Project 1: Nachos Threads

Final Design Document

Albert Luo (aj), Brian Tan (am), Japheth Wong (cf), Jonathan Eng (cb), and Zhirong Gong (al)

TA: Daniel Bruckner

Section 105 (Tu 11-12)

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.

```
parentThread = null;
define join() {
        disable machine interrupts;
        if parentThread is not null or finished executing, then: Enable interrupts and return;
        else:
                set parentThread to currentThread
                PriorityScheduler ps = new PriorityScheduler();
                ps.getThreadState(this).joinedParentThread = parentThread;
                ps.getThreadState(this).updateEffectivePriority(true);
                put currentThread to sleep
        Enable interrupts;
}
```

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): then parentThread.ready();
        ...
}
```

Correctness Invariants

- If `threadA` calls `threadB.join()` but `threadB` has already finished execution, 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

For `sleep()`, we first disable interrupts to prevent other concurrency issues. The thread should automatically reacquire the lock after being woken up.

```
define sleep() {
        disable interrupts
        put currentThread on scheduler
        release conditionLock
        KThread.sleep();
        acquire conditionLock
        enable interrupts
}
```

`wake()`checks that there is a thread on the wait queue. If there is nothing on wait queue, then it does nothing. Otherwise, it puts the next thread on the ready queue. `wakeAll()`runs `wake()` on all threads in the wait queue.

```
wake() {
       disable interrupts
       if waitQueue is not empty: then scheduler.nextThread().ready();
       enable interrupts
}
wakeAll() {
       while waitQueue is not empty:  wake();
}
```

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

Correctness Invariants

- Current threads must hold lock when doing conditional variable operations or 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`.
- 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()` or `wake()` on this condition variable don't crash the program.

Part 3: `Alarm` Class

Our Solution

Our solution contains two main parts: `waitUntil()`, and `timerInterrupt()` methods, as well as new fields:

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

`waitUntil()`: `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.

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

`timerInterrupt()`: The `timerInterrupt()` method 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() {
        Disable interrupts
        if priority queue not empty:
                while (firstThread.wakeTime <= currentTime): firstThread.ready();
        Enable interrupts
}
```

LowPriorityComparator

This compares `ThreadAlarmTime` objects according to their `wakeTime`.. It's a standard Java comparator.

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 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

We decided that we would need three conditional variables to allow the threads to communicate:

`okToSpeak`	Used to ensure that speakers will wait for a listener to be present before speaking.
`okToListen`	Used to ensure that a listener will wait for a speaker to be present before listening.
`okToFinish`	Used by speakers and listeners to notify others (i) when speaker has transmitted message, and (ii) when listener is ready to receive message. This also helps us ensure that the message transmitted is not clobbered by another speaker before listener has a chance to read its value.

We designed `speak()` to wait until a listener is available. Once one is available, our speaker waits until the last transmitted message has been received by a listener before transmitting its own message. Upon transmission, the speaker tells the listener to read the message by calling `okToFinish`.

```
speak(word) {
      lock.acquire();
      increment numSpeakers;
      if no listeners or have waiting speakers, then: okToSpeak.sleep();
      okToListen.wake();
      while messageFieldInUse:  okToSpeak.sleep();  // Wait for message field to clear.
      transmitMessage(word);
      decrement numSpeakers;
      okToFinish.wake();  // Signal that message is ready to be read.
      lock.release();
      return;
}
```

Similarly, `listen()` follows the same idea, first waiting for a speaker to become available. Once a waiting listener is woken up by a speaker, it will read the first message that is transmitted and return it.

```
Listen() {
      lock.acquire();
      increment numListeners;
      if no speakers or have waiting listeners, then: okToListen.sleep();
      okToSpeak.wake();
      while not messageFieldInUse:  okToFinish.sleep();  // Wait for message to propagate.
      toReturn = retrieveMessage();
      decrement numListeners;
      lock.release();
      return toReturn;
}
```

