Project 2: Multiprogramming

The second phase of Nachos development is to support multiprogramming. As in the first assignment, we give you some of the code you need; your job 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 them via *system calls*.

The first step is to read and understand the part of the system we have written for you. The kernel files are in the `nachos.userprog` package, and there are a few additional machine simulation classes that get used:

- `Processor` simulates a MIPS processor.
- `SerialConsole` simulates a serial console (for keyboard input and text output).
- `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.

The new kernel files for this assignment include:

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

In this assignment we are giving you a simulated CPU that models a real CPU (a MIPS R3000 chip). By simulating the execution, we have 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. Our simulator can run normal programs compiled from C to the MIPS instruction set. The only caveat is that floating point operations are not supported.

The code we provide can run a single user-level MIPS program at a time, and supports just 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.

We have provided several other example MIPS programs in the `test` directory of the Nachos distribution. You can use these programs to test your implementation, or you can write new programs. Of course, you won't be able to run the programs which make use of features such as I/O until you implement the appropriate kernel support! That will be your task in this project.

The `test` directory includes C source files (`.c` files) and Nachos user program binaries (`.coff` files). The binaries can be built while in the `test` directory by running `gmake`, or from the `proj2` directory by running `gmake test`.

To run the `halt` program, go to the `test` directory and `gmake`; then go to the `proj2` directory, `gmake`, and run `nachos -d ma`. Trace what happens as the user program gets loaded, runs, and invokes a system call (the `'m'` debug flag enables MIPS disassembly, and the `'a'` debug flag prints process loading information).

In order to compile your own test programs, you need a MIPS **cross-compiler**. We will provide a cross-compiler on the server klwin00.ucmerced.edu (mips-gcc). If you are not using the server, you can also download the cross-compiler for your system (from Resources directory on Crops) and set the ARCHDIR environment variable accordingly.

There are multiple stages to building a Nachos-compatible MIPS binary (all of which are handled by the `test Makefile`):

1. Source files (`*.c`) are compiled into object files (`*.o`) by `mips-gcc`.
2. Some of the object files are linked into `libnachos.a`, the Nachos standard library.
3. `start.s` is preprocessed and assembled into `start.o`. This file contains the assembly-language code to initialize a process. It also provides the **system call "stub code"** which allows system calls to be invoked. This makes use of the special MIPS instruction `syscall` which traps to the Nachos kernel to invoke a system call.
4. An object file is linked with `libnachos.a` to produce a Nachos-compatible MIPS binary, which has the extension `*.coff`. (COFF stands for Common Object File Format and is an industry-standard binary format which the Nachos kernel understands.)

You can run other test programs by running

```
nachos -x PROGNAME.coff
```

where `PROGNAME.coff` is the name of the MIPS program binary in the `test` directory. Feel free to write your own C test programs -- in fact, you will need to do so for testing your own code!

As with Project 1, you should submit and document your test cases (for both the Java and C components of your code) with the project. For this project most test cases will be implemented as C programs which test your system calls, but some "internal" testing in Java may be possible as well.

I. (30%, 125 lines) 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`; it is best for you to place your new system calls here too. Note that you are *not* implementing a file system; rather, you are simply giving user processes the ability to access a file system that we have implemented for you.
   - We have provided you the assembly code necessary to invoke system calls from user programs (see `start.s`; the `SYSCALLSTUB` macro generates assembly code for each syscall).
   - 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.

- You should make it so that the `halt()` system call can only be invoked by the "root" process -- that is, the first process in the system. If another process attempts to invoke `halt()`, the system call should be ignored and return immediately.
- 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 of 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, which is going to be useful when you want to support the `exec` system call (see below). 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 non-negative 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.

II. (25%, 100 lines) Implement support for multiprogramming. The code we have given you is restricted to running one user process at a time; your job is to make it work for multiple user processes.

- Come up with a way of allocating the machine's physical memory so that different processes do not overlap in their memory usage. Note that the user programs do not make use of `malloc()` or `free()`, meaning that user programs effectively have no dynamic memory allocation needs (and therefore, no heap). What this means is that you know the complete memory needs of a process when it is created. You can allocate a fixed number of pages for the process's stack; 8 pages should be sufficient.

  We suggest maintaining a global linked list of free physical pages (perhaps as part of the `UserKernel` class). Be sure to use synchronization where necessary when accessing this list. Your solution must make **efficient** use of memory by allocating pages for the new process wherever possible. This means that it is **not** acceptable to only allocate pages in a contiguous block; your solution must be able to make use of "gaps" in the free memory pool.

  Also be sure that all of a process's memory is freed on exit (whether it exits normally, via the syscall `exit()`, or abnormally, due to an illegal operation).

