**Phase#1 Build a thread system**

1. Tasks:

1.1. (5%, 5 lines) Implement KThread.join(). Note that another thread does not have to call join(), but if it is called, it must be called only once. The result of calling join() a second time on the same thread is undefined, even if the second caller is a different thread than the first caller. A thread must finish executing normally whether or not it is joined.

1.2. (5%, 20 lines) Implement condition variables directly, by using interrupt enable and disable to provide atomicity. We have provided a sample implementation that uses semaphores; your job is to provide an equivalent implementation without directly using semaphores (you may of course still use locks, even though they indirectly use semaphores). Once you are done, you will have two alternative implementations that provide the exact same functionality. Your second implementation of condition variables must reside in class nachos.threads.Condition2.

1.3. (10%, 40 lines) Complete the implementation of the Alarm class, by implementing the waitUntil(long x) method. A thread calls waitUntil to suspend its own execution until time has advanced to at least now + x. This is useful for threads that operate in real-time, for example, for blinking the cursor once per second. There is no requirement that threads start running immediately after waking up; just put them on the ready queue in the timer interrupt handler after they have waited for at least the right amount of time. Do not fork any additional threads to implement waitUntil(); you need only modify waitUntil() and the timer interrupt handler. waitUntil is not limited to one thread; any number of threads may call it and be suspended at any one time.

1.4. (20%, 40 lines) Implement synchronous send and receive of one word messages (also known as Ada-style rendezvous), using condition variables (don't use semaphores!). Implement the Communicator class with operations, void speak(int word) and int listen().

speak() atomically waits until listen() is called on the same Communicator object, and then transfers the word over to listen(). Once the transfer is made, both can return. Similarly, listen() waits until speak() is called, at which point the transfer is made, and both can return (listen() returns the word). This means that neither thread may return from listen() or speak() until the word transfer has been made. Your solution should work even if there are multiple speakers and listeners for the same Communicator (note: this is equivalent to a zero-length bounded buffer; since the buffer has no room, the producer and consumer must interact directly, requiring that they wait for one another). Each communicator should only use exactly one lock. If you're using more than one lock, you're making things too complicated.

1.5. (35%, 125 lines) Implement priority scheduling in Nachos by completing the PriorityScheduler class. Priority scheduling is a key building block in real-time systems. Note that in order to use your priority scheduler, you will need to change a line in nachos.conf that specifies the scheduler class to use. The ThreadedKernel.scheduler key is initially equal to nachos.threads.RoundRobinScheduler. You need to change this to nachos.threads.PriorityScheduler when you're ready to run Nachos with priority scheduling.

Note that all scheduler classes extend the abstract class nachos.threads.Scheduler. You must implement the methods getPriority(), getEffectivePriority(), and setPriority(). You may optionally also implement increasePriority() and decreasePriority() (these are not required). In choosing which thread to dequeue, the scheduler should always choose a thread of the highest effective priority. If multiple threads with the same highest priority are waiting, the scheduler should choose the one that has been waiting in the queue the longest.