We felt that it was essential to protect the message field, which is used by speakers to send the message and by listeners to retrieve the message. As such, we took extra care to maintain a `messageFieldInUse` flag which marks whether there is a message waiting to be retrieved, switched upon each invocation of `transmitMessage()` and `retrieveMessage()`. Because there may be speakers waiting for the message field to clear, it is important for us to signal the speakers in `retrieveMessage()` to allow them to send their message.

```
transmitMessage(word) {
      assert no message waiting to be retrieved (!messageFieldInUse)
      set messageFieldInUse flag (set it to true)
      message = word;
}

retrieveMessage() {
      assert message exists to be read;
      toReturn = message;
      clear messageFieldInUse flag (set it to false)
      okToSpeak.wake();
      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()`. 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 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 that there are no threads asleep.
    - Call `threadC.listen()` and then `threadD.speak()`. Check that there are no threads asleep.

Part 5: Priority Scheduling

Our Solution

In our implementation, we created two major classes:

- **`PriorityScheduler.PriorityQueue`** (extends `ThreadQueue` class). Our `PriorityQueue` sorts threads by priority and if multiple threads have the same priority, we sort them by the time they were added.
- **`PriorityScheduler.ThreadState`**, which holds information about a thread's priority, effective priority, and any object that it owns, and the queue that it might currently be waiting for.

We also had a few minor classes:

- **`PriorityScheduler.PriorityQueueEntry`**, which is a wrapper class for `ThreadState` that pairs the `ThreadState` with the entry time that it is submitted into a queue. A `ThreadState` is wrapped into this `PriorityQueueEntry` before it is entered into the `PriorityQueue` itself. Thus, `PriorityQueue` is a queue that only contains instances of `PriorityQueueEntry`.
- **`PriorityScheduler.PriorityComparator`**, which extends the normal Java `Comparator` class that is passed into `Java.PriorityQueue`, allowing it to sort the `KThreads` in the queue by priority. The outputs for the comparator are flipped from the normal Java Comparator so that entries into PriorityQueue are sorted into descending order by priority. This `PriorityScheduler` is only used for the `Java.PriorityQueue` in `ThreadState`, which keeps track of the `KThreads` waiting on this thread.
- **`PriorityScheduler.PriorityTimeComparator`**, which extends the normal Java `Comparator` class that is passed into `PriorityQueue`, allowing it to sort the instances `PriorityQueueEntry` in the queue by priority. The outputs for the comparator are flipped from the normal Java Comparator so that entries into `PriorityQueue` are sorted into descending order by priority. When `PriorityQueueEntries` have the same priority, the `Comparator` breaks the tie by sorting based on entry time. Threads that have been waiting for a longer period of time are given preference over other threads of the same priority.

Overview:

We wanted a modified `TheadState` class to be able to accommodate calculating and cache effective priorities while being able to maintain its “native” (non-effective) priority. Each `ThreadState` keeps track of all threads waiting on it as a `PriorityQueue` called “parents” that stores these threads as their associated `ThreadState` for easy access. Each `ThreadState` can only wait on one resource at a time, and can thus only wait on one other thread at a time. The thread that each `ThreadState` is waiting on is called its “child” and is stored as a `KThread`.

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.

PriorityQueue Class:

This modified version of a Java native priority queue is a type of threadQueue, which is initialized by the following constructor:

```
PriorityQueue(boolean transferPriority) {
        this.transferPriority = transferPriority;
        int initialCapacity = 10;
        this.waitingThreads = new PriorityBlockingQueue(initialCapacity, new
                PriorityTimeComparator());
        this.currentThread = null;
}
```

waitingThreads is a PriorityBlockingQueue, taking in PriorityQueueEntry objects, and comparing them with the PriorityTimeComparator. We added a few fields:

waitingThreads	PriorityQueue of threads that are waiting to gain access to some resource such as a lock.
currentThread	Pointer that points to the thread that is currently "active" (ex: holds the lock) and thus is not on the waiting queue (waitingThreads)

We also added methods to PriorityQueue:

(1) isEmpty(), which checks threads waiting on this resource
(2) waitForAccess(), which creates a PriorityQueueEntry object from the thread's schedulingState and current time that is added to our queue of waitingThreads.

