Project 1: Nachos Threads

Initial Design Document

Albert Luo, Brian Tan, Japheth Wong, Jonathan Eng, and Zhirong Gong

Part 1: `KThread.join()`

Our Solution

When `join()` is called on a thread, we will check the status of the thread to see if it has finished execution. If the thread has finished executing, we can return immediately; otherwise, we make the parent thread wait until the child thread wakes it up upon finishing execution. In order to accomplish this, we have added a field to keep track of the parent thread.

```
KThread parentThread = null;
join() {
	if (status == statusFinished) {
		return;
	} else {
		parentThread = KThread.currentThread();
		currentThread.sleep();
	}
}
```

We also added a mechanism to wake up a parent thread that is waiting for a child thread to finish execution: (gray-colored code indicates code that was provided)

```
finish() {
	...
	currentThread.status = statusFinished;
	if (parentThread != null) {
		parentThread.ready();
	}
	...
}
```

Correctness Invariants

- If `threadA` calls `threadB.join()` but `threadB` has already finished execution, we should immediately return.
- If `threadA` calls `threadB.join()` but `threadB` has not finished execution, `threadA` should wait until `threadB` finishes.
- Multiple calls to `threadB.join()` are undefined.
- Threads are supposed to finish execution regardless of whether `join()` is called.
- `threadA` must be woken up after `threadB.join()` is successfully called and `threadB` finishes.

Testing Strategy

- Have `threadA` fork `threadB` and allow `threadB` to finish execution. Call `threadB.join()` and check that `threadA.status == statusReady` or `threadA.status == statusRunning`. (So `threadA` was not forced to wait.)
- Have `threadA` fork `threadB` and interrupt `threadB`. Call `threadB.join()` and check that `threadA.status == statusBlocked`.
- Try calling `threadB.join()` and make sure thread B doesn't join twice.
- Try calling join on thread B before it has started running (before its `status == statusRunning`)
- Try calling join on thread B after it has finished running (when its `status == statusFinished`)

Part 2: Conditional Variables

Our Solution

Without the use of semaphores, we needed a way to be able to jump back to the `semaphore.P()` call when `V()` is called. To allow this behavior, we split the `sleep()` method implemented in `Condition` into two parts: `sleep()` -- which releases the lock and puts the thread to sleep, and `sleep2()` -- which reacquires the lock when the thread is woken back up.

```
sleep() {
      Lib.assertTrue(conditionLock.isHeldByCurrentThread());
      waitQueue.add(currentThread);
      conditionLock.release();
      currentThread.sleep();
}
```

```
sleep2() {
      Lib.assertTrue(!conditionLock.isHeldByCurrentThread());
      conditionLock.acquire();
}
```

As such, this means that `wake()` is responsible for ensuring that a thread that wakes up reacquires its lock. We disable interrupts in `wake()` provide atomicity. `wakeAll()` is responsible for signalling to *all* waiting threads, so it just calls `wake()` on each waiting thread.

```
wake() {
      disable machine interrupts here
      Lib.assertTrue(conditionLock.isHeldByCurrentThread());
      if (!waitQueue.isEmpty()) {
            thread = waitQueue.removeFirst();
            thread.ready();
            thread.sleep2();
      }
      reenable machine interrupts here
}

wakeAll() {
      while(!waitQueue.isEmpty()) {
            wake();
      }
}
```

The `waitQueue` helps ensure that calls to `wake()` or `wakeAll()` become no-ops if there are no waiting threads. The following is the constructor:

```
Condition2(Lock conditionLock) {
      this.conditionLock = conditionLock;
      waitQueue = new LinkedList();
}
```

Correctness Invariants

- Current threads must hold lock when doing conditional variable operations. If it does not have the lock, it must be asleep.
    - Threads calling `Condition2.sleep()` must atomically release the lock held by the thread and go to sleep (without using semaphores).
    - When waiting threads wake up, they must acquire lock before returning.
- `Condition2.sleep()` must sleep and wait for subsequent `wake()` or `wakeAll()` before proceeding. A `wake()/wakeAll()` called before `sleep()` should still force `sleep()` to wait.
- `Condition2.wake()` should wake up at most one thread. (If there are threads in the queue, wake up one thread. Otherwise, don't wake anything up.)
- `Condition2.wakeAll()` should wake up all threads in queue.

