**Project #1 – Exceptions and Simple System Calls**

**Introduction to Operating Systems** **CSE421**
**Assigned: January 26, 2005** **Due: February 27, 2005 11:59:59 PM**

The first project is designed to further your understanding of the relationship between the operating system and user programs. In this assignment, you will implement simple system call traps. In Nachos, an exception handler handles all system calls. You are to handle user program run time exceptions as well as system calls for IO processing. We give you some of the code you need; your job is to complete the system and enhance it.

***Phase 1: Understand the Code***

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 –rs 1023 -x ../test/halt'**. As before, trace what happens as the user program gets loaded, runs, and invokes a system call.

The files for this assignment are:

| | |
|---|---|
| **progtest.cc** | test routines for running user programs. |
| **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.*** | 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. In assignment four, we'll implement the Nachos file system for real on a simulated disk. |
| **translate.*** | 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. You will generalize this to allow multiple user programs to be run concurrently in a later lab. |
| **machine.*** | 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.*** | 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. |
| **synchconsole.*** | routine to synchronize lines of I/O in Nachos. Use the synchconsole class to ensure that your lines of text from your programs are not intermixed. |
| **../test/*** | C programs that will be cross-compiled to MIPS and run in Nachos |

***Phase 2: Design Considerations***

In order to fully realize how an operating system works, it is important to understand the distinction between kernel (system space) and user space. If we remember from class, each process in a system has its own local information, including program counters, registers, stack pointers, and file system handles. Although the user program has access to many of the local pieces of information, the operating system controls the access. The operating system is responsible for ensuring that any user program request to the kernel does not cause the operating system to crash. The transfer of control from the user level program to the system call occurs through the use of a "system call" or "software interrupt/trap". Before invoking the transfer from the user to the kernel, any information that needs to be transferred from the user program to the system call must be loaded into the registers of the CPU. For pass by value items, this process merely involves placing the value into the register. For pass by reference items, the value placed into the register is known as a "user space pointer". Since the user space pointer has no meaning to the kernel, we will have to translate the contents of the user space into the kernel such that we can manipulate the information. When returning information from a system call to the user space, information must be placed in the CPU registers to indicate either the success of the system call or the appropriate return value.

In this assignment we are giving you a simulated CPU that models a real CPU. In fact, the simulated CPU is the same as the real CPU (a MIPS chip), but 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.

NOTE : You should -→***NOT***← alter the code within the machine directory, only the code within the *userprog* directory.

***Phase 3: [75%] Exceptions and IO System Calls***

Implement exception handling and handle the basic system calls for file IO. (All system calls are listed in syscall.h) We have provided you 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 do the following steps :

a) Alter *exception.cc* to handle all of system exceptions as listed in *machine/machine.h* Most of the exceptions listed in this file are comprised of run time errors, from which the user program will be unable to recover. The only special cases are *no exception,* which will return control to the operating system and *syscall exception,* which will handle our user system calls. For all other exceptions, the operating system should print an error message and Halt the simulation.

b) Create a control structure that can handle the various Nachos system calls. Test your control structure by re-implementing the **void Halt()** system call. Make sure that this call operates in the same manner as we discussed during the Nachos walkthrough; it should cause the Nachos simulation to terminate immediately. Test the call's accuracy with the test user program.

c) All system calls beyond **Halt()** will require that Nachos increment the program counter before the system call returns. If this is not properly done, Nachos will execute the system call forever. Since the MIPS emulator handles look ahead program counters, as well as normal ones, you will have to emulate the program counter increment code as found in the machine directory. You will have to copy the code into your *syscall* exception handler and insert it at the proper place. For now, you will have to use the **Halt()** system call at the end of each of your user programs.

d) Implement the **int CreateFile(char *name)** system call. The createfile system call will use the Nachos Filesystem Object Instance to create a zero length file. Remember, the filename exists in user space. This means the buffer that the user space pointer points to must be *translated* from user memory space to system memory space. The createfile system call returns 0 for successful completion, -1 for an error.

e) Implement the **int CreateMailBox(char *name, int buffersize)** system call. The CreateMailBox system call will create a temporary read/write buffer created within the operating system. The name parameter will associate a unique name with each mailbox. The buffersize will indicate the number of characters that can be stored within the named buffer. The buffer should be created outside of the scope of the thread, as we will be using the mailbox in project 2 to implement interprocess communication. There will be a pre-defined limit (create a constant) for the number of MailBoxes that can be created within the kernel. For now, set this limit to 10. The CreateMailBox system call returns 0 for successful completion, -1 for an error. It is up to you to determine how the user program could cause this system call to fail.

f) Implement the **OpenFileID Open(char *name, int type)** and **int Close(OpenFileID id)** system calls. The user program can open two types of "files", actual files created with CreateFile and MailBoxes created with CreateMailBox. Each process will allocate a fixed size file descriptor table. For now, set this size to be 20 file descriptors. The first two file descriptors, 0 and 1, will be reserved for console input and console output respectively. The open file system call will be responsible for translating the user space buffers when necessary and allocating the appropriate kernel constructs. For the case of actual files, you will use the filesystem objects provided to you in the filesystem directory. (NOTE: We are using the FILESYSTEM_STUB code) For the case of the MailBox, you will be responsible for creating the correct system level structure or classes. Both calls will use the Nachos Filesystem Object Instance to open and close files. The **Open** system call returns the file descriptor id (OpenFileID == an integer number), or –1 if the call fails. Open can fail for several reasons, such as trying to open a file or mailbox that does not exist or if there is not enough room in the file descriptor table. The *type* parameter will be set to 0 for a standard file and 1 for a mailbox. If the type parameter is set to any other value, the system call should fail. The close system call will take a file descriptor as the parameter. The system call will return –1 on failure and 0 on success.