```
waitForAccess(KThread thread) {
   assert: interrupts disabled, thread != null, and associated ThreadState != null
   if (currentThread == null), then: acquire(thread);
   PriorityQueueEntry pqe = new PriorityQueueEntry(getThreadState(thread), current time);
   Add pqe to waitingThreads;
   getThreadState(thread).waitForAccess(this);
}
```

Some assumptions we made and edge cases we can deal with:

- Machine interrupts are disabled when we call waitForAccess() because we don't want any interruptions as we add our PriorityQueueEntry() to the waitingThreads queue.
- The thread we pass in to waitForAccess should also not be null.
- The schedulingState of the thread we pass in should also not be null, or else we would be trying to create a PriorityQueueEntry from a null object.
- If the current thread is null, need to acquire thread before calling waitForAccess() on threadState.

(3) acquire() sets currentThread and acquires its associated ThreadState object. When this happens, the ThreadState object checks to ensure its not already waiting on a queue, since it doesn't make sense for it to wait on two queues, and then removes the child it is waiting on from the waitQueue.

```
acquire(KThread thread) {
   assert: disabled interrupts, currentThread = null, waitingThreads is empty, thread
      and its ThreadState aren't null
   currentThread = thread;
   getThreadState(thread).acquire(this);
}
```

Assumptions we made:

- Before we tell our priority queue to acquire a thread, we need to make sure interrupts are disabled.
- We also need to check that `currentThread` is `null` and the thread that gets passed in does not equal `null` because the current thread that is about to be set to the thread passed in always needs to be pointing to some thread.

Edge cases we can catch:

- If current thread is `null`, need to acquire thread before calling `waitForAccess()` on its `ThreadState`.
- When a `ThreadState` calls `acquire()`, it needs to set the current queue its waiting on to `null` so that it doesn't wait on both the current queue and the queue passed in as its argument.
- We need to make sure the `ThreadState`'s child is actually pointing to something before removing it.

(4) `nextThread()` spits out the next thread queued in the `PriorityQueue`. This thread should be the highest priority thread remaining in the resource queue. When `nextThread()` is called, this means that the previous thread has finished its process and the resource is now open for another thread to access the resource. Thus, all threads in the queue must set their child pointers from the old thread to this new incoming thread and the old thread needs to remove all of its pointers to threads that are no longer waiting on it as a result. The new incoming thread is removed from the `PriorityQueue` (as it is no longer waiting), allowed to access the resource, and becomes the new current thread.

```
nextThread() {
   assert: interrupts are disabled
   if no waitingThreads, then: set currentThread = null and return null;
   // Process currentThread's parents and update effective priority
   oldParents = getThreadState(currentThread).disownParents(this);

   currentThread = waitingThreads.poll().thread();
   getThreadState(currentThread).becomeCurrentThread(oldParents, this);
   return currentThread;
}
```

We assumed that interrupts must be disabled before running `nextThread()` because we don't want to mess pointer changes that have to occur for this function to run. If the `PriorityQueue` is empty, set the current thread to `null` and return `null` because there's nothing else to return.

(5) `updateThreadPriority()`: For `PriorityQueues`, the queue is unable to automatically restructure/update the priority of one its elements whose priority is modified. The changed thread needs to be removed from and reinserted into the queue. Essentially: search the `PriorityQueue` for the `PriorityQueueEntry` associated with the `ThreadState` that we wish to remove and, once we find it, we remove it and re-add it. The `PriorityQueue`'s `Comparator` does the rest of the work for us.

```
updateThreadPriority(ThreadState tState) {
   PriorityQueueEntry[] pqeEntries = waitingThreads in array form
   for each pqe in pqeEntry:
         if (tState == pqe.threadState()) {
               assert that tState and pqe.threadState's effective priorities match
               remove, then add pqe to waitingThreads
         }
}
```

When we find the `PriorityQueueEntry` associated with the `ThreadState` (which are associated with some thread) that we want to remove, the priority and that thread must necessarily be equal. If no `PriorityQueueEntry` associated with the `ThreadState` that we want to remove from the `PriorityQueue` can be found, do nothing.

ThreadState Class: Construct a new ThreadState object and associate it with the specific KThread that it takes in as an argument. ThreadState objects store and calculate the effective priorities associated with its thread, as well as keep a copy of the thread's native, non-donated priority.

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 its parent. Thread B no longer donates its priority to Thread A and thus Thread A should change its effective priority accordingly. Only parents affect child effective priorities.

Some important methods:

(1) Self-explanatory getters: `getPriority()` and `getEffectivePriority()`, which return those fields.
(2) `setPriority()`: Every ThreadState has an associated priority, and sometimes this priority needs to be changed for various reasons. Once this method is called, the ThreadState saves the new priority and runs `updateEffectivePriority()` in order to check whether or not this thread has another thread of even higher priority donating to it. If it does, then the `setPriority()` is later overwritten in the `updateEffectivePriority()`, else the `setPriority` stays as it is.

