Operating System: Project 1

Instructed by *Wei Xu*

Due on Oct 29, 2020

Task I

`join()` and `finish()` are modified. A `swap` thread is used to record the current thread in `join()`, which is suspended. When a thread calls `finish()`, it makes its `swap` thread ready.

`JoinTest()` is the test function, and it is called inside `KThread.selfTest()`. It creates three threads X, Y, Z. Z runs first, then X, finally Y. X joins both Z and Y, but Y starts later than X. So this function tests the two condition of `join()`.

For task I through IV, all the test functions are called in `ThreadedKernel.selfTest()`.

Task II

Test (`Condition2.selfTest()`) is a Consumer-Producer problem, where 5 consumers and 5 producers are ordered in a random manner. We need to check whether there is no item left in the end.

Task III

In the test (`Alarm.selfTest()`), 5 threads are created and each of them waits for a random time. A random time interval is placed between contiguous threads. The purpose of the test is to check whether some threads do not get enough wait time or wait for too long.

Task IV

Test (`Communicator.selfTest()`) is a modified Consumer-Producer problem, where each item is a random number. We need to make sure that all Listeners get exactly one word and all the words they get are different.

Task V

Test (`Lock.schedulerTest`) is implemented in `Lock.java`, where we construct 9 threads which are waiting for a lock that is held by a thread with relatively low priority. Meanwhile, a thread with higher priority consumes CPU time as long as the lock is not free and it revokes the *yield* function periodically. Without priority donation, thread 0 will be stuck forever.

Test (`Condition2.schedulerTest`) relies on the Consumer-Producer model to test whether the queue is correctly organized when the number of consumers get larger.

In both of the test, we print the information of the queue at some critical points.

Task VI

Only `Boat.java` needs to be modified to solve the problem on boating. The key to a successful solution is to have more people heading for Molokai than coming back to Oahu. Therefore, we must always have as many children at Molokai as possible to move the boat back to Oahu (which is the only way to make progress).

All testing cases are contained in test function `Boat.selfTest()`. Since the structure of the problem is fixed, the only manipulatable parameters are the numbers of adults and children on board. Nevertheless, the number of adults really does not matter much, since they always cross the river one-at-a-time. The number of children is crucial only in the sense of its parity (but not its absolute value). On the whole, it is actually reasonable to only test two types of cases:

- An even number of children and several adults;
- An odd number of children and several adults.

We also test on the edge case, where there are no adults.