Testing Strategy

- Make a new condition variable that is accessed by multiple threads running simultaneously. Have one thread call `sleep()` on the condition variable. Check to see if the condition lock can be acquired.
- Check to see if all threads that call `sleep()` on this condition variable (prior to `wake()` being called) get added to the condition variable's `waitQueue`. Check to see if all threads who called `sleep()` prior to `wake()` have their status changed to `thread.status == statusBlocked`.
- When `wake()` is called on the condition variable, make sure the lock is acquired again. Check to see if the woken thread was removed from the waitQueue and has if thread.status == statusReady.
- Edge cases:
    - Two threads simultaneously call `Condition2.wake()` with only one waiting thread, make sure that possible thread interleaving doesn't result in two calls to `waitQueue.removeFirst()` when there's only one thing in the `waitQueue`.
    - Instantiate a condition variable with no lock (possible because of default constructor). Make sure calls to `sleep()`, `sleep2()`, or `wake()` on this condition variable don't crash the program.

Part 3: `Alarm` Class

Our Solution

Our solution contains two parts: `waitUntil()` is responsible for adding the thread to a priority queue `threadsWaitingUntil`, which will keep track of the next thread that needs to be woken up. We decided to use a priority queue because it allows us to maintain the threads to be returned sorted by the time they should be woken up and quickly check if the next item to be woken up should be woken up.

```
PriorityQueue threadsWaitingUntil = new PriorityQueue();
Lock pqLock = new Lock()   // Lock for priority queue.

waitUntil(long x) {
      wakeTime = current time + x;
      Disable interrupts
      pqLock.acquire();
      threadsWaitingUntil.add([Thread , wakeTime]);
      pqLock.release();
      Enable interrupts
}
```

The `timerInterrupt()` method then accesses this priority queue and checks to see if the next thread to wake up should be woken up. Since it is possible that more than one thread needs to be woken up, we will continue to wake up threads from the priority queue until we reach a thread which should remain asleep.

```
timerInterrupt() {
      KThread.currentThread().yield();
      Disable interrupts
      pqLock.acquire();
      while (thread in threadsWaitingUntil.peek().wakeUpTime <= currentTime) {
            KThread t = threadsWaitingUntil.pop();
            readyQueue.waitForAccess(t);
      }
      pqLock.release();
      Enable interrupts
}
```

Correctness Invariants

- Thread is on readyQueue no earlier than `time + x`.

- Each thread which calls `waitUntil()` is independent of the other threads that call it -- they will end up on the `readyQueue` at the appropriate time (as specified above) no matter how many other threads call `waitUntil()`.

Testing Strategy

- Use one thread with some time `x`. At time `(time + x - 1)`, check that the thread's `status == statusBlocked`. Wait for the first timer interrupt after `(time + x -1)` and then check that the thread's `status == statusReady`.
- Repeat above with 2-3 threads, scheduled in such a way that only one thread needs to be woken upon each call of `timerInterrupt()`.
- Repeat above with threads scheduled so that more than one thread is supposed to be woken up with a single invocation of `timerInterrupt()`.

Part 4: Implementing `Communicator` Class

Our Solution

One challenge in designing this class was determining a way to be able to correctly determine if speakers and/or listeners are available. We decided to include two fields - queues - which would enable us to see if a particular thread should wait or transmit/receive a message:

```
Queue listenerQueue, speakerQueue;
```

These queues serve one purpose: to keep a count of how many threads are currently waiting. When a speaker or listener is created, we add an object to the respective queue to serve as a counter. Whenever a thread is woken up to speak or listen, one of these counters is removed from the respective queue.