```
setPriority(int priority) {
   if (this.priority == priority) then return;
   set this.priority = priority;
   if queueWaitingOn != null: updateEffectivePriority(queueWaitingOn.transferPriority);
   else: updateEffectivePriority(true);
}
```

Every ThreadState has an associated priority which may need to be changed for various reasons. Once called, the ThreadState saves new priority and runs `updateEffectivePriority()` in order to check whether or not this thread has another thread of even higher priority donating to it. If it does, then the `setPriority()` is later overwritten in the `updateEffectivePriority()`, else `priority` stays as it is.

Assume that this is only called by ThreadState constructor and by `PriorityScheduler.setPriority()`. The ThreadState constructor sets it to given default priority (which must be between the set minimum and maximum priority), and the `PriorityScheduler.setPriorty()` method (which checks to ensure that the priority that it's setting the method to is between minimum and maximum priority). Thus the priority that the ThreadState is set to will never be outside the boundaries of the given minimum and maximum priority boundaries. Note: If this ThreadState is not waiting in a queue, assume that priority donation is active.

(3) `waitForAccess()`: Called by `PriorityQueue.waitForAccess()` when some outside thread tries to access this resource but finds it already in use. All such threads call `waitForAccess()` to signal their status as being waiting on the resource. This means that the ThreadState has to save the pointer of the queue that it's waiting on and another pointer to the thread hogging the resource as the waiting thread's child before updating child's priority. The thread has already been added to the resource's waitQueue by the PriorityQueue's `waitForAccess()` method.

```
waitForAccess(PriorityQueue waitQueue) {
   assert that waitQueue != null and queueWaitingOn == null
   queueWaitingOn = waitQueue;
   addChild(waitQueue.currentThread, waitQueue); //Auto sets childThreadState.parent
   getThreadState(child).updateEffectivePriority(waitQueue.transferPriority);
}
```

If this thread has had waitForAccess called on it, then it means that it has just emerged out of a previous queue and has attempted to acquire this resource. Since threads cannot wait on more than one resource at a time, this means that it must not have been waiting on any other resource before attempting to acquire this resource. Thus, `this.queueWaitingOn` must be null, because it's not waiting on anything else.

If `waitForAccess` is called on a thread, that means that the thread is waiting in a queue to access some resource—thus, there must be some queue associated to this resources and so `waitQueue` cannot be `null`.

(4) `acquire()`: When called, thread has acquired the resource that this `PriorityQueue` is guarding. Thus, it is no longer waiting on any resource (until it tries to call `acquire()` on another resource) so, `queueWaitingOn` variable to cleared to `null`. Assume that threads can only wait on one resource at a time.

```
acquire(PriorityQueue waitQueue)  {
   queueWaitingOn = null;
   if child not null, then: removeChild(waitQueue);
}
```

(5) `updateEffectivePriority()`: Called when we need to change its cached effective priority. In order to update its effective priority, this thread then takes the highest priority of the parents that are donating their priority to it and sets it to its own priority (if this donated priority is higher than its own, natural priority). This thread then calls `updateEffectivePriority()` on its child so that the priority changes get propagated down the inheritance chain. If this thread has no parents, don't even worry about changing the values or propagating the priority changes down the chain.

```
updateEffectivePriority(boolean transferPriority) {
   if transferPriority == false, then: return;
   // Consider parent's donated priority to get "donated" effective priority.
   int oldEffectivePriority = this.effectivePriority;
   int highestPriority = this.priority;
   int parentPriority = highestPriority - 1;
   KThread parent;

   // Consider parent's donated priority to get the "donated" effective priority
   if no parents, then:  parents = new java.util.PriorityQueue<KThread>();
   if parents exist: highestPriority=max(highestPriority,nextThreadState's eff. priority)
   if joined: highestPriority = max(highestPriority, joinedParentThread's eff. priority)
   this.effectivePriority = highestPriority;