g) Implement the **int Read(char *buffer, int charcount, OpenFileID id)** and **int Write(char *buffer, int charcount, OpenFileID id)** system calls. These system calls respectively read and write to a file descriptor ID. Remember, you must translate the character buffers appropriately and you must differentiate between console IO (OpenFileID 0 and 1) and File or Mailbox IO (any other valid OpenFileID). The read and write interaction will work as follows:

For console read and write, you will use the SynchConsole class, instantiated through the gSynchConsole global variable. You will use the default SynchConsole behaviors for read and write, however you will be responsible for returning the correct types of values to the user. Read and write to Console will return the *actual* number of characters read or written, not the requested number of characters. In the case of read or write failure to console, the return value should be –1. If an end of file is reached for a read operation from the console, the return value should be –2. End of file from the console is returned when the user types in Control-A. Read and write for console will use ASCII data for input and output. (Remember, ASCII data is NULL (\0) terminated)

For file read and write, you will use the supplied classes in file system. You will use the default filesystem behaviors, however, you will return the same type of return values as for synchconsole. Both read and write will return the *actual* number of characters read and written. Both system calls will return –1 for failure and read will return –2 if the end of file is reached. Read and write for files will use binary data for input and output.

The mailbox construct must maintain a pointer into the mailbox buffer. Both mailbox read and write will start from the current position of the pointer and will update this pointer by the number of characters read or written. Both read and write will return the *actual* number of characters read and written. Both system calls will return –1 for failure and read will return –2 if the end of buffer is reached. Read and write for mailbox will use binary data for input and output.

h) Implement the **int Seek(int pos, OpenFileID id)** system call. Seek will move the file cursor to a specified location. The parameter *pos* will be the absolute character position within a file. If *pos*

is a –1, the position will be moved to the end of file. The system call will return the actual file position upon success, -1 if the call fails. Seeks on console IO will fail. Seeks on the mailbox will move the mailbox pointer to the proper position.

i) Implement a **createfile** user program to test the createfile system call. You are not going to pass command line arguments to the call, so you will have to either use a fixed filename, or prompt the user for one when you have console IO working.

j) Implement a **help** user program. All help does is it prints a list to standard output of all the user programs you are going to create or have created in the test directory. Help should list each program and a brief 1 line description of each program. (NOTE : This is nothing fancy, just calls using the Write() system call to standard output.)

k) Implement an **echo** user program. For each line of input from the console, the line gets echoed back to the console.

l) Implement a **cat** user program. Ask for a filename, and display the contents of the file to the console.

m) Implement a **copy** user program. Ask for a source and destination filename and copy the file.

n) Implement an **outfile** user program. This program asks for a destination file, takes input from the console, and writes it to the destination.

o) Implement a **testmailbox** user program. This program should demonstrate whether your mailbox solution works correctly.

p) Implement any other tests you feel are necessary to ensure the correctness of your solution. BIG HINT: Implement tests to cover some of the things we changed that are not tested by parts i through o.

**NOTE:** A large portion of your grade will depend not only on the correctness of your implementation but how accurately your code conforms to the system call specifications presented in this document. As we are building a robust operating system, the user should not be able to perform any system call that will crash Nachos. The worse case scenario is that the user program might generate a Nachos runtime exception, which you will be handling within part a.

***Phase 4: [10%] Implement a new system call***

Implement a new system call

```
/* Run the executable, stored in the Nachos file "argv[0]", with
 * parameters stored in argv[1..argc-1] and return the
 * address space identifier
 */
SpaceId ExecV(int argc, char* argv[]);
```

Excev will have code 12. You will have to update the **Start.s** file in the test directory and the **syscall.h** file in the **userprog** directory. Recompile and test it with a sample program **cmdLine**. In this project we will only print out a message containing argc and argv from inside the exception handler. In the next project we will implement the code for exceV.

***Phase 5: [15%] Documentation***

This includes internal documentation (comments) and a ***BRIEF, BUT COMPLETE*** external document (read as: paper) describing what you did to the code and why you made your choices. **DO NOT PARAPHRASE THIS LAB DESCRIPTION AND DO NOT TURN IN A PRINTOUT OF YOUR CODE AS THE EXTERNAL DOCUMENTATION**

**Deliverables and grading**

When you complete your project, remove all executables and object files. If you want me to read a message with your code, create a README.NOW file and place it in the nachos *code* directory. Tar and compress the code, and submit the file using the online submission system. It is important that you follow the design guidelines presented for the system calls. I will be running my own shells and test programs against your code to determine how accurately you designed your lab, and how robust your answers are. Grading for the implementation portion will depend on how robust and how accurate your solution is. Remember, the user should not be able to do anything to corrupt the system, and system calls should trap as many error conditions as possible.