CIS 657 Programming Assignment 3

Submit two files:

- Create your Assignment 3 document (doc or pdf), follow the instruction and then submit it to the **Turnitin link** on the blackboard (under Assignments).
  - You can submit multiple times before the due.
  - After the due, you cannot resubmit newer version.
- Do a "make clean" in code/build.linux, then compress your Nachos folder on the VM, download the compressed file into your local machine and name the compressed file **Ass3_yourNetId.zip (Note: Zip only!!)**. Submit the zipped file to the **Assignment Code Submission link** of Blackboard.
  - You need to submit the entire nachos directory
    - We will build and run your nachos, not individual files
  - Use naming convention
  - When we uncompress your zip file, we want to see your nachos like

```
\Ass3_netid
        \code
                \threads
                \machine
                ...
```

- Do not change compiler option in Makefile
- You have to make sure your submission is correctly made
  - **If you don't have a confirm email, you should check again.**
- **You should raise an appeal within a week after grade is posted.**

Due: December 2 (Monday, end of the day)

Late submission: you will have 2^d penalty of your late days. If you submit your files on different days, later day will be considered.

Follow the Lab1 instruction and create a new fresh Nachos folder. You need multitasking and virtual memory, use your assignment 2 implementation.

Overview

NachOS uses the "stub" filesystem implementation, which simply translates NachOS file system calls to UNIX file system calls. You have been using this file system implementation for the previous assignments. NachOS also comes with a very basic file system implementation that uses the NachOS

simulated disk. For this assignment, your task is to improve on this basic implementation.

In this programming assignment, you have to implement the system calls of NachOS. In order to protect the users and the kernel, it is important to consider some basic security issues. If a user is only allowed to use predefined system calls, he can only harm the kernel by passing invalid arguments or extremely oversized parameters. Therefore, do never forget to check the parameters before using them and limit their size. A description of what the calls mentioned below should do and which parameters are passed and/or returned should be found in the corresponding interface in code/userprog/syscall.h.

Getting Started:

The files to focus on are:

- filesys.h, filesys.cc – top-level interface to the file system.
- directory.h, directory.cc – translates file names to disk file headers; the directory data structure is stored as a file.
- filehdr.h, filehdr.cc – manages the data structure representing the layout of a file's data on disk. This is the NachOS equivalent of a UNIX i-node.
- openfile.h, openfile.cc – translates file reads and writes to disk sector reads and writes.
- synchdisk.h, synchdisk.cc – provides synchronous access to the asynchronous physical disk, so that threads block until their requests have completed.
- disk.h, disk.cc – emulates a physical disk, by sending requests to read and write disk blocks to a UNIX file and then generating an interrupt after some period of time. The details of how to make read and write requests varies tremendously from disk device to disk device; in practice, you would want to hide these details behind something like the abstraction provided by this module.

Task 1 (60 %) Implement Complete Basic NachOS FileSystem:

1. Nachos file system has a UNIX-like interface, so you may also wish to read the UNIX man pages for creat, open, close, read, write, lseek, and unlink (e.g., type "man creat"). Nachos file system has calls that are similar (but not identical) to these calls; the file system translates these calls into physical disk operations. Create (like UNIX creat), Open (open), and Remove (unlink) are defined on the FileSystem object, since they involve manipulating file names and directories. FileSystem::Open returns a pointer to an OpenFile object, which is used

for direct file operations such as Seek (lseek), Read (read), Write (write). An open file is “closed” by deleting the OpenFile object.

- int Create(char *name, int protection)
  - Creates a new NachOS file named “name”, but does not open it.
  - Create an empty file
  - Protection attribute: rwx 3-bit combination
    for example, r - - (4), - w – (2), r w – (6), r w x (7), …
- int Remove(char *name)
  - Removes a NachOS file named “name”.
  - When a file is removed, processes (NachOS threads) that have already opened that file should be able to continue to read and write the file until they close the file. However, new attempts to open the file after it has been removed should fail. Once a removed file is no longer open by for any process, the filesystem should actually remove the file, reclaiming all of the disk space used by that file, including space used by its header.
- OpenFileId Open(char *name, int mode)
  - Opens the file called “name” and returns an ID to be used as a file descriptor for the file in subsequent Read and Write calls. Mode specifies the requested access mode to this file (RO=1, RW=2, APPEND=3).
  - If protection permission is not valid for the given mode, error occurs
  - Do not use pointer for OpenFileId (int)
  - Per-process open file table (per-AddrSpace)
  - System-wide open file table
- int Write(char *buffer, int size, OpenFileId id)
  - Writes “size” bytes from “buffer” to the open file.
  - Makes Write syscall to perform console I/O
  - Returns # of bytes successfully written
- int Read(char *buffer, int size, OpenFileId id)
  - Reads “size” bytes from the open file into “buffer”.
  - Makes Read syscall to perform console I/O
  - Returns the number of bytes actually read, which does not always have to equal the number of bytes requested.
- int Seek(int position, OpenFileId id)
  - Set the current position within the open file called “id”
  - Each time a file is opened, NachOS returns an OpenFileId to the calling process. There should be a separate file (seek) position associated with each such OpenFileId. The Read and Write system calls modify this position implicitly, while the Seek system call lets a process explicitly change an open file's seek position so it can read or write any portion of the file.
- int Close(OpenFileId id)
  - Releases a file after it is not needed anymore.
  - Clean up all data structures representing the open file
- All system calls return -1 when they are failed.

