Project 2: Multiprogramming
CSE-150-01 **Points: 100**

Overview

The second Nachos project is to implement system calls. As in the first project, we give you some of the code that you will need, and your task is to complete the system and enhance it. Up to now, all the code you have written for Nachos has been part of the operating system kernel. In a real operating system, the kernel not only uses its procedures internally, but allows user-level programs to access some of its routines via *system calls*. The goal of this project is to enable user-level programs to invoke Nachos routines that you implement in the Nachos kernel.

Background

The changes you make to Nachos will be in these two files in the `userprog` directory:

- `UserKernel.java` – a multiprogramming kernel.
- `UserProcess.java` – a user process; manages the address space and loads a program into virtual memory.

You will also want to familiarize yourself with the other classes in `userprog`:

- `UThread.java` – a thread capable of executing user MIPS code.
- `SynchConsole.java` – a synchronized console; makes it possible to share the machine's serial console among multiple threads.

as well as a couple of classes in the `machine` directory:

- `Processor` simulates a MIPS processor.
- `FileSystem` is a file system interface. To access files, use the `FileSystem` returned by `Machine.stubFileSystem()`. This file system accesses files in the test directory.

Nachos emulates user programs executing on a real CPU (a MIPS R3000 chip). By emulating execution, Nachos has complete control over how many instructions are executed at a time, how the address translation works, and how interrupts and exceptions (including system calls) are handled. The emulator can run normal programs compiled from `C` to the MIPS instruction set. The only caveat is that floating point operations are not supported.

Nachos initially is only able to run a single user-level MIPS program at a time and fully supports one system call: `halt`. All `halt` does is ask the operating system to shut the machine down. This test program is found in `test/halt.c` and represents the simplest supported MIPS program.

Design Document and Review

You must make sure that you have a working design before you attempt to implement the task listed below (see **Project 2 Initial Design Document** assignment on CatCourses for more details). To make sure you have a working plan, you will be meet with your TA to review your design. These meetings will be held during your lab session via Zoom around ten days before the due date for the final code and should take around 15 - 20 minutes. In addition, you will turn in your final design document reflecting the actual implementation the day when your final code is due (might not change from the design review document if you design well).

Task (100%)

Implement the file system calls `creat`, `open`, `read`, `write`, `close`, and `unlink`. Their semantics and specifications are documented in `test/syscall.h`, and the calling conventions are documented in the comments to `UserProcess.handleSyscall`. You will see the code for `halt` and skeleton code for `exit` in `UserProcess.java`. Implement the other system calls following the same pattern. Note that you are *not* implementing a file system. Rather, you are simply giving user processes the ability to access a file system that Nachos already implements.

- Nachos already provides the assembly code necessary for user-level programs to invoke system calls (see `test/start.s`; the `SYSCALLSTUB` macro generates assembly code for each syscall).
- When implementing the system calls, you will need to "bullet-proof" the Nachos kernel from user program errors. There should be nothing a user program can do to crash the operating system (with the exception of explicitly invoking the `halt` syscall). In other words, you must be sure that user programs do not pass bad arguments to the kernel (e.g., a NULL pointer value of 0x0, or an invalid address) that cause the kernel to crash or corrupt its internal state or that of other processes.
- To handle large `read`/`write` calls, you should use a page-sized buffer to pass data between the file and user memory.
- Since the memory addresses passed as arguments to the system calls are virtual addresses, you need to use `UserProcess.readVirtualMemory` and `UserProcess.writeVirtualMemory` to transfer data between the user process and the kernel.
- User processes store filenames and other string arguments as null-terminated strings in their virtual address space. The maximum length for strings passed as arguments to system calls is 256 bytes (not including the terminating null).
- System calls should return the appropriate value as documented in `test/syscall.h`. When a system call needs to indicate an error condition to the user, it should return -1. In particular, it should not assert or otherwise thrown an exception.
- When any process is started, its file descriptors 0 and 1 must refer to standard input and standard output. Use `UserKernel.console.openForReading()` and `UserKernel.console.openForWriting()` to implement these semantics. A user process is allowed to close these descriptors, just like descriptors returned by `open`.
- A stub file system interface to the UNIX file system is already provided for you, and the interface is implemented by the class `machine/FileSystem.java`. You can access the stub filesystem through the static field `ThreadedKernel.fileSystem`. (Note that since `UserKernel` extends `ThreadedKernel`, you can still access this field.) This filesystem is capable of accessing the `test` directory in Nachos, which is going to be useful when you implement the `exec` system call described below. You do not need to implement any file system functionality, but you should examine carefully the specifications for `FileSystem` and `StubFileSystem` to determine what functionality you need to implement, and what is handled by the file system.
- Do not implement any kind of file locking, the file system is responsible for it. If `ThreadedKernel.fileSystem.open()` returns a non-null `OpenFile`, then the user process is allowed to access the given file; otherwise, you should return an error. Likewise, you do not need to worry about the details of what happens if multiple processes attempt to access the same file at once; the stub filesystem handles these details for you.
- Each file that a process opens should have a unique *file descriptor* associated with it (see `syscall.h` for details). The file descriptor should be a non-negative integer that is simply used to index into a table of currently open files by that process. Your implementation should have a file table size of 16, supporting up to 16 concurrently open files per process. Note that a given file descriptor can be reused if the file associated with it is closed, and that different processes can use the same file descriptor value to refer to different files (e.g., in every process file descriptor 0 refers to stdin).

