EECS 211
Advanced System Software
Winter 2007

Assignment 4

Posted: February 19, 2007
Due: March 5, 2007

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.

Preparation: 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. However, note that you will only need to change files in the **`userprog`** and **`test`** directories for this assignment. All other files should be left unmodified.

Extra Credit: (up to 30 extra points)

Discuss this assignment on the course noteboard! Post at least one useful question, a helpful comment, or answer/discuss one of the posted questions!

Example topics for the discussion include the compilation, loading, execution, system-call, and termination of user programs in the Nachos environment, or the difference and boundaries between user- and kernel-land in Nachos. Details on the following implementation tasks are interesting topics as well, of course. Please, however, do not post solutions in form of complete source code!

Task 1: 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.

Note that you will need to copy data from kernel address space into user space. For example, for the SC_Open system call, you need to read a filename provided in user land. To achieve this cleanly, implement a set of dedicated memory copy functions in the kernel, with the following signatures:

```
void CopyToKernel(
     int FromUserAddress,
     int NumBytes,
     void *ToKernelAddress);
void CopyToUser(
     void *FromKernelAddress,
     int NumBytes,
     int ToUserAddress);
```

In addition, it will be convenient to have copy functions which handle null-terminated strings, as follows:

```
void CopyStringToKernel(
     int FromUserAddress,
     char *ToKernelAddress);
void CopyStringToUser(
     char *FromKernelAddress,
     int ToUserAddress);
```

To implement these functions, you can use the functions **`ReadMem()`** and **`WriteMem()`** which are declared in **`machine.h`** and implemented in **`translate.cc`**. Note that we will re-use these functions just for simplicity (actually, this is considered "dirty" because this uses internal functions of the machine simulation; see the comment above the function declaration in **`machine.h`**).

Further, 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.

Finally, please 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. Try to make sure that your implementation takes care of this as much as possible.

Deliverable 1: (30 points)

a) The extended source file **`exception.cc`**.
b) The extended source files **`addrspace.h`** and **`addrspace.cc`**.
c) A text file **`task1.txt`** that briefly outlines your implementation (i.e. status, open issues, and decisions taken).

Task 2: 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. Your user programs should include:

(a) Program **`HelloWorld.c`**:
should print the string "Hello Nachos World" to the console and then cleanly exit
(b) Program **`Echo.c`**:
should let the user type in a text string and echo it to the console when the user hits enter; should repeat until the user enters "quit"
(c) Program **`List.c`**:
should ask the user for a file name and print the contents of that file to the console; a non-existing file should be handled properly

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

(d) Program **`StoreAtZero.c`**:
tries to store the value 42 at memory address 0
(e) Program **`WriteToInvalidFile.c`**:
tries to write into an unopened file
(f) Program **`ReadFromStdout.c`**:
tries to read a string from the standard output stream

Deliverable 2: (30 points)

For each of the user programs listed above, submit the source file (e.g. **`HelloWorld.c`**). Submit also a text file **`task2.txt`** with a brief explanation and corresponding execution logs for each file. This should show that the program successful runs (or successfully fails!) when running in your Nachos environment.

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:
March 5, 2007, 11:59pm.

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