files in "userprog", there are a few additional machine simulation files in "machine", and a stub file system in "filesys". The user programs are in "test", and utilities to generate a Nachos loadable executable are in "bin". Since Nachos executes MIPS instructions (and there aren't very many MIPS machines left!), we also provide you a cross-compiler. The cross-compiler runs on Linux and compiles user programs into MIPS format.

The code we provide 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. To run the "halt" program, in the "userprog" directory, make and then run nachos. Trace what happens as the user program gets loaded, runs, and invokes a system call.

You may find Narten's "road map" to Nachos helpful; there is a pointer to it on the class Web page. Also, it is OK to change the constants in "machine.h", for example, to change the amount of physical memory, if that helps you design better test cases. The files for this assignment include:

> **userkernel.h, userkernel.cc** --- routines for booting and testing a multiprogramming kernel.
>
> **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.

translate.h, translate.cc -- translation table routines. In the code we supply to run "halt", 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.h, 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.

synchconsole.h, synchconsole.cc -- provides synchronized access to the console device.

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 by simulating the execution, we have 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 "test" subdirectory for a few examples. 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.

1. 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 can be implemented for extra credit (see #5). 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.

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).

2. Implement multiprogramming. 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 (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) add synchronization to the routines that create and initialize address spaces so that they can be accessed concurrently by multiple programs. Note that scheduler.cc now saves and restores user machine state on context switches. You will probably want to write some utilities like UNIX “cp” and “cat”, and use the “shell” provided in the “test” subdirectory to verify that system call handling and multiprogramming are working properly.

3. Implement multilevel feedback or lottery scheduling to reduce average response time for running jobs on your system. Demonstrate that your scheduling algorithm reduces response time relative to the original Nachos scheduling algorithm (round robin) for at least one workload.

4. The current version of 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!

5. (Extra credit) Implement multithreaded 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 the threads. (Hint: you will need to change the kernel to allocate memory in the user address space for each thread stack.)