EECS 211
Advanced System Software
Winter 2006

Assignment 4

Posted: February 9, 2006
Due: February 16, 2006

Topic: User programs and system calls in Nachos

Instructions:

The goal of this fourth assignment is to develop, implement and test support for user programs making system-calls to the Nachos kernel. This assignment follows the first task of “Nachos Assignment 2” described in the file **`doc/userprog.ps`** of the Nachos installation. The instructions below assume that you read **`doc/userprog.ps`** in parallel.

Task 1: Understand the given framework

Go into the **`userprog`** directory. Run the given program **`nachos`** with the given user-program **`../test/halt`** to test the given code. Trace the execution path by using the built-in debugging facilities. Run the program step by step using the debugger **`gdb`**. Finally, read in detail through the given sources provided in the **`userprog`** directory.
Make sure you understand what is going on when the user program is compiled, is loaded, executes, issues a system call, and dies.

To fully understand the user program execution on the emulated MIPS machine, read also the sources in other directories (e.g. **`machine`**), as listed in the **`doc/userprog.ps`** document. Note that, however, you will only need to change files in the **`userprog`** directory for this particular assignment. All other files should be left unmodified.

Deliverable 1: (20 points)

Briefly describe in a text file **`task1.txt`** the compilation, loading, execution, system-call, and termination of user programs in the Nachos environment (5-10 sentences).
Briefly describe also the boundary between user- and kernel-land in Nachos. Specify for critical functions whether they belong to kernel- or to user-land (about 5 sentences).

Task 2: Implement basic exception handling and system calls for file I/O

See item 1 in **`doc/userprog.ps`**.
Modify and complete the code in file **`exception.cc`** to support the exception types listed in **`../machine/machine.h`** and the system calls listed in **`syscall.h`**. To do this, implement a (big) **`switch`** statement in the function **`ExceptionHandler()`**with one **`case`** for each exception type. The **`SyscallException`** should be handled by a new function **`SystemCall`** that again contains a (big) **`switch`** statement to handle each type of system call. All this code should go into file **`exception.cc`**.
Note that, except for the **`SyscallException`**, all exceptions are fatal errors for the user program at this time (in later assignments, we will change that). Thus, the kernel should print an error message (for us to observe the error) and then cleanly terminate the user program.

We will first limit ourselves to support only basic system-calls. For this assignment, your code should support the following 7 system-calls:

(a) SC_Halt
(b) SC_Exit
(c) SC_Create
(d) SC_Open
(e) SC_Read
(f) SC_Write
(g) SC_Close

For the file I/O system calls, you should support input from the console (**`OpenFileId ConsoleInput`**, alias **`stdin`**), output to the console (**`OpenFileId ConsoleOutput`**, alias **`stdout`**), and input and output to regular files (**`OpenFileId > 1`**). For console I/O, it will be necessary to implement a synchronous console class (for simplicity, place the class **`SynchConsole`** into the file **`exception.cc`**). You will find the class **`SynchDisk`** provided in the **`filesys`** directory very helpful as it contains very similar functionality.

To properly handle the file I/O system calls, you will need to maintain a list of open files for each process. Class **`AddrSpace`** (in files **`addrspace.h`** and **`addrspace.cc`**) is a good place to keep this list and its maintenance functions because each process is now assigned such a space (via the **`Thread->space`** pointer). To keep things simple, maintain an array of 5 entries for open files. The first two entries should be reserved for **`ConsoleInput`** (alias **`stdin`**) and **`ConsoleOutput`** (alias **`stdout`**). Make sure to check parameters provided by I/O system calls properly and cleanly abort user programs which attempt to write into an unopened file or try to read from **`stdout`**, etc. Also, make sure to close any files left open when the user program exits or is aborted.

Note that in order to have a “bullet-proof” kernel, all possible “bad” things a user program may do (e.g. raising unsupported exceptions or providing invalid arguments to system calls), must not disturb any kernel data structures, nor any other processes. Instead, a misbehaving application must be properly terminated and cleaned up. Make sure that your implementation takes care of this.

Deliverable 2: (20 points)

a) Extended source file **`exception.cc`**.
b) Extended source files **`addrspace.h`** and **`addrspace.cc`**.

Task 3: Validate your implementation using simple test programs

To test your exception handling and the implemented system calls, create a set of simple Nachos user programs as test cases and run them on your kernel. To start, you may want to take a look at the few examples that are already provided in the **`test`** directory.

(a) Program **`HelloWorld.c`**:
should print the string HelloWorld to the console and then cleanly exit
(b) Program **`Name.c`**:
should ask the user for her/his name and then print it backwards
(c) Program **`Copy.c`**:
should ask the user for two file names and copy the contents of the first file into the second file

You should also test if you kernel is “bullet-proof”. Create and run the following “bad” examples:

(d) Program **`WriteToNull.c`**:
tries to assign the value 42 to memory address 0
(e) Program **`DivisionByZero.c`**:
tries to divide 42 by 0
(f) Program **`WriteToStdin.c`**:
tries to write a character ‘**`x`**’ to the standard input stream

Deliverable 3: (30 points)

For each of the programs above, submit its source file (e.g. **`HelloWorld.c`**) and a corresponding log file (e.g. **`HelloWorld.log`**) showing that the program successful runs (or fails!) on your Nachos kernel.

Submission instructions:

To submit your homework, send an email with subject “EECS 211 HW 4” to the course instructor at doemer@uci.edu. Please include the deliverables listed above as attachments.

To ensure proper credit, be sure to send your email before the deadline: February 16, 2006, 11:59pm.

--
Rainer Doemer (ET 444C, x4-9007, doemer@uci.edu)