CS-230 Programming Assignment #2

Due date: Monday, November 22, 1993

The next phase of Nachos is to support multiprogramming. As before, you get 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 that has been written for you. That code can run a single user-level 'C' program at a time. As a test case, you are provided 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:

extra.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.

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

filesys.h, 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.

pte.h, pte.cc – page table management routines. We will not ask you to implement virtual memory support until the next assignment; 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 them via "system calls".

In this assignment you get a simulated CPU that models a real CPU. In fact, the simulated CPU is the same as the real CPU (a MIPS chip), but you 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.

The simulator can run normal programs compiled from C – see the Makefile in the 'test' subdirectory for an example. The compiled programs must be converted into Nachos format, using the program "coff2flat" (which is supplied), before being loaded onto the Nachos disk. Two caveats: (i) floating point operations are not supported and (ii) the 'main' procedure must be listed first in its C file (if there are multiple '.o' files, the one containing 'main' must be listed first on the 'ld' command line). This ensures that 'main' will be loaded at virtual address 0. In UNIX, 'main' can be located anywhere in a program, because a special procedure 'START' is located at a well-known virtual address (and START calls main).

1. Implement the 'exec' and 'wait' system calls. You are provided with 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'). The routine 'StartProcess' in extra.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).

2. Implement multiprogramming with time-slicing. The code given to you is restricted to running one user program at a time. You will need to: (a) provide a way of copying data to the kernel from 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 (b) 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.

3. Write a shell. A shell reads a command from the user via the console, then runs the command by invoking the kernel system call 'exec'. The UNIX program 'csh' is an example of a shell. Test your shell by running the programs 'io' and 'cpu'. Also try 'perftest' which runs 'io' and 'cpu' as concurrent processes. Explain what happens.

4. 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. Explain how you might go about implementing this feature (you don't have to implement this, just explain how you would implement it).