We decided that it is essential to protect the message being passed between a speaker to a listener thread to ensure that it is not overwritten while in transit. Two threads pass a message by saving it to the field `message`, which is protected by a lock. We make use of two condition variables, `okToSpeak and okToListen`, to notify a speaker thread when a listener is available and to notify a listener that a speaker has transmitted the message. This mechanism is demonstrated in detail below:

```
transmitMessage(word) {
        assert message == NO_MESSAGE;
        message = word;
}

retrieveMessage() {
        assert message != NO_MESSAGE;
        toReturn = message;
        message = NO_MESSAGE;      // Reset message so new message can be transmitted.
        return toReturn;
}
```

To implement `speak()`, we decided that as long as there are no waiting listeners *or* there are other speakers waiting as well, a speaker will be forced to wait. Once a listener is available, the speaker will write the message to be transmitted and then notify a waiting listener that the message is ready to be received. Similarly, `listen()` was designed so that when ready, a listener will notify a speaker. When notified by the speaker, the thread will retrieve the message and reset it for the next speaker before returning the message.

```
speak(word) {
        lock.acquire();
        add to speakerQueue (to maintain count of waiting speakers);
        while(listenerQueue is empty or other speakers in speakerQueue) {
                okToSpeak.wait();
        }
        transmitMessage(word);
        speakerQueue.pop();
        okToListen.signal();
        lock.release();
        return;
}

listen() {
        add to listenerQueue;
        while(speakerQueue is empty or other listeners in listenerQueue) {
                okToListen.wait();
        }
        okToSpeak.signal();
        int toReturn = retrieveMessage();
        listenerQueue.pop();
        return toReturn;
}
```

Correctness Invariants

- If thread A calls `listen()`, it should not return until another thread calls `speak()`.
- If thread A calls `speak()`, it should not return until another thread calls `listen()`.
- Although it is OK for multiple threads to wait to `speak()` *or* `listen()`, it is NOT OK to have *both*.
- Exactly one lock should be used for the `Communicator` class.

Testing Strategy

- Use three threads, `threadA`, `threadB`, and `threadC`.
  - Call `threadA.listen()`, then check that `threadA.status == statusBlocked`.
  - Call `threadB.listen()`, then check that `threadA.status == statusBlocked` *and* `threadB.status == statusBlocked`.
  - Call `threadC.speak()`, and check that (i) `threadC` returns and (ii) *either* (but *not* both) `threadA` or `threadB` returns.
  - Check that the remaining listening thread still has status of `statusBlocked`.
- Use four threads, `threadA`, `threadB`, `threadC`, and `threadD`.
  - Call `threadA.speak()` with some message `x`, then check that `threadA.status == statusBlocked`.
  - Call `threadB.speak()` with some message `y`, then check that `threadA.status == statusBlocked` *and* `threadB.status == statusBlocked`.
  - Call `threadC.listen()`, and check that (i) `threadC` returns and (ii) *either* (but *not* both) `threadA` or `threadB` returns. Check that the returned message is either `x` or `y`.
  - Check that the remaining listening thread still has status of `statusBlocked`.
  - Call `threadD.listen()`, and check that (i) `threadD` returns and (ii) the remaining speaker thread returns. Check that the returned message is the remaining message to be returned (if `x` returned earlier, we should see `y`, and vice versa).
- Use four threads, `threadA`, `threadB`, `threadC`, and `threadD`.
  - Call `threadA.speak()` and then `threadB.listen()`. Check to make sure there are no threads asleep.
  - Call `threadC.listen()` and then `threadD.speak()`. Check to make sure there are no threads asleep.

Part 5: Priority Scheduling

Our Solution

Our solution contains two parts:

1) `PriorityScheduler.PriorityQueue`

We implemented the `PriorityScheduler.PriorityQueue` class to be able to organize by priority and time it has waited in the priorityQueue (if there are multiple threads in the same queue with the highest priority).

2) `PriorityScheduler.ThreadState`

A modified TheadState class to be able to accommodate calculating and caching effective priorities while being able to maintain its “native” (non-effective) priority. It is implemented in a multiple parents (ThreadStates) but a single child (Kthread) fashion so that it can easily update the appropriate priorities. The ThreadState.child of “this” thread is the thread that “this” thread is waiting on. A `KThread` can only be waiting for one resource at a time so it can only have 1 child. The ThreadState.parents of “this” thread are all the threads waiting on “this”.

A ThreadState can have multiple parent ThreadStates. It takes the max of its parents’ effective priority and its own priority to determine its effective priority. Of course, we must dynamically update the effective priorities when the parent child relationships change. For instance, if thread B was waiting on thread A for a lock, and thread B get its turn to acquire the lock, thread A should no longer have thread B as it’s parent and thus should update its (thread A’s) effective priority accordingly. Only parents affect child effective priorities.

More details below when we discuss the ThreadState class.

First (1), we take a look at the `PriorityScheduler.PriorityQueue` class.

- `localThreads`: An internal java native priority queue to store all the threads and sort all the threads by priority.
- `currentThread:` Is a pointer to point to the thread currently that has is currently “active” (ex: holds the lock) and thus is not on the waiting queue (`localThreads`)
- `queueEntryTimes:` A hash table that hashes threads as the key and the entry time given by `Machine.timer().getTime()`

```
protected class PriorityQueue extends ThreadQueue {
private PriorityQueue<KThread> localThreads = null;
private KThread currentThread = null;
private Hashtable queueEntryTimes = null;
private Lock pqLock = new Lock();
```

For the constructor of a `PriorityQueue` we initialize `localThreads` to be a `PriorityQueue` that uses a `PriorityComparator()` so that the priority queue properly sorts the threads and pops out threads with the highest priority first. It also initializes the hashTable.

```
PriorityQueue(boolean transferPriority) {
	this.transferPriority = transferPriority;

	//Set up the PriorityQueue's internal way of keeping track of threads.
	localThreads = new PriorityQueue<KThreads>;
	localThreads.comparator = new PriorityComparator();

	currentThread = null;
	queueEntryTimes = new HashTable();
}
```

We provided a public interface for getting the currentThread from our implementation of a `PriorityQueue` so `ThreadStates` and other objects can get a pointer to the currently active thread.

```
public KThread getCurrentThread()  {
	return currentThread;
}
```

`waitForAccess()` is where we have to do some processing to ensure that all the effective priorities are updated properly for the priority queue.

Because machine interrupts are disabled, I am assuming that there is no way for us to get preemted to switch to another thread. Thus we shouldn't need to worry about two things accessing the priority queue at the same time and getting inconsistent results. For instance, no thread should be trying to put a high priority Thread A into localThreads (in the method `waitForAccess()`) and then get switched to another thread which calls `nextThread()` on the `PriorityQueue` without Thread A having been properly put on there (perhaps Thread A was going to be the most important thread in `localThreads` had it been successfully put on).

Because we have another thread now waiting for the currentThread to finish, we must add this new thread as a parent of the currentThread and then update the priorities accordingly. We must also input the time in which this thread entered the queue to be able to break ties with ties in priority (and pick the thread that has been waiting in the thread the longest).

```
public void waitForAccess(KThread thread) {
	Lib.assertTrue(Machine.interrupt().disabled());
	if (currentThread != null) {
		curThreadState = getThreadState(currentThread);
		curThreadState.parents.add(thread);
		curThreadState.calculateEffectivenessPriority();
	}
	queueEntryTimes.put(thread, Machine.timer().getTime());
	localThreads.add(thread);
	// So the threadState can access it's priorities.
	getThreadState(thread).waitForAccess(this);
}
```

When we acquire a resource (ex: a lock or the cpu), we are saying that we now have the resource and that any child we were waiting on for this resource, isn't necessary anymore. And thus we need to:

- Remove "this" thread from the child's parent list.
- Set our child to null as we have already gotten the resource we needed (this is done in `ThreadState.aquire()` as it should be taken care of on the abstract level of a thread state, and it is not the queues concern to manage parent child relationships as long as they don't affect the `PriorityQueue` structure).

We also make the assumption that when we acquire a thread directly like this instead of calling `nextThread()` that there should be no threads in the queue, else `nextThread()` should have been called.

```
public void acquire(KThread thread) {
	Lib.assertTrue(Machine.interrupt().disabled());
	assert localThreads.size() == 0;
	assert currentThread == null;
	// No need to add to the waitingQueue as it has the actual thing now.
	currentThread = thread;
	getThreadState(thread).acquire(this);
}
```

`nextThread()` is responsible for selecting the next thread we are going to run and "discard" the currentThread as the next thread we select will be taking its place. There are a few cases we take care of:

- CASE: If there are no more threads in `localThreads`.
  - Then we return `null` and the object which is using this queue (a `KThread` or `Lock`) will have it's backup precautions (run the idle thread, or release the lock respectively).
- CASE: If there are threads in `localThreads`.
  - Select the highest priority threads (all tied with the highest priority in `localThreads`).
  - Select the one of the highest priority threads that has been waiting for it the longest.
    - Because this is a single-cpu and single thread phase of the project, we assumed that no two threads should have the same input time, else we could add `random()` call to randomly select threads that have both the same priority (highest one) and the same entry time into `localThreads`.
  - But because we have selected the next thread (t) to replace currentThread, we must remove all the parents in `localThreads` from `currentThread.parents` as these parents are no longer waiting for `localThreads`.
  - We must also update the parents of the *new* `currentThread`.
    - Note that we do not have to recalculate priorities as the new `currentThread` should have the highest priority anyways.

```
public KThread nextThread() {
      Lib.assertTrue(Machine.interrupt().disabled());
      if (localThreads.peek() == null) {
            return null;
      }
      //Process currentThread's parents
      getThreadState(currentThread).removeParents(localThreads.allThreads());
      getThreadState(currentThread).calculateEffectivePriority();
      KThread t;
      int highestPriority = getThreadEffectivePriority(localThreads.peek());
      ArrayCollection<KThread> highestPriorityThreads = new
            ArrayCollection<KThread>();
      while (localThreads.peek() != null) {
            t = localThreads.poll();
            if (getThreadEffectivePriority(t) == highestPriority) {
                  highestPriorityThreads.add(t);
            }
      }
      if (highestPriorityThreads.size() > 1) {
            t = chooseThreadLongestWaitTime(queueEntryTimes);
      } else {
            t = highestPriorityThreads.remove(0);
      }
      //Process the NEW currentThread's parents
      queueEntryTimes.remove(currentThread);
      currentThread = t;
      currentThread.parents.add(localThreads.allThreads())
      return t;
}
```

Secondly (2), we take a look at the `PriorityScheduler.ThreadState` class.

- `effectivePriority`: Caches the effective priority we calculate for a given ThreadState. It should always be up to date as we should be recalculating it when appropriate.
- `parents`: An array collection of ThreadState objects that are waiting for "this" thread
- `child`: The KThread "this" thread is waiting on.

```
protected class ThreadState {
	protected int effectivePriority = -1;
	protected ArrayCollection<ThreadState> parents = null;
	protected KThread child = null;

	public ThreadState(KThread thread) {
		this.thread = thread;
		this.parents = new ArrayCollection<ThreadState>();

		setPriority(priorityDefault);
		calculateEffectivePriority();
	}

	public int getPriority() {
		return priority;
	}

	public int getEffectivePriority() {
		return effectivePriority;
	}
```

`setPriority()` takes a thread and changes its native priority (not effective priority).

- CASE: If the new priority is the same as our old priority, our effectivePriority and hence our descendants priority (child and deeper descendants) are guaranteed to not change.
- CASE: If our new priority is different from our previous native priority, we have to recalculate our effectivePriority which may thus in turn recursively affect all of our descendants' effective priority.

```
public void setPriority(int priority) {
	if (this.priority == priority)
		return;
	this.priority = priority;
	if (child != null) {
		getThreadState(child).calculateEffectivePriority();
	}
}
```

`calculateEffectivePriority()` is the key method for accurately updating the priority of all the appropriate threads. It is based on the key observation that only a parent can change a child's `effectivePriority` and thus because each `KThread` can only wait for one resource at a time (thus only have 1 child), the chain effect of recalculating priorities can be completed efficiently.

For efficiency we took care of a couple of cases:

- CASE: If our effective priority is different than before, our child needs to recalculate its effective priority to guarantee efficiency.
- CASE: If our effective priority remains the same
  - Then it does not affect any other effective priorities and we can cut off any further recalculation and thus save processing power.
  - Thus on average we expect this to save us processing power especially in systems that don't have priority changes at every step of the descendancy chain (parent -> child -> its child...).

```
public void calculateEffectivePriority() {
	highestPriority = -infinity;
	for thread in parents {
		if (getThreadState(thread).getEffectivePriority()) > highestPriority) {
```

```
                highestPriority = getThreadState(thread).getEffectivePriority()
            }
        }
        highestPriority = max(highestPriority, getPriority());
        int oldEffectivePriority = this.effectivePriority;
        this.effectivePriority= highestPriority;
        if (highestPriority != this.oldEffectivePriority) {
            if (child != null) {
                ( (ThreadState)child.scheduledState).calculateEffectivePriority();
            }
        }
        return;
}
```

`removeParents()` is a helper method to remove parents from the `ThreadState.parent` field. It does not recalculate `effectivePriorities`. So any call to `removeParents()` should be followed by a call to `calculateEffectivePriority()`.

```
protected void removeParents(ArrayCollection parentsToRemove = null) {
        if (parentsToRemove == null) {
            this.parents = new ArrayCollection<ThreadState>();
        } else {
            for (parent in parentsToRemove) {
                this.parents.remove(parent);
            }
        }
}

public void waitForAccess(PriorityQueue waitQueue) {
        child = waitQueue.getCurrentThread();
}
```

`acquire()` needs only to update its `child.parent` field because it now has the resource and no longer depends on the child. Thus we remove ourselves as a parent and update the `effectivePriority` of our child. The child field is reset to null as we are no longer waiting for any thread to attain a resource.

```
public void acquire(PriorityQueue waitQueue) {
        if (child != null) {
            ThreadState childThreadState = (getThreadState(thread).child);
            childThreadState.parent.remove(thread);
            child.calculateEffectivePriority();
            child = null;
        }
}
```

Correctness Invariants

- Priority donation must be transitive.
- The thread with the highest priority must always be the first one to run.
- Between threads of the same priority, the thread that has been waiting the longest must be the first to run
- Recalculating effective priorities should only occur when new threads are added to the queue and only for the threads that rely on that thread or for the threads that the incoming threads rely upon.

Testing Strategy

Run multiple threads of various priorities. Turn on priority donation. Have a high priority thread donate priority to a lower priority thread, run `getEffectivePriority()` on the lower priority thread to see if priority was effectively donated.

With priority donation on: Thread A has priority 7, B has 5, C has 4, D is 2.

Case 1: If C donates to D, and B donates to C, and A donates to B (in this order) Thread D should have a priority of 7.

Case 2: If D donates to C, Thread C should still have a priority of 4.

Run multiple threads of various priorities with no priority donation. No matter what happens, thread priorities should always be the same.

Check to make sure that the thread with the highest priority is always chosen.

Check to make sure that between threads of equal priority, the one that has been waiting the longest is always chosen.

Keep track of when `calculateEffectivePriority()` is called, this should only occur after a thread is added to the queue, and only called for threads related to the incoming thread (threads waiting upon the incoming thread, or threads that the incoming thread is waiting upon).

Part 6: `Boat` Problem

Our Solution

We will define the behavior of each thread as following:

Child:	Always want to get on the boat. If on Oahu, then it will be willing to be a rider, otherwise it only wants to pilot.
Adult:	Only wants to get on the boat if on Oahu, or if the adult is the only one on the island.

For this to happen, each thread must have knowledge of:

1 Which island it is currently on
2 How many people are on the island it is on
3 Where the boat is
4 Who is already on the boat
5 Is there a rider or not? Can it be a rider?

In order to satisfy (1), we will create a subclass of KThread that has a private variable that will be updated each time a `rowTo()` or `rideTo()` method is called, and a private variable so that it knows whether it is a parent or a child in order to call the correct methods. It also has methods to access and change that information.

```
Class PersonThread extends KThread {
   /* creates a PersonThread that knows what type it is, Child or Adult */
   constructor PersonThread(Runnable r, String type)  {
   super(r);
   private String myType = type
   private String myIsland = “Oahu” //each thread initializes to being on Oahu

   /* returns myType */
   getType() {
      return myType
```

```
        }

        /* returns myIsland */
        getIsland() {
           return myIsland
        }

        /* changes myIsland to the one that it is not currently on */
        changeIsland() {
        if (myIsland = “Oahu”) {
              then set myIsland = “Molokai”
        } else {
              set myIsland = “Oahu”
        }
    }
```

For (2) - how many people are on the island it is on), we will have two private variables that contain the counts for the number of people on each island. In order to control access to these variables, we will implement two get() methods that require the thread to verify that it is actually on the island before giving it the information. If the thread doesn’t have access to that information, it returns -1. A thread will only request the information to determine whether or not it is the only person on an island, so if it receives a negative number, it just won’t begin any special behavior.

```
    /* in begin() method */
    private numPeopleOnOahu; //initialized in begin
    private numPeopleOnMolokai; //initialized as 0.
    /* in boat class */
    getNumOnOahu()
       if(currentThread.getIsland()==Oahu) return numPeopleOnOahu
          else return -1
    getNumOnMolokai()
       if(currentThread.getIsland()==Molokai) return numPeopleOnMolokai
          else return -1
```

For (3 - where the boat is), we will instantiate a public variable to hold that information, and it will be updated each time a `RowTo()` method is called.

```
    private variable currentIsland = “Oahu”; //instantiated with “Oahu”
```

For (4 - who is already on the boat), we will have a conditional lock so that no more than 1 person can pilot the boat at a time.

```
    public global Condition pilot;
```

For (5 - Is there a rider or not? Can it be a rider?), there will be a public boolean that threads can check against, and an additional conditional variable so that only one rider goes at a time. Only child threads will check this boolean and access the conditional variable.

```
    public global Condition rider;
    public global boolean childPilot; //true if a child thread is a pilot.
```

Begin() is defined as follows: It initializes all variables. When a thread leaving Oahu sees that it is the last person on the island, it will finish(). Since after that thread leaves, there should be no more people on Oahu, the simulation will end.

```
public static void begin( int adults, int children, BoatGrader b) {
        bg = b; // Store the externally generated autograder in a class variable to be
           accessible by children.
        public global Condition rider, pilot;
        public global boolean childPilot = false; //true if a child thread is a pilot, starts
           as false as there is no pilot
        private variable currentIsland = “Oahu”; //instantiated with “Oahu”
        private numPeopleOnOahu = adults + children;
        private numPeopleOnMolokai = 0;

    for(int i=0; n<children; n++)
    Runnable r = new Runnable() {
       public void run() {
             ChildItinerary();
       };
       PersonThread n = new PersonThread(r, “child”);
       n.setName(""+i);
       n.fork();
    }

    for(int i=children; n<children+adults; n++)
    Runnable s = new Runnable() {
       public void run() {
             AdultItinerary();
       };
       PersonThread n = new PersonThread(s, “adult”);
       n.setName(""+i);
       n.fork();
    }
    return;
}
```

Adult behavior is defined as follows: it checks the boat’s current location and whether or not there is only one person at its current island. If the adult is at Molokai and it’s not the only person there, or the boat is not at Oahu, then it sleeps. When the adult wakes up, it rows to Molokai and updates all the counts accordingly.

```
    static void AdultItinerary() {
       pilotlock.acquire();
       while(currentThread.getIsland()==”Molokai” and numPeopleAtCurrentIsland!=1 or
          currentIsland!=”Oahu”)
             then sleep();
       if(currentThread.getIsland()==”Oahu”) then bg.AdultRowToMolokai(); else
          AdultRowToOahu();
       modifyCountsOfPeopleAtIslands();
       currentIsland = the other island.
       pilotlock.release();
    }
```

Child behavior is defined as follows: it checks the boat’s current location. If the boat is not at the child’s current island, then it sleeps. When it wakes up, it checks if another child is piloting the boat already. If not, then it becomes the pilot. Then, it rows to the other island and modifies the numbers accordingly. If it was the last person to leave Oahu, it signals the parent thread to `begin()` , because that means everybody has moved out of Oahu. It then wakes up another child so that child can ride to Molokai. If there is a pilot already, and it is currently on Oahu, then it rides to Molokai and updates the numbers accordingly.

```
    static void ChildItinerary() {
       pilotlock.acquire();
       while(currentIsland!=currentThread.getIsland()) then sleep();
       if (childPilot is false) {
       childIsPilot = true;
             if(currentThread.getIsland()==”Oahu”)
                    then ChildRowToMolokai(); else ChildRowToOahu();
             if(numPeopleAtOahu == 1),  finish();
       modifyCountsOfPeopleAtIslands;
       wakeChildOnCurrentIsland();
       pilotlock.release();
       return;
       }
       else {
             riderlock.acquire()
       if(currentThread.getIsland()==”Oahu”) {
              bg.ChildRideToMolokai();
              modifyCountsOfPeopleAtIslands;
             currentIsland= the other island
       }
       childIsPilot = false;
       pilotlock.release();
       riderlock.release();
       return;
       }
       pilotlock.release();
       return;
    }
```

Correctness Invariants

- Boat must have pilot in each direction.
- There should never be two consecutive calls to `rowToMolokai()` or `rowToOahu()`.
- `ChildRideToMolokai()` or `ChildRideToOahu()` should only be called if the respective `rowTo()` method is called by a child.
- `AdultRideTo()` methods should never be called; if an adult is on a boat, it is the pilot.
- A boat can carry only: (i) two children, or (ii) one adult.
- A thread can only access state variables associated with the island it is on.
- `finish()` must be called at the end after the last transport to Molokai

Testing Strategy

- Fork a single child thread on Oahu. Attempt to access variables associated with Molokai, but we should not be able to read the value of those variables. Repeat with an adult thread on Oahu, adult thread on Molokai, and a child thread on Molokai.
- Fork a single thread on Oahu and run the simulation. Check that exactly one boat trip is made.
- Fork two threads representing children and one thread representing an adult on Oahu. Check that when the adult goes to Molokai, he has no passengers. Check that if the child is the pilot *and* there is a child remaining, that the second child is a passenger.
- Fork a few threads and call `rowToMolokai()` or `rowToOahu()` twice in a row. Check that this action is not allowed.
- Fork a few child threads on Oahu and call `ChildRideToMolokai()`. (Note that there is no pilot.) Check that this action is not allowed. Repeat for `AdultRideToMolokai()`.