System Calls

You are to implement system calls for user programs in Nachos. The current implementation supports only one system call: halt. All halt does is ask the operating system to shut the system down. There is a test program in test/halt.c that shows how it is used. Running this program will also help you getting more acquainted with running user programs as .coff files in Nachos. Additionally, there are other test programs that you can use to test your implementation, or you can write your own programs (you will need a cross compiler to compile from your architecture to MIPS).

Task 1: System Calls (100%)

Implement the file system calls (creat, open, read, write, close, and unlink, documented in syscall.h). You will see the code for halt in UserProcess.java; you should implement the other system calls in this same file. Remember that you are not implementing a file system, you are simply providing user programs the functionality to use the existing file system. Some important points:

The assembly code needed to invoke system calls from user programs is already provided (see start.s).

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 bogus arguments to the kernel which causes the kernel to corrupt its internal state or that of other processes. Also, you must take steps to ensure that if a user process does anything illegal – such as attempting to access unmapped memory or jumping to a bad address – that the process will be killed cleanly and its resources freed.

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 memory 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.

When a system call wishes to indicate an error condition to the user, it should return -1 (not throw an exception within the kernel!). Otherwise, the system call should return the appropriate value as documented in test/syscall.h.

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 make this easier. 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; the interface is given 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 your Nachos distribution. You do not need to implement any file system functionality. You should examine carefully the specifications for FileSystem and StubFileSystem in order to determine what functionality you should provide in your syscalls, and what is handled by the file system.

Do not implement any kind of file locking; this is the file system's responsibility. If ThreadedKernel.fileSystem.open() returns a non-null OpenFile, then the user process is allowed to access the given file; otherwise, you should signal 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.

Your implementation should support at least 16 concurrently open files per process. Each file that a process has opened should have a unique file descriptor associated with it (see syscall.h for details). The file descriptor should be a nonnegative integer that is simply used to index into a table of currently-open files by that 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 (i.e. integer) to refer to different files.

You can use the binary executables (using "-x filename.coff") in the test directory to test your code.

Submission

Submit a zip file with your solution. Your submission should include source code and a written report with a detailed discussion of your design and implementation choices. After submission I will meet with each group to discuss your solutions. The final grade will be 50% for your written report, and 50% for the correctness of your implementation.

This project is worth 50 points.