To implement this task, you need to declare an array of OpenFiles for the opened files in the process, i.e., "*protected OpenFile[] myFileSlots*;".
And initialize it in the UserProcess() function as follows,
"*// Project 2 Task 1: Initialize OpenFiles array*
myFileSlots = new OpenFile[16];

```
// Project 2 Task 1: Initialize stdin/stdout slots in OpenFiles array
// File descriptor 0 refers to keyboard input (UNIX stdin)
myFileSlots[0] = UserKernel.console.openForReading();
// File descriptor 1 refers to display output (UNIX stdout)
myFileSlots[1] = UserKernel.console.openForWriting();"
```

In summary, you need to implement 7 functions, i.e., private int handleCreate(int myAddr), private int handleOpen(int myAddr), private int handleRead(int slotNum, int vaddr, int numBytes), private int handleWrite(int slotNum, int vaddr, int numBytes), private int handleClose(int slotNum), private int handleUnlink(int myAddr), and public int handleSyscall(int syscall, int a0, int a1, int a2, int a3).

Code Submission

As before, you may use the `scp` command to upload your code to the course server `klwin00.eng.ucmerced.edu`. You will be submitting your entire `nachos` directory and its contents to the server. Note that you only need to upload the (uncompiled) source code to the server (it will be compiled when autograded so `.class` files need not be submitted). If you have multiple submissions, clearly place your final code in an obviously named directory, like `P2Final`, so the TA knows what to grade. It is your team's responsibility to maintain an organized directory structure in the server and make sure there are no accidental erasures of prior versions of code which may be useful.

Autograder Testing

At any time, after copying your nachos directory to the server, you may run the command `test-subm proj2-test` from outside the same directory as the submitted nachos directory to test your code with a subset of the test-cases that your code will be put through during final testing. For example, if your nachos directory is placed inside the folder `P2`, you may navigate to (or `cd` into) `P2` and run the `test-subm` command. It will compile the files and run the appropriate tests while giving you the results of each test. We will run the autograder using the same command, `test-subm` (but with a larger set of test-cases), so make sure it works on the server.

For test cases of Project 2, you can find all the test cases on CatCourse/Files/Projects/Project/Project 2 Test Cases.pdf. You just need to focus on the testID 0, 2, 3, 5, and 10 in Filesystem Syscall Tests but ignore the reaming test cases. If you pass the above 5 cases, it should show as follows,

```
*******************************************************************
Running tests for assignment proj2-test, logging to standard output.
make[1]: Entering directory `/home/F20/2L/f20-2l-g1/test/2P_notask2'
make[1]: Nothing to be done for `studentclasses_helper'.
make[1]: Leaving directory `/home/F20/2L/f20-2l-g1/test/2P_notask2'
starting test grade-exec-02 8
finished test grade-exec-02
starting test grade-exec-11 7
finished test grade-exec-11
starting test grade-file-00 6
finished test grade-file-00
starting test grade-file-02 5
finished test grade-file-02
starting test grade-file-03 4
```

```
finished test grade-file-03
starting test grade-file-05 3
finished test grade-file-05
starting test grade-file-06 2
finished test grade-file-06
starting test grade-file-10 1
finished test grade-file-10
starting test lottery-inherit-01 0
finished test lottery-inherit-01
-------------------------------------------------------------

Running test grade-exec-02...
tests your syscall join to a child

Failure while running test grade-exec-02
Output up to 8000 bytes:
Picked up _JAVA_OPTIONS: -Xms128m -Xmx512m
nachos 5.0j initializing... config interrupt timer processor console user-check grader console

nachos.machine.AssertionFailureError: Unknown system call! Syscall: 2
at nachos.machine.Lib.assertTrue(Lib.java:89)
at nachos.machine.Lib.assertNotReached(Lib.java:106)
at nachos.userprog.UserProcess.handleSyscall(UserProcess.java:631)
at nachos.userprog.UserProcess.handleException(UserProcess.java:649)
at nachos.userprog.UserKernel.exceptionHandler(UserKernel.java:82)
at nachos.userprog.UserKernel$1.run(UserKernel.java:28)
at nachos.machine.Processor$MipsException.handle(Processor.java:603)
at nachos.machine.Processor.run(Processor.java:101)
at nachos.userprog.UThread.runProgram(UThread.java:31)
at nachos.userprog.UThread.access$000(UThread.java:11)
at nachos.userprog.UThread$1.run(UThread.java:20)
at nachos.threads.KThread.runThread(KThread.java:160)
at nachos.threads.KThread.access$000(KThread.java:30)
at nachos.threads.KThread$1.run(KThread.java:149)
at nachos.machine.TCB.threadroot(TCB.java:235)
at nachos.machine.TCB.access$100(TCB.java:25)
at nachos.machine.TCB$1.run(TCB.java:93)
at java.lang.Thread.run(Thread.java:619)

-------------------------------------------------------------

Running test grade-exec-11...
writeVM small valid range, make sure wrote right data

Failure while running test grade-exec-11
```

```
Output up to 8000 bytes:
Picked up _JAVA_OPTIONS: -Xms128m -Xmx512m
nachos 5.0j initializing... config interrupt timer processor console user-check grader console

nachos.machine.AssertionFailureError: Unknown system call! Syscall: 2
at nachos.machine.Lib.assertTrue(Lib.java:89)
at nachos.machine.Lib.assertNotReached(Lib.java:106)
at nachos.userprog.UserProcess.handleSyscall(UserProcess.java:631)
at nachos.userprog.UserProcess.handleException(UserProcess.java:649)
at nachos.userprog.UserKernel.exceptionHandler(UserKernel.java:82)
at nachos.userprog.UserKernel$1.run(UserKernel.java:28)
at nachos.machine.Processor$MipsException.handle(Processor.java:603)
at nachos.machine.Processor.run(Processor.java:101)
at nachos.userprog.UThread.runProgram(UThread.java:31)
at nachos.userprog.UThread.access$000(UThread.java:11)
at nachos.userprog.UThread$1.run(UThread.java:20)
at nachos.threads.KThread.runThread(KThread.java:160)
at nachos.threads.KThread.access$000(KThread.java:30)
at nachos.threads.KThread$1.run(KThread.java:149)
at nachos.machine.TCB.threadroot(TCB.java:235)
at nachos.machine.TCB.access$100(TCB.java:25)
at nachos.machine.TCB$1.run(TCB.java:93)
at java.lang.Thread.run(Thread.java:619)
```

