EECS 211
Advanced System Software
Winter 2011

Assignment 5

Posted: February 25, 2011
Due: March 9, 2011 at 2pm

Topic: Exception handling and system calls in Nachos

Instructions:

The goal of this assignment is to develop, implement and test basic support for user programs in the Nachos kernel. To keep things simple, we will focus only on a basic subset of exceptions and system calls that need to be supported in this assignment.

This assignment is a direct extension of the previous Assignment 4, and as such also follows Task 1 of "Nachos Assignment 2" described in the file **`doc/userprog.ps`** of the original Nachos package. Again, the instructions below assume that you read **`doc/userprog.ps`** in parallel.

Preparation: Patch the given framework

For this assignment, we will work in the **`userprog`** directory. To make the task of extending the given Nachos kernel easier, we will use a few files that the instructor has prepared for this assignment. Copy these prepared files into your Nachos installation, as follows:

```
cd userprog
cp ~doemer/eecs211/addrspace.h.W11patchedA5 ../userprog/addrspace.h
cp ~doemer/eecs211/addrspace.cc.W11patchedA5 ../userprog/addrspace.cc
cp ~doemer/eecs211/console.cc.W11patchedA5 ../machine/console.cc
cp ~doemer/eecs211/exception.cc.W11templateA5 ../userprog/exception.cc
gmake
```

The final compilation step should run through cleanly.

Next, run the generated **`nachos`** kernel with the user programs that were part of the Nachos installation, as well as the ones that you developed in Assignment 4. For example:

```
./nachos –x ../test/halt
```

and/or

```
./nachos –d X –x ../test/halt
```

You will see that only the **halt** program runs successfully. All others will need extension of the kernel.

Before you start coding, you may want to 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.

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** directory for this assignment, in particular file **exception.cc**. All other files should be left unmodified!

If you are not sure about any issue, please make use of the course message board for technical discussion!

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 extend the **switch** statement in the function **ExceptionHandler()**with one **case** for each exception type. Also, the **SyscallException** is handled by a function called **SystemCall()** that again contains another **switch** statement to handle each type of system call. All the necessary code should go into the file **exception.cc** (please start from the provided template file, see above!)

Note that, except for the **SyscallException**, most 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), then clean up any resources of the running program, and finally terminate the user program. While some situations may call for shutting down the machine immediately, you will see that killing the user program will also exit the Nachos execution (when no more processes/threads are available, the scheduler shuts down the machine).

We will limit this assignment to support only the basic system calls for file I/O. Specifically, your code should fully 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`**). We will assume a maximum of 5 open I/O streams per user program, see below.

Implementation Hint 1:

For safe and easy 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 the files **`addrspace.h`** and **`addrspace.cc`**. One instance of the synchronous console is automatically allocated with the creation of an address space for a process. Thus, it is readily present when any system calls need to use it.

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, we will use a set of dedicated memory copy functions in the kernel. Appropriate function prototypes with the following signatures are provided in the template file **`exception.cc`**:

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

In addition, it is 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 table because each process is now assigned such a space (via the **Thread->space** pointer).

To keep things simple, we maintain for each process a fixed array of 5 entries for open files. Please see the **FileTable** defined in files **addrspace.h** and **addrspace.cc** for details. An initial function **CloseAllFiles()** is also provided in file **exception.cc**.

The first three entries in the **FileTable** are reserved for **ConsoleInput** (index 0, alias **stdin**), **ConsoleOutput** (index 1, alias **stdout**), and error handling (i.e. **stdout**, a future extension, not supported now). Make sure to check the arguments passed to the 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! As bare minimum, your implementation must safely handle the test cases that we implemented in Assignment 4.

Task 2: Validate your implementation using the test programs

To test your exception handling and the implemented system calls, use the set of Nachos user programs implemented for Assignment 4 and run them on your kernel:

(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, run the “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) Extended source file **`exception.cc`**.

b) Six log files that show your test programs running on your kernel: **`HelloWorld.c`**, **`Reverse.c`**, **`ListFile.c`**, **`MemError.c`**, **`FileError.c`**, and **`IOError.c`**.

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 9, 2011, 2pm (sharp!).

--
Rainer Doemer (EH 3217, x4-9007, doemer@uci.edu)