Project #1 – Implementing File-System API

Introduction to Operating Systems **CSE421/521**
Assigned: September 7, 2005 **Due: October 4, 2005 11:59:59 PM**

1. Objectives: To enhance Nachos application interface by adding a file system application programming interface (API). This API will include a set of system calls that will allow programmatic calls to Nachos file system from a C application program.

2. Project Description:
Read and understand components and architecture of Nachos system. See detailed documentation available in Nachos Roadmap. Section 5 of roadmap has details about Nachos implementation of file system. The code for the file systems can be found in the directory nachos-3.4/code/filesys. We have given below a class diagram that shows the various classes and the relationship among them.

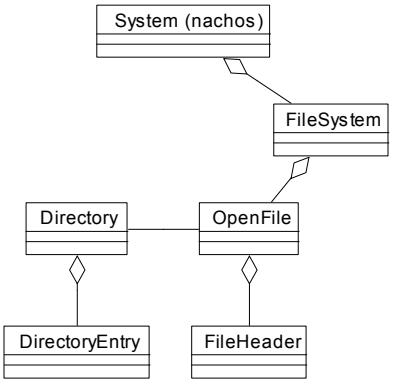

Figure 1: Nachos File System (nachos-3.4/code/filesys/)

Figure 1 shows nachos system (code/threads/system.h, system.cc) instantiating an object of FileSystem class. FileSystem maintains all openfiles (OpenFile objects). A directory object has an Openfile object. It also has a table of objects of DirectoryEntry. All the methods such as read and write defined in the classes are accessible at kernel level and not at application level. The current Nachos file system is implemented by directly making the corresponding calls to the UNIX file system. The API you build will facilitate operations such as creation, reading, writing, *seeking into* and deletion of different *types of files* from an application program. To accomplish this you will make use of the classes nachos provides.

2.1 Understand the Code

The first step is to read and understand the existing code. After you expand the tar distribution of nachos, examine the *code* directory. Then "gmake all" from the *code* directory to carry out a preliminary compile and link of code in all the directories. Makes sure compilation finishes without errors. (Get help from TAs in case you have errors.) There is a trivial test provided with the distribution *code/test* directory, 'halt'; all halt does is to turn around and ask the operating system to shut the machine down. Change directory into *userprog* directory. Run the command **'nachos –rs 1023 -x ../test/halt'**. This should halt the simulated MIPS machine and type out statistics for that particular run.

Examine the following files to understand Nachos:

code/userprog

syscall.h: This provides the code and function prototype for system calls that user level test programs can invoke.
exception.cc: The handler for system calls and other user-level exceptions, such as page faults. Currently only the 'halt' system call is supported.

code/machine

machine.* emulates 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.

code/threads
thread.* implements thread (unit of work) for nachos. Methods to control operation of threads. All the thread specific (**local**) properties are define here.
system.* implements all the system facilities. The common (**global**) components of the machine are declared, instantiated here. For example, FileSystem object, currentThread object (pointer to current thread), etc.

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.

code/test/* C programs that will be cross-compiled to MIPS and run in Nachos; **start.s** has all the assembly language stubs for the system calls.

code/filesys
openfile.h a stub defining the Nachos file system routines.

Read the **Makefile** in the various directories. In general, while working on Nachos projects you will add classes (your code) to **userprog** and C programs to **test** directory. You may modify existing classes in other directories.

The following are the steps in adding and testing a **new system call** to Nachos:

1. In **userprog/syscall.h** file, define a code for the system call, and add the C function prototype corresponding to the system call. (Remember it is C language interface).
2. In **start.s** add a "macro" stub corresponding to the syscall. This is a set of assembly language instructions and directives that will help compiler substitute the C call with this stub code. See start.s for examples.
3. **ExceptonHandler** function in **exception.cc** provides the entry point into kernel for handling the system call. Add the code for exception handler for the new syscall to exception.cc.
4. If you added any new supporting classes in **userprog**, include them **Makefile.common** definitions for USERPROG_H, USERPROG_C, and USERPROG_O.
5. Add a C test program (say, **trial1.c**) that uses this system and change Makefile in test directory to compile and link it. See examples in the current Makefile in the test directory. "gmake"
6. "gmake" in userprog and test the new system call by executing: **nachos –rs 5678 –x ../test/trial1**

2.2 Design of File Syscall API

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.