2. Many of the data structures in our file system are stored both in memory and on disk. To provide some uniformity, all these data structures have a “FetchFrom” procedure that reads the data off disk and into memory, and a “WriteBack” procedure that stores the data back to disk. Note that the in-memory and on-disk representations do not have to be identical.

3. Complete the NachOS basic file system by **adding synchronization to allow multiple threads to use file system concurrently.** Currently, the file system code assumes that it is accessed by a single NachOS thread at a time. In addition to ensuring that internal data structures are not corrupted, your file system must observe the following constraints (these are the same as in UNIX):
   - The same file may be read/written by more than one NachOS thread concurrently. Each thread separately opens the file, giving it its own private seek position within the file. Thus, two threads can both sequentially read through the same file without interfering with one another.
     - During lecture, we discussed about this
   - **All file system operations must be atomic and serializable.** For example, if one thread is in the middle of a file write, a thread concurrently reading the file will see either all of the change or none of it. Further, if the OpenFile::Write operation finishes before the call to OpenFile::Read is started, the Read must reflect the modified version of the file.
     - **Directory operations and read/write operations execute atomically too**
   - When a file is deleted, threads with the file already open may continue to read and write the file until they close the file. Deleting a file (FileSystem::Remove) must prevent further opens on that file, but the disk blocks for the file cannot be reclaimed until the file has been closed by all threads that currently have the file open.
   - *Hint: to do this part, the file system needs to maintain tables of open files: per-process and system-wide.*

4. Modify the file system to allow the maximum size of a file to be as large as the disk (128Kbytes). In the basic file system, each file is limited to a file size of just under 4Kbytes. Each file has a header (class FileHeader) that is a table of direct pointers to the disk blocks for that file. Since the header is stored in one disk sector, the maximum size of a file is limited by the number of pointers that will fit in one disk sector. Increasing the limit to 128KBytes will probably but not necessarily require you to implement doubly indirect blocks.

5. Implement dynamically extensible files. In the basic file system, the file size is specified when the file is created. One advantage of this is that the FileHeader data structure, once created, never changes. In UNIX and most other file systems, a file is initially created with size 0 and is then expanded every time a write is made off the end of the file. Modify the file system to allow this; **as one test case, allow the directory file to expand beyond its current limit of files. In doing this part, be careful that concurrent accesses to the file header remain properly synchronized.**

6. ***Once NachOS is no longer using its stub file system, the behavior of the system should change.***

   // Filesystem-related flags: defined in main.cc
   // -f forces the Nachos disk to be formatted
   // -cp copies a file from UNIX to Nachos
   // -p prints a Nachos file to stdout
   // -r removes a Nachos file from the file system
   // -l lists the contents of the Nachos directory
   // -D prints the contents of the entire file system
   For example, you will no longer be able to simply run: ./nachos -x ../test/halt
   need to find user programs in the NachOS FileSystem

   a) **./nachos –f : format the NachOS disk and initialize NachOS FileSystem (empty) on it (you can use your own flag)**
      - **You must format the NachOS disk before storing any file on it for the first time**

   b) **./nachos –cp ../test/user_prog1 prog1: load user program executable in the UNIX file system to NachOS File System**
      - **Protection (permission) bit – x bit - of executable files must be 1**
      - **Since user application programs are compiled in UNIX directory (~/code/test/), you may want to load several application programs or data files into the NachOS file system over and over when recompiling those application programs. You may use different flag to load files.**

   c) **Then ./nachos –x prog1**

7. ***Remember to test your File System Calls and File System Operation correctly. You must write small test programs (application programs) that demonstrate that your project works correctly.***

You need to present sufficient test cases to show the correctness of your File System.

- Ensure that all requirements work properly with the basic file system.
- To test your implementation, write program(s), which uses the implemented system calls. You need to have some logging mechanism in your output to show that your implementation is correct. This is very crucial. Please provide your test cases in the report document containing information about what you have made, the test programs you have written, and etc.
  - Exercise the file system calls with both legal and illegal parameter values.