   // Compare parent's highest "donated" priority with my own native priority
   if (effectivePriority != oldEffectivePriority && child != null):
         getThreadState(child).updateEffectivePriority(transferPriority);
   if (effectivePriority != oldEffectivePriority && queueWaitingOn != null):
         queueWaitingOn.updateThreadPriority(this);
}
```

(6) `addChild()`: Called when this thread is in a queue waiting for a resource and the thread that was hogging the resources just got popped off and got replaced by a new thread. Thus, it needs to flush out the old thread as its child and called `addChild` on this new thread. The child thread needs to also add this thread as its parent to mark the fact that this thread is waiting on its child. The method as a whole is just a way of bookkeeping, making sure that the threads have a pointer towards the correct child/parent as a result of changes within the queue. The target thread of `addChild()` is a recipient of a lot of new parent threads donating their priority to it—so it's effective priority needs to be updated.

```
addChild(KThread c, PriorityQueue waitQueue) {
   assert: child is null, c is not null
   this.child = c;
   add this thread to our child ThreadState's parent field;
   getThreadState(child).parents.add(this.thread);
   getThreadState(child).updateEffectivePriority(waitQueue.transferPriority);
}
```

We assumed that this thread must have already have cleared out its old child pointer and must not be waiting on any other thread or resource than this one, since threads can only wait for one resource at a time.

Also, Threads cannot add `null` threads into their child pointer. If they're waiting in a queue for a resource, that resources better be being used right now.

(7) `removeChild()`: Called on all threads waiting in a queue for a particular resource when a thread finishes using that resource and it removed from the lock. Thus, this thread is no longer waiting on any particular thread to finish using the resource, since currently no one has been given the go to acquire the resource. The child thread updates its list of parents to reflect this, and removes this thread from its list of parents.

```
removeChild(PriorityQueue waitQueue) {
   assert that child != null;
   remove this thread from our child ThreadState's parents;
   updateEffectivePriority(waitQueue.transferPriority) on child's ThreadState
   child = null;
}
```

We assumed this thread must have already have cleared out its old child pointer and must not be waiting on any other thread or resource than this one, since threads can only wait for one resource at a time. Threads cannot add null threads into their child pointer. If they're waiting in a queue for a resource, that resources better be being used right now.

(8) `disownParents()`: Called on a thread when it finishes using a resource, releases the lock on that resource, and some other thread is called by `PriorityQueue.nextThread()` to acquire the lock for that resource. All threads in the queue are no longer waiting on the original thread and so, they no longer donate their priority and are thus no longer the thread's parents. So, this thread needs to empty out its list of parents and make sure that all threads previously pointed to this thread as their child now set child to null. Assume that a thread can only wait on one queue at a time and `disownParents()` is called before the exiting thread can acquire another resource.

```
disownParents(PriorityQueue currentQueue) {
   PriorityQueueEntry[] oldParentsCurrentQueuePQE = currentQueue.waitingThreads as array
   KThread[] oldParentsCurrentQueue = new KThread[oldParentsCurrentQueuePQE.length];
   KThread p;
   for PriorityQueueEntry pqe in oldParentsCurrentQueue:
         p = pqe.thread();
         Store p in oldParentsCurrentQueue;
         getThreadState(p).child = null;
   }
   Clear parents;
   updateEffectivePriority(currentQueue.transferPriority);
   return oldParentsCurrentQueue;
}
```

(9) `becomeCurrentThread()`: Called when `PriorityQueue.nextThread` is called. This method just sets the internal bookkeeping process properly, making sure that every thread waiting for this resource sets this thread as their child, and updates this thread's effective priority based on these new parents. Assume this thread should no longer be waiting for a resource at the time that `PriorityQueue.nextThread()` is called and so it should also not be waiting for any particular thread to finish as a result.

```
becomeCurrentThread(KThread[] oldParents, PriorityQueue waitQueue) {
   queueWaitingOn = null, child = null;
   for parent in oldParents:
         if parent in parents and parent != this.thread:
               getThreadState(oldParents[i]).addChild(this.thread, waitQueue);
   updateEffectivePriority(waitQueue.transferPriority);
}
```

(10)`gotJoined()`:When a thread gets joined, the effective priority value of the thread needs to be updated. Assume that only one parent can be joined at any time.

```
gotJoined(KThread parThread) {
   this.joinedParentThread = parThread;
   if (queueWaitingOn != null), then:
      updateEffectivePriority(queueWaitingOn.transferPriority);
   else: updateEffectivePriority(queueWaitingOn.transferPriority);
}
```

`PriorityQueueEntry` Class:

A `PriorityQueueEntry` keeps track of a `ThreadState` and stores entry times of the thread it is associated with. There are two private fields associated with a `PriorityQueueEntry` object: `threadState` and `entryTime`. `priority()` gets the effective priority of the thread its connected to, `entryTime()` allows us to ask for the stored `entryTime`, and `thread()` returns the pointer to the actual `KThread` that is a part of the `threadState`.

`PriorityComparator` Class: (implements `Comparator`)

This is a simple comparator that compares two `KThreads` according to their effective priorities. It implements the behavior of a normal Java Comparator based on the priority of the thread.

`PriorityTimeComparator` Class: (implements `Comparator`)

The `PriorityTimeComparator` compares two `PriorityQueueEntries` by getting the effective priorities associated with the two `PriorityQueueEntries`, which are stored in the `PriorityQueueEntries`' `threadStates`. If two threads have the same priority, then they are sorted based on their entry times.

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 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, 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 longest is 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

Our basic algorithm is to first have all the children go over and have one child acting as the boat ferry, piloting the boat back and forth between the islands so that each adult can go over.
To do this, we first have 4 global variables that tell us how many people are on each island.

```
        int ActualNumChildOnOahu, ActualNumChildOnMolokai, ActualNumAdultOnOahu,
        ActualNumAdultOnMolokai = 0;
```

We also have a variable to keep track of the boat's location.

```
        String boatLocation = "Oahu";
```

We then have Condition variables and appropriate locks to ensure that only one `Adult` thread or two `Child` threads are using the boat at a time, and two booleans to check each child's role on the boat:

```
        Lock riderLock, pilotLock;
        Condition rider = new Condition(riderLock);
        Condition pilot = new Condition(pilotLock);
        boolean childIsPilot, boatHasChildAsRider;
```

Lastly, we have a lock and condition variable so that the threads will be able to signal `begin()` when they think the simulation is finished, and a boolean so that the threads will know when to stop running.

```
        Lock finishLock = new Lock();
        Condition finishCondition = new Condition(finishLock);
        boolean finished = false;
```

`begin()` should start out by creating and forking the appropriate number of `Adult` and `Child` threads. It then sleeps as long as the number of people on Molokai is not the same as the number of people it started with. Once that condition is true, it should terminate the simulation.

```
begin( int adults, int children, BoatGrader b ) {
        acquire finishLock;
        for each adult:
                Create thread n with Runnable with run() = AdultItinerary();
                name and fork n;
        for each child:
                Create thread n with Runnable with run() = ChildItinerary();
                name and fork n;
        while (numPeopleOnMolokai != adults + children):  finishCondition.sleep();
        finished = true;
        finishCondition.sleep();
```

We will have five local variables for each thread that allows it to count how many people are on each island each time it visits, and a way for it to know which island it is on.

```
int numChildOnOahu, numAdultOnOahu, numChildOnMolokai, numAdultOnMolokai;
String currentIsland;
```

When first run, the `Adult` thread should check into Oahu. Then, it should keep running until the simulation is finished. While running, it should consider two cases: either it is on Oahu or on Molokai. If it is on Oahu, it should wait until there is only one child left on Oahu and the boat is at Oahu, then go to Molokai. If it is on Molokai, it should make sure that there is at least one child on Molokai that can pilot the boat back to Oahu. Each time it rows to a new island, it should count how many people are on that island and update the variables accordingly.

```
AdultItinerary() {
         checkInToIsland;
         Instantiate private variables
         acquire(lock)
         while not finished {
                 if currentIsland is Oahu {
                         countNumberOfPeopleOnIsland();
                         while numChildOnOahu != 1 or boat not at Oahu {
                                 sleep;
                                 countNumberOfPeopleOnIsland();
```

```
                    }
                    AdultRowToMolokai();
                    updateVariables();
                }
                else {
                    countNumberOfPeopleOnIsland();