- Modify `UserProcess.readVirtualMemory` and `UserProcess.writeVirtualMemory`, which copy data between the kernel and the user's virtual address space, so that they work with multiple user processes.

  The physical memory of the MIPS machine is accessed through the method `Machine.processor().getMemory()`; the total number of physical pages is `Machine.processor().getNumPhysPages()`. You should maintain the `pageTable` for each user process, which maps the user's virtual addresses to physical addresses. The `TranslationEntry` class represents a single virtual-to-physical page translation.

  The field `TranslationEntry.readOnly` should be set to `true` if the page is coming from a COFF section which is marked as read-only. You can determine this using the method `CoffSection.isReadOnly()`.

  Note that these methods should not throw exceptions when they fail; instead, they must always return the number of bytes transferred (even if that number is zero).

- Modify `UserProcess.loadSections()` so that it allocates the number of pages that it needs (that is, based on the size of the user program), using the allocation policy that you decided upon above. This method should also set up the `pageTable` structure for the process so that the process is loaded into the correct physical memory pages. If the new user process cannot fit into physical memory, `exec()` should return an error.

Note that the user threads (see the `UThread` class) already save and restore user machine state, as well as process state, on context switches. So, you are not responsible for these details.

III. (30%, 125 lines) Implement the system calls (`exec`, `join`, and `exit`, also documented in `syscall.h`).

- Again, all the addresses passed in registers to `exec` and `join` are virtual addresses. You should use `readVirtualMemory` and `readVirtualMemoryString` to transfer memory between the kernel and the user process.
- Again, you must bullet-proof these syscalls.
- Note that the state of the child process is entirely private to this process. This means that the parent and child do not directly share memory or file descriptors. Note that two processes can of course open the same file; for example, all processes should have file descriptors 0 and 1 mapped to the system console, as described above.
- Unlike `KThread.join()`, only a process's parent can join to it. For instance, if A executes B and B executes C, A is not allowed to join to C, but B *is* allowed to join to C.
- `join` takes a *process ID* as an argument, used to uniquely identify the child process which the parent wishes to join with. The process ID should be a **globally unique positive integer**, assigned to each process when it is created. (Although for this project the only use of the process ID is in `join`, for later project phases it is important that the process ID is unique across all running processes in the system.) The easiest way of accomplishing this is to maintain a static counter which indicates the next process ID to assign. Since the process ID is an `int`, then it may be possible for this value to overflow if there are many processes in the system. For this project you are not expected to deal with this case; that is, assume that the process ID counter will not overflow.
- When a process calls `exit()`, its thread should be terminated immediately, and the process should clean up any state associated with it (i.e. free up memory, close open files, etc). Perform the same cleanup if a process exits abnormally.
- The exit status of the exiting process should be transferred to the parent, in case the parent calls the `join` system call. The exit status of a process that exits abnormally is up to you. For the purposes of `join`, a child process exits normally if it calls the `exit` syscall with any status, and abnormally if the kernel kills it (e.g. due to an unhandled exception).
- The last process to call `exit()` should cause the machine to halt by calling `Kernel.kernel.terminate()`. (Note that only the root process should be allowed to invoke the `halt()` **system call**, but the last exiting process should call `Kernel.kernel.terminate()` **directly**.)

IV. (15%, 50 lines+existing priority scheduler) Implement a lottery scheduler (place it in `threads/LotteryScheduler.java`). Note that this class extends `PriorityScheduler`, you should be able to reuse most of the functionality of that class; the lottery scheduler should not be a large amount of additional code. The only major difference is the mechanism used to pick a thread from a queue: a lottery is held, instead of just picking the thread with the most priority. Your lottery scheduler should implement priority

donation. (Note that since this is a lottery scheduler, priority inversion can't actually lead to starvation! However, your scheduler must do priority donation anyway.)

- In a lottery scheduler, instead of donating priority, waiting threads *transfer tickets* to threads they wait for. Unlike a standard priority scheduler, a waiting thread always adds its ticket count to the ticket count of the queue owner; that is, the owner's ticket count is the **sum** of its own tickets and the tickets of all waiters, not the **max**. Be sure to implement this correctly.
- Your solution should work even if there are billions of tickets in the system (i.e. do not keep an array containing an entry for every ticket).
- When `LotteryScheduler.increasePriority()` is called, the number of tickets held by a process should be incremented by one. Similarly, for `decreasePriority()`, the number should be decremented by one.
- The total number of (real) tickets in the system is guaranteed not to exceed `Integer.MAX_VALUE`. The maximum individual priority is now also `Integer.MAX_VALUE`, rather than 7 (`PriorityScheduler.priorityMaximum`). If you wish, you may also assume that the minimum priority is increased to 1 (from 0).