CS350 **Operating Systems** **Fall 2006**

Assignment 1: Threads, Processes, Synchronization

Due date: 12:00 (noon), Tuesday, October 10, 2006

1 Requirements

This assignment requires that you enhance the Nachos operating system. It helps you get started with Nachos and covers some basic aspects of operating systems: threads, processes, and synchronization. To receive full marks for this assignment, you must add the following functionality to Nachos:

1. Implement kernel exception handling for the `AddressErrorException`. This exception, like all other exceptions, is defined in `code/machine/machine.h`. The simulated machine will generate this exception if a user process performs an unaligned memory reference or a memory reference that falls outside of the process' virtual address space. Your kernel should take some reasonable action when this exception occurs. This includes printing some details about the nature of the exception (e.g., memory location to be accessed) to the `dbgSysCall` debug stream.

2. Implement the `GetId` and `GetParentId` system calls. The desired behaviour of these and all other system calls is defined in `code/userprog/syscall.h`.

3. Implement the `SetPriority` and `GetPriority` system calls. `SetPriority` can be used to set the scheduling priority of the calling thread. `GetPriority` can be used to determine the scheduling priority of the calling thread. There are four possible priority levels: `P_HIGH`, `P_NORMAL`, `P_LOW`, and `P_VERY_LOW` (you will need to add `P_HIGH` to the definitions in `code/userprog/syscall.h`). Threads initially start at normal priority, but they can move to another priority level by making a call to `SetPriority`.

   A thread should never run unless there are no runnable threads at a higher priority level. This implies that if a thread with priority `P_LOW` creates a new thread by calling `ThreadFork` (cf. Problem 6), then the new thread (which has priority `P_NORMAL`) will be executed immediately, without returning to the original thread first.

4. Implement the `LockOpen` and `LockClose` system calls. The `LockOpen` call is used to give a process access to a named lock. `LockOpen` takes a single parameter, which is a lock name (a string), and it returns a `LockId`. If a lock with the specified name already exists in the system, the `LockOpen` call should return a `LockId` that the calling process can use to refer to that lock. If there is no lock with the specified name, the system should create a lock with that name and return a `LockId` that the calling process can use to refer to the new lock. The idea is that if two (or more) processes call `LockOpen` with the same name, the processes will get `LockId`s that refer to the same lock.

   `LockClose` takes a single `LockId` as a parameter. A process calls `LockClose` when it is finished using a lock. A call to `LockClose` releases the specified `LockId` in the calling process (meaning that the process can no longer acquire the lock). When a process terminates (either voluntarily or involuntarily), any locks it has opened should be closed.

5. Implement the `LockAcquire` and `LockRelease` system calls. The `LockAcquire` call takes two parameters, a `LockId` and a mode flag, which is either shared mode (`S_MODE`) or exclusive mode (`X_MODE`). Once a thread has successfully acquired a lock, i.e., once the `LockAcquire` system call has returned, the thread is said to *hold* the lock. It has a shared hold or an exclusive hold, depending on the mode that was specified when the lock was acquired. Lock acquisition must obey the following rules:

   - Any number of threads may concurrently hold a given lock in shared mode.
   - If a thread holds a given lock in exclusive mode, no other thread may hold that lock concurrently in either mode.

If a thread attempts to acquire a lock in such a way that these rules would be violated, it should be blocked (in the call to `LockAcquire`) until it can acquire the lock without violating the rules. Your implementation is not required to support lock mode (de)escalation. If a process currently holds a given lock, it may not re-acquire that lock in a different mode without first releasing the lock. For example, if a process holds a given lock in shared mode and it wishes to acquire the same lock in exclusive mode, it must first release the lock and then re-acquire it in the new mode.

The `LockRelease` call is used to release a thread's current hold on a lock. This may allow other blocked threads to acquire the lock. When a thread terminates, either voluntarily or because it is killed by the kernel, all locks currently held by the thread should be released.

If a process attempts to close a lock (using `LockClose`) that is currently held by one or more of its threads in either mode, the system should defer closing the lock until it has been released by the thread(s) that currently hold(s) the lock. Threads of the same process that are currently trying to acquire the lock, or that try to acquire the lock after the call to `LockClose` has occurred, should return immediately with an error code that indicates that the lock is no longer available to the process.

You are *not* required to implement deadlock detection or prevention for locks.

6. Implement the `ThreadFork`, `ThreadYield` and `ThreadExit` system calls, so that Nachos will support multithreaded processes. The `ThreadFork` call should create a new thread within the same process as the calling thread. `ThreadFork` takes two arguments. The first is a pointer to the function that the new thread should execute. The second is an (integer) argument for that function. The `ThreadYield` call should cause the calling thread to yield the processor to another runnable thread, if there is one. The other runnable thread may be in the same process as the thread that is yielding, or in a different process. Finally, `ThreadExit` should cause the calling thread to terminate. If the calling thread is the only thread in its process, this call should also cause the process itself to terminate, as if by a call to `Exit(0)`. If the calling thread is not the only thread in its process, the process should not terminate. See `code/test/concurrent.c` for an example of a multithreaded Nachos application program that uses these thread calls.

   When implementing `ThreadFork`, you may assume that the newly created thread will always exit properly, by calling either `Exit` or `ThreadExit` (or by crashing, for instance due to an illegal memory access), but will never return from the function that is passed to `ThreadFork` when creating the new thread.

