**CPS 536: Homework 2** **Dr. Clark**

Nachos Assignment #2: System Calls
Due Friday, February 28

The goal of this assignment is to begin adding support for user level programs and multiprogramming to the Nachos system. Nachos is currently able to run a single user level program using the *-x progname* command line argument.

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". At this time, only the *halt* system call is supported by Nachos.

In this assignment we are giving you a simulated CPU that models the real MIPS 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.

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.

1. 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.

   **You must copy some new files for this assignment.** The files are under the directory */home/rjc/pub/cps536/proj2* on the sun systems. There are new files for both the *test* and *userprog* directories. Also, in the *threads* directory there is a copy of a working lock implementation in case you were having trouble with yours. Copy the files into the correct directories in your nachos source.

   The general 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.

   synchconsole.h, synchconsole.cc – provides a synchronous interface to the system console.

   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. For the next assignment you will generalize this to allow multiple user programs to be run concurrently. 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.

2. Implement I/O system calls. For this assignment, you must support all of the following I/O system calls in syscall.h:

   (a) *Create(char *name)* Create a file with the given name. (see filesys.h)

   (b) *Open(char *name)* Open a file with the given name and return a *OpenFileId* that can be used for future system calls related to this file. You should return some invalid file identifier (e.g. -1) if the open fails. (see filesys.h)

   (c) *Read(char *buffer,int size, OpenFileId id)* Read *size* bytes from the open file into *buffer*. Return the number of bytes actually read – if the open file isn't long enough, or if it is an I/O device, and there aren't enough characters to read, return whatever is available (for I/O devices, you should always wait until you can return at least one character). (see openfile.h)

   (d) *Write(char *buffer,int size, OpenFileId id)* Write *size* bytes from the *buffer* into the open file. (see openfile.h)

   (e) *Close(OpenFileId id)* Close the file after finishing reading and writing. (see openfile.h)

   The other system calls will be implemented as part of the next assignment. You will need to implement both console and disk I/O in this assignment. When an address space starts up, it has two open files, representing keyboard input and display output (in UNIX terms, stdin and stdout). These are defined in Nachos as *ConsoleInput* and *ConsoleOutput*. Read and Write can be used directly on these, without first opening the console device. **Note: I strongly recommend that you start by first implementing Read and Write to the console. This is much easier than file I/O and will help you learn your way around the system.**

   Nachos provides 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 will need to look at and likely modify the routine *StartProcess* in "progtest.cc".

   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).

   To support the system calls that access the console device, you will want to use the *SynchConsole* class that I have provided. This class provides the abstraction of a synchronous access to the console. This class is based on the *SynchDisk* class that is part of the filesystem code.

3. Test! Test! Test! Make sure your system calls work and that it is impossible to crash Nachos by calling a system call with a bogus argument. Testing will be easier now since you can write test programs in user space that use your system calls and you won't need to recompile Nachos to run a new test. **Note:** The test programs are written in C, not C++. The cross compiler doesn't handle C++ constructs and comments. Also, there is no C runtime library!!!