Task 2 (40 %) Write a shell application using the sample in test/shell.c: The shell -- user program that keeps reading commands from the console and executing them. Each command is a file name of another user program. The shell is running the command in a child process using the **Exec** system call that you implemented in the assignment 2. In order to make the parent (shell) program to wait for the child process to complete, you need to implement **Join** system call. After the child process completes, the shell will wait for user command.

1. Write and Read syscalls in shell.c use
   OpenFileId input = ConsoleInput;
   OpenFileId output = ConsoleOutput;
   Write(prompt, 2, output);
   Read(&buffer[i], 1, input)
2. Modify **Exec** system call to run user programs in your NachOS file system.
   a. Takes the filename argument whose file is loaded in the file system, not from a host system (UNIX) directory.
   b. Returns SpaceId
3. int **Join** (SpaceId childId) system call:
   a. This is called by a parent process to wait for the termination of the child process whose SpaceId is given by the childId argument. If the child process is still active, then Join blocks until the child exits. When the child has exited, Join returns the child's exit status (Exit system call) to the parent. To simplify the implementation, each child may be joined on at most once. Nachos Join is basically equivalent to the Unix wait system call.
   b. childID must be non-zero.
   c. You may need to keep a list of all child processes of each process.
      1) Need to handle the case when a child process calls Exit syscall before the parent calls Join syscall.
      2) Join/Exit relationship!!
4. UNIX-like utility programs (user programs in ~/test)– used as commands in the shell: You may need **additional system calls to perform the following utility programs (Extend your kernel to support your own shell features)**. You may define your own shell command syntax and semantics

(but close to UNIX system utility) – need to handle string arguments to Exec system call
   a. ls
   b. cd
   c. pwd
   d. mkdir
   e. cp
   f. mv
   g. rm
   h. rmdir
   i. chmod

5. Multiple commands separated by “;”, a semicolon:
   a. The shell executes all of the commands concurrently and waits for them all to complete before accepting the next command.
6. Sample Output:

```
./nachos –x myShell
>> prog1
display something from prog1
>> pwd
/usr/app
```

Bonus (10 pts)

1. Implement interprocess communication using pipes. This will redirect the output of one process to the input of another. An example of a Unix pipe would be "program1 | program2" from the command line of a shell. In this example, program1 would normally output text to the standard output, but the pipe will redirect this output to the input of program2. program2 would normally take its input from a file. This problem will require some imagination since files do not exist, but should not be too difficult to implement. In your kernel, program1 is the first child process created by Exec and program2 is the second child process. The standard output of program1 should be bound to the pipe and the standard input of program2 should be bounded to the output of the preceding process, program1, from the pipeline.
   a. Sequence of pipes, each with one reader and one writer.
   b. Think about Exec/Join and Pipe Redirection
   c. Pipes are implemented as producer/consumer bounded buffers with a maximum buffer size of N bytes. If a process writes to a pipe that is full, the Write call blocks until the pipe has drained sufficiently to allow the write to continue. If a process reads from a pipe that is empty, the Read call blocks until the sending process exits or writes data into the pipe. If a process at either end of a pipe exits, the pipe is said to be broken: reads from a broken pipe drain the pipe and then stop returning data, and writes to a broken pipe silently discard the data written.

2. More shell utilities
   a. cat (You can think of the following cases)
      1) read a file and write it to standard output (OpenFileId 1)
      2) read standard input (OpenFileId 1) and write it to a file
      3) read standard input and write it to standard output
   b. link
   c. unlink

Your document must include the followings:

- Cover Page
- Disclosure Form
- How much time did you spend to do:
  - Analyze the problem, determine specifications, and create your design
  - Implement the design
    - write the program
  - Test/debug the program
    - Find/fix errors
    - Create test cases, and try them out
- List of your files with directory name that you modified or created for this assignment
- Design/your solution for each requirement
  - **We are looking for your own answers**
- Implementation of your solution (Code snippet with good comments)
  - Do not include Nachos original code
  - We want to see your own code/modification
  - Should be text (no image/photo of code)
- Testing
  - How to run your tests
  - What each test case is testing (you need to prove your implementation of each requirement in the following output snapshots)
  - **You must test all requirements of the basic NachOS file system and shell application.**
  - You need to verify that all system calls are correctly working.
- Output Snapshots

Testing:

We will build and run your Nachos on the VM. You must test your program on the VM and also should provide proper test scenario in your document.

Grading:

- Syntax/Runtime/Logic Errors with proper Makefile [-50, -15]
- Data structure design/class definition/declaration respectively [-40, -10]
- Quality of your report/comments [-20, -5]
- Quality of user programs [-30, -5]
- Satisfy all implementation requirements [-100, -5]
- Garbage collection (handle all objects at the end) [-10, -5]
- Input/output design (meaningful messages to prove your implementation) [-30, -10]
- Output(by Students)/Test(by TAs) [-50]
- **If a student turns in one submission (either code or document), he/she could get up to 50 pts.**