Since you will not implement virtual memory support as part of this assignment, the combined address spaces of all running processes will have to fit within the physical memory of the (simulated) machine. As provided to you, this memory is quite small (16K bytes). You may wish to increase the amount of available memory so that there will be enough to share among several running processes. You may do this by changing the `NumPhysPages` in the file `code/machine/machine.h`. Note that this is the only aspect of the machine simulation that you are allowed to change. Please read the comments at the top of `machine.h` carefully.

Your design and implementation should be such that the operating system is isolated from user processes. There should be *nothing* that a user program can do (such as providing bogus parameter values to system calls) to corrupt the operating system or cause it to crash.

Proper design, testing, implementation and documentation of the first five features described above will be worth 80 marks out of a possible 100 marks. The remaining 20 marks are for multithreaded processes (`ThreadFork`) and their correct behaviour. In order to get full marks for each part, you must provide complete documentation for both your implementation and your test cases, explaining why you chose the implementation you did choose, explaining how it works, and explaining how the test cases make sure that your implementation is correct.

For questions 1-5, there is no distinction between a thread and a process. Each process has exactly one thread, as `ThreadFork` has not been implemented yet. For question 6, however, there is a difference, and killing a process, for instance, is not the same as killing a thread within that process. You should keep this in mind when designing the data structures you need for solving the earlier questions.

Threads within the same process share a common address space, and each process has exactly one address space. Thus, when implementing support for multiple threads within the same process, you may think of

an address space as a process, treat the address space's `id` as a process ID and an `AddrSpace` object as a process control block (PCB).

Multithreaded processes are the most difficult part of the assignment. For this reason, it is strongly recommended that you implement and test everything except multithreaded processes first, and work on multithreaded processes only if you have time. This will ensure that you are eligible to receive most of the assignment marks.

2 Getting Started

Your first step should be to read the assignment-related information on the course web page, including the instructions on how to install and build Nachos. Next, you should spend some time reading and understanding those parts of Nachos which are relevant to this assignment, and trying Nachos out.

The `code/test` directory in the Nachos distributions contains a number of Nachos application (user) programs. You can use these programs to test Nachos and try it out. You may also wish to build on these programs to test your Assignment 1 work. Of course, you should also create new test programs of your own design.

A trivial example of such a user program can be found in the file `code/test/halt.c`. All that it does is ask the operating system to shut down the simulated machine. Once you have installed Nachos, built it, and built the test programs, you can run the `halt` program on Nachos using the command `nachos -x ../test/halt`. Run this command in the Nachos build directory. You can use the built-in trace facility of Nachos to see what happens as the test program gets loaded, is executed, and invokes a system call (`Halt`). For example, you might try the command `nachos -x ../test/halt -d t`. This runs the `halt` program with thread-related (`t`) debugging messages enabled. You will find a complete list of the possible debugging flags in the file `code/lib/debug.h`.

The Nachos source code is spread across several directories. For the purposes of this assignment, you will be particularly concerned with the directories `code/userprog` and `code/threads`, especially the former. In the `code/userprog` directory you will find (among others) the following files:

`addrspace.*`: This code will create an address space in which to run a user program, and load the program code and data from a file into the address space.

`syscall.h`: Contains the Nachos system call interface - a complete list of the defined system calls and their prototypes.

`exception.cc`: The handler for system calls and other exceptions is here.

`synchconsole.*`: This is a simple, synchronous interface to the console, built on top of the machine's asynchronous interface.

`proctable.*`: This implements the Nachos process tables. Among other things, it tracks parent/child interprocess relationships and manages process exit status.

In `code/threads` directory you will find:

`main.cc`: The Nachos `main()` is here, as is a complete list of the possible command line arguments to Nachos. This is the place to start your code walkthrough.

`kernel.*`: All but two of the Nachos "global" variables are encapsulated in a `Kernel` object, defined here.

`thread.*`: The Nachos thread package is here. You probably don't need to change this code (though you are allowed to) but you do need to understand how to use threads.

`scheduler.*`: This implements the ready list. You'll want to understand this when you are working on `SetPriority`.

`synch.*`: This implements a set of synchronization primitives for Nachos threads: semaphores, locks, and condition variables (the last two of which can be used to implement monitors). You will want to use the primitives.

`synchlist.*`: This is essentially a list data structure implemented as a monitor, using the synchronization primitives from `synch.h`. It is used several places in system. You are also free to use it. It is also a good example to follow in case you want to implement any similar, synchronized data structures.

Finally, you will also want to take a look at the machine simulation, which is found in the `code/machine`. Remember not to change any parts of the machine simulation, except for `NumPhysPages` in `machine.h`. In this directory, you should focus on the interface (`*.h`) files. In particular:

`machine.h`: This is the most important file. Here you will find the constant `NumPhysPages`, which controls the amount of memory the simulated machine has. You may increase it if you wish to (see above). You will find a list of possible exception types defined. These are the exception types that your operating system must handle. The methods `ReadRegister` and `WriteRegister` are how your operating system examines and changes the simulated machine's registers. The machine's memory is defined as an array of characters (bytes) called `mainMemory`. Your operating system can examine and change the contents of memory by reading and writing from this array. See `code/userprog/addrspace.cc` for an example of operating system code that does this.