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" (in **bin** directory). Floating-point operations are not supported.

Implement exception handling and handle the basic system calls for file system. Figure 2 depicts the role of file syscall API.

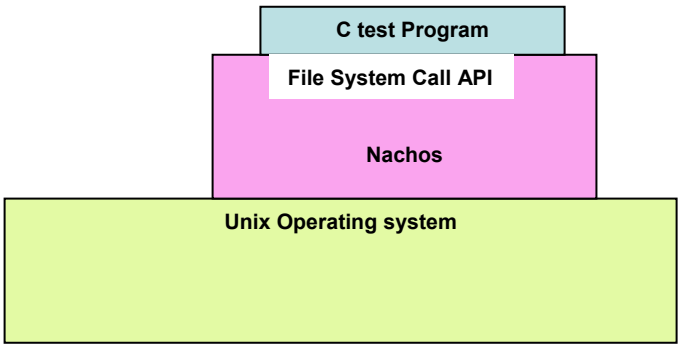

Figure 2: File Sys Call Interface between a user program and Nachos kernel

Signatures of the file system call API are given below. Implement the signatures exactly as given here. OpenFileID is integer type.

File System Call API
int CreatFile (char *name) OpenFileID Open (char *name, int type) int Read (OpenFileID fd, char *buffer, int nbytes int Write (OpenFileID fd, char *buffer, int nbytes) int Seek (OpenFileID fd, int pos) int Close (OpenFileID fd) int DeleteFile (char *name)

You will need to do the following steps:

a) Implement the **int CreateFile(char *name)** and **int DeleteFile(char * name)** system calls. 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 both system call return 0 for successful completion, -1 for an error.

b) Implement the **OpenFileID Open(char *name, int type)** and **int Close(OpenFileID id)** system calls. The user program can open three types of "files": type 1 is read only (RO), type 2 is read and write (RW), and type 3 is read and execute (RX). If the type parameter is set to any other value, the system call should fail. Each process will allocate a fixed size file descriptor table. For now, set this size to be 8 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). 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 that does not exist or if there is not enough room in the file descriptor table. The close system call will take a file descriptor as the parameter. The system call will return –1 on failure and 0 on success.

c) Implement the **int Read(OpenFileID fd, char *buffer, int nbytes)** and **int Write(OpenFileID fd, char *buffer, int nbytes)** system calls. These system calls respectively read and write to a file defined by file descriptor **fd**. Remember, you must translate the character buffers appropriately and you must differentiate between console IO (OpenFileID 0 and 1) and different types of files. The read and write interaction will work as follows:

For console read and write, you will use the SynchConsole class (code/threads directory), instantiated through the gSynchConsole global variable (see threads/system.h,.cc). 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 number of characters read or written. 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). Also for file types RO and RW use ASCII reads, and for RX use BINARY read and write.

d) Implement the **int Seek(OpenFileID fd, int pos)** 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 cursor 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. Seek differs from other calls above since you will have to implement the equivalent Nachos code yourself in the appropriate file in ***filesys*** directory.

e) Implement a new system call "**Delete" int DeleteFile (char *name)**. **Delete** will have code 13. 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 **delete**.

f) Implement a **createFile** C 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.

g) 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.

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

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

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

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

l) Implement a **testSeek** user program. This program should demonstrate whether your seek solution works correctly.

m) 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 e through k.

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. Make sure there is a default case in the exception handler that will print out a message and halt the system in the case.

3. 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.

4. 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. We 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.