Running test grade-file-00...
Creates a file, checks syscall creat works

Test grade-file-00 succeeds.

Running test grade-file-02...
tests if your syscall close actually closes the file

Test grade-file-02 succeeds.

Running test grade-file-03...
tests if your syscall open fails gracefully when stubFileSystem's openfile limit's exceeded

Test grade-file-03 succeeds.

Running test grade-file-05...

tests your syscall read by reading some bytes from a file

Test grade-file-05 succeeds.

Running test grade-file-06...
tests that each of your processes's file descriptors are independent from other processes

```
Failure while running test grade-file-06
Output up to 8000 bytes:
Picked up _JAVA_OPTIONS: -Xms128m -Xmx512m
nachos 5.0j initializing... config interrupt timer processor console user-check grader console

nachos.machine.AssertionFailureError: Unknown system call! Syscall: 2
at nachos.machine.Lib.assertTrue(Lib.java:89)
at nachos.machine.Lib.assertNotReached(Lib.java:106)
at nachos.userprog.UserProcess.handleSyscall(UserProcess.java:631)
at nachos.userprog.UserProcess.handleException(UserProcess.java:649)
at nachos.userprog.UserKernel.exceptionHandler(UserKernel.java:82)
at nachos.userprog.UserKernel$1.run(UserKernel.java:28)
at nachos.machine.Processor$MipsException.handle(Processor.java:603)
at nachos.machine.Processor.run(Processor.java:101)
at nachos.userprog.UThread.runProgram(UThread.java:31)
at nachos.userprog.UThread.access$000(UThread.java:11)
at nachos.userprog.UThread$1.run(UThread.java:20)
at nachos.threads.KThread.runThread(KThread.java:160)
at nachos.threads.KThread.access$000(KThread.java:30)
at nachos.threads.KThread$1.run(KThread.java:149)
at nachos.machine.TCB.threadroot(TCB.java:235)
at nachos.machine.TCB.access$100(TCB.java:25)
at nachos.machine.TCB$1.run(TCB.java:93)
at java.lang.Thread.run(Thread.java:619)
```

Running test grade-file-10...
tests that stdin uses console

Test grade-file-10 succeeds.

Running test lottery-inherit-01...
Small lottery scheduler test with priority donation

Unknown exit status (1) for test lottery-inherit-01

```
Output up to 8000 bytes:
Picked up _JAVA_OPTIONS: -Xms128m -Xmx512m
nachos 5.0j initializing... config interrupt timer processor console user-check grader

java.lang.NullPointerException
at nachos.threads.KThread.<init>(KThread.java:51)
at nachos.threads.KThread.<init>(KThread.java:68)
at nachos.threads.ThreadedKernel.initialize(ThreadedKernel.java:35)
at nachos.userprog.UserKernel.initialize(UserKernel.java:23)
at nachos.ag.AutoGrader.start(AutoGrader.java:48)
at nachos.machine.Machine$1.run(Machine.java:62)
at nachos.machine.TCB.threadroot(TCB.java:235)
at nachos.machine.TCB.start(TCB.java:118)
at nachos.machine.Machine.main(Machine.java:61)

--------------------------------------------------
```

5/9 tests passed

```
make: *** [tests] Error 1
make: Target `all' not remade because of errors.

<<PROBLEM: One or more tests failed (terminated with non-zero exit code 512).

*********************************************************************
```