Join

Our implementation of join function is like the following:

1. We first disable interrupt, restore it after the joining thread is done.
2. The current thread is put to sleep while waiting. We use a `restoreQueue` in joining thread to record the threads that are waiting for it to finish.
3. When the joining thread is finished, we enumerate all the threads that are waiting, making them ready.

Join test

We set a simple `join_test` function in `KThread` Class. It will first creat a `join_thread` that simply pings multiple times. Then we have another thread `A` that calls `join_thread.join()`. We can see that in the output, `join_thread` will first ping once, then we see that `A` starts to wait, `join_thread` will keep pinging and finish. Then `A` comes back and finishes.

It can be called in `KThread.selfTest`, called `join_test`.

Condition2

Very similar to `Condition`, we just put `Thread` in the `waitQueue` instead of `Semaphore`.

We let the current thread sleep if condition is going to sleep. When condition is awaken, we let the top thread in the queue be ready.

Condition2 test

It can be called in `KThread.selfTest`, called `condition2_test`. It creates two threads, waiting for the same lock. Thread 1 will first get the lock, print message and then sleep. Thread 2 then could get the lock, print message and wakes up 1. Then 1 will wake up 2, release lock and done. Then 2 is done.

In System output, we can see that 2 will not proceed until 1 first sleeps. 1 will not proceed until 2 wakes up it. 2 will not proceed to end until 1 release lock.

Alarm

We keeps every thread and its wait-until time in corresponding order in two linked lists. Every time we have a `waituntil(x)`, we just update the two lists together. For `timerInterrupt`, we simply check from the first time in the time list, if it's already smaller then we make its thread ready and keep going until we meet some time that's larger.

Alarm test

It can be called in `KThread.selfTest`, called `alarm_test`. We simply check 300 and 500 together.

Communicator

So basically we want speakers and listeners to line up in two separate queues. Once both queues have at least one element, we pair them up. If no pair can be made, we simply make them wait in one queue.

So for every `speak()`, acquire lock first, then we first check if there is listener, if not we just make it sleep, once it's awaken (which means some listener comes in and is waiting for a word), we push his word into the queue, release the lock.

If there is a listener, we simply put the word in the queue, awake the listener.

Similar logic applies to `listen()`.

Communicator test

It can be called in `KThread.selfTest`, called `communicator_test`. We set two speakers and two listeners. They pair up in the order they are in the thread queue originally.

Priority Scheduler

For a single priority queue, we just use a Linked List `waitList` to represent the queue. Everytime we call `pick_nextThread()`, we just enumerate the queue, find the element with largest effective priority and return that thread.

For cases where we consider priority donation, we need to keep track of multiple queues for each thread, since it may hold multiple locks and each lock corresponds to one priority queue. So each time `acquire()` is called, we add the corresponding priority queue to that thread's queue set, namely `wait_set`. Since only when a thread acquires a lock or is called as `join`, it will potentially need donation from other queues and thus changes its effective

priority. Thus we set a special value for its effective priority when it starts to hold a lock or join. In `getEffectivePriority()`, we check all queues in `wait_set`, find the maximum effective priority and set it. In this way we could minimize the calculation needed.

PriorityScheduler test

It can be called in `KThread.selfTest`, called `priorityscheduler_test`. We create three threads with priority 2,7,5 where thread 3 will join thread 1. So the order should be thread 2,1,3 and the output agrees.

Boat

The main logic for completing the itinerary is that:

1. if there are no children in molokai and boat is in Oahu, we send two children there.
2. if there are at least one child in molokai, at least one adult in Oahu and boat is in Oahu, then we let the adult bow to molokai.
3. if boat is in molokai, we let one child bow back to oahu.

We use three condition variables sharing one lock plus several sharing variables to make sure the boarding process is consisteng, i.e.:

1. By maintaining `boat_in_oahu`, we make sure that everytime a thread is awaken to proceed and it checks it has boat to board, the boat won't be taken by other people.
2. By maintaining `adult_ready`, we make sure that for people in oahu, it's either adult or child to board, instead of both of them.
3. By maintaining `need_pilot`, we make sure that if two children depart from oahu, exactly one of them will call `row` and the other call `ride`.

For adult, its logic is relatively simple, grab the lock, when it sees an opportunity to board and `adult_ready = true && boat_in_oahu == true`, it simply gets board, maitaining the number of adults in oahu so that child threads know when to stop (`nadult_oahu == 0`) as well as other sharing variables.

For child, if it's in oahu and `adult_ready == false`, then he needs to pair up with another child to drive to molokai. If `need_pilot==true` then it will be pilot and wake another child in oahu. Otherwise he will ride. In both cases he needs to maintain the number of children on both sides, and check for termination. Also add himself to molokai's waiting queue by calling `child_molokai.sleep()`.

If a child is in molokai, his sole job is to row back to oahu. When back, if there are at least one adult in oahu and at least one child in molokai, he can update `adult_ready = true` and wake an adult in oahu. Otherwise it's still children's turn and he will wake a child in oahu.

Boat test

It can be called in `KThread.selfTest`, called `boat_test`. We just test the three cases provided in `Boat.selfTest()`.