Self Testing:
You may also implement additional testing of your code and document it in the design document. More specifically, it is your responsibility to implement your own tests beyond the ones provided to you by the autograder to thoroughly exercise your code to ensure that it meets the requirements specified for each part of the project. Testing is an important skill to develop, and the Nachos projects will help you to continue to develop that skill. You can add calls to testing code in `ThreadedKernel.selfTest` and add class-specific code in `selfTest` methods of each class.

As a testing strategy, first start with simple tests and then implement more complicated tests. When something goes wrong with a simple test, it is easier to pinpoint what aspect of your implementation has a bug. When something goes wrong with a more complicated test, it is more difficult to determine where the bug may be, unless you can rule out all the causes that your simple tests have shown to already be correct. We also strongly recommend implementing tests as separate methods, rather than making changes to just one or a few methods. Instead of making a change to an existing test to evaluate new functionality, copy the test into a new method and make the change. This way, your earlier tests are always there in case you need to use them again. You can comment out calls to previous tests so that you can concentrate on one test at a time.

To help you get jumpstarted on testing, here are a handful of testing strategies for some of the tasks (sample test programs are provided in the assignment page):

- **Join testing:** Implement tests that verify if `B` calls join on `A` and `A` is still executing, `B` waits; if `B` calls join on `A` and `A` has finished executing, then `B` does not block; if a thread calls join on itself, Nachos asserts; if join is called more than once on a thread, Nachos asserts; one thread can join with multiple other threads in succession; independent pairs of threads can join with each other without interference.
- **Alarm testing:** Implement tests that verify that a thread waits (approximately) for its requested duration; if the wait parameter is `0` or negative, the thread does not wait; multiple threads waiting on the alarm are woken up at the proper times, and in the proper order. For examples and strategies for implementing tests, see the Testing section below.
- **Condition variable testing:** Implement tests that verify that `sleep` blocks the calling thread; wake wakes up at most one thread, even if multiple threads are waiting; `wakeAll` wakes up all waiting threads; if a thread calls any of the synchronization methods without holding the lock, Nachos asserts; `wake` and `wakeAll` with no waiting threads have no effect, yet future threads that sleep will still block (i.e., the `wake`/`wakeAll` is "lost", which is in contrast to the semantics of semaphores).

Note that Nachos has a number of command line arguments, two of which are particularly useful for debugging this project:

- Invoking `nachos -d t` will display thread-related debugging information, such as context switches and thread state changes. You can add your own debugging output for this flag using `Lib.debug(dbgThread, ...)`.
- Invoking `nachos -s <number>` with different `number`s will change when context switches happen.