EECS 211
Advanced System Software
Winter 2009

Assignment 5

Posted: February 19, 2009
Due: March 4, 2009 at 12:00pm (noon)

Topic: User programs and system calls in Nachos

Instructions:

The goal of this assignment is to develop, implement and test basic support for user programs in Nachos. For simplicity, we will use only very simple user programs that make only a few system calls to the Nachos kernel.

This assignment follows Task 1 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 (as outlined in **`doc/userprog.ps`**).
Make sure you understand what is going on when the user program is compiled, is loaded, executes, issues a system call, and dies. (If you are not sure about an issue, make use of the course message board for technical discussion!)

To fully understand the user program execution on the emulated MIPS machine, review also the sources in other directories (i.e. **`machine`**). 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!

Task 1: Implement exception handling and system calls for basic 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, you need to implement a (rather large) **`switch`**

statement in the function **ExceptionHandler()** with one **case** for each exception type. The **SyscallException** should be handled by a new function called **SystemCall()** that again contains another (rather large) **switch** statement to handle each type of system call. All the necessary code should go into the file **exception.cc** (please use the provided template file as starting point, see below!)

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 a specific error message (for us to observe the error) and then cleanly kill the user program and shut down the machine.

For now, we will limit this assignment to support only the basic system calls. Specifically, 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 (c) through (g), 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**). You may assume a maximum of 5 open files per user program.

Implementation Hint 1:

For safe console I/O (i.e. input from **stdin** and output to **stdout**), it is necessary to use a synchronous console class. For your convenience, such a class **SynchConsole** is provided to you in a template file that you should use as starting point for this assignment. Simply copy the file
/users/faculty/doemer/eecs211/exception.cc.W09templateA5
over the existing file **exception.cc** in the **userprog** directory.

Implementation Hint 2:

You will need to copy data from the kernel address space into user space, and vice versa. For example, for the SC_Open system call, the kernel needs to read a filename provided by the program in user land.
To implement this cleanly, use a set of dedicated memory copy functions in the kernel. Appropriate function prototypes with the following signatures are provided in the template file:

```
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 that handle zero-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`**; however, for our purposes right now, this is just fine!).

Implementation Hint 3:

To properly handle the file I/O system calls, you will need to maintain a set of open files for each process. Class **`AddrSpace`** (implemented in files **`addrspace.h`** and **`addrspace.cc`**) is a good place to keep this set and its maintenance functions because each process is now assigned such a space (via the **`Thread->space`** pointer).

To keep things simple, maintain for each process a fixed array of 5 entries for open files. The first two entries should be reserved for **`ConsoleInput`** (index 0, alias **`stdin`**) and **`ConsoleOutput`** (index 1, alias **`stdout`**). Make sure to check the parameters provided by the I/O system calls properly, and cleanly abort user programs which attempt to write into unopened files or try to read from **`stdout`**, etc. Also, make sure that your OS closes any files left open when the user program exits or is aborted.

Implementation Hint 4:

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 (killed!) and all its resources (i.e. open files) must be carefully cleaned up (i.e. closed).

Make sure that your implementation takes care of this protection as much as possible! At minimum, your implementation must safely handle the test cases listed below.

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 **`Print.c`**:
should print the famous string “Hello World!” to the console

(b) Program **`Reverse.c`**:
should let the user enter some text string (e.g. “This is a test”) and then print it backwards (e.g. “tset a si sihT”)

(c) Program **`Show.c`**:
should ask the user for a file name and print the contents of that file to the console

All these “good” test programs should run fine and exit cleanly.

You should also test if your kernel is “bullet-proof”. For this purpose, create and run the following “bad” examples:

(d) Program **`MemError.c`**:
attempts to store the word 42 into the invalid memory address 1

(e) Program **`FileError.c`**:
attempts to read data from an unopened file

(f) Program **`IOError.c`**:
attempts to write a string to the standard input stream

All these “bad” test programs should be properly killed by the OS before they do any damage.

Deliverables:

a) The extended source files **`addrspace.h`**, **`addrspace.cc`** and **`exception.cc`**

b) The six test programs **`Print.c`**, **`Reverse.c`**, **`Show.c`**, **`MemError.c`**, **`FileError.c`**, and **`IOError.c`**, and six log files (**`Print.log`**, **`Reverse.log`**, **`Show.log`**, **`MemError.log`**, **`FileError.log`**, and **`IOError.log`**) that show the successful run of the tests

c) A description (in the body of your email!) that briefly outlines your implementation, i.e. status, open issues, problems solved, and decisions taken

Submission instructions:

To submit your homework, send an email with subject "EECS211 HW5" to the course instructor at doemer@uci.edu. Please include the files listed above as attachments, and put your brief description in the body of your email.

To ensure proper credit, be sure to send your email before the deadline:
March 4, 2009, 12:00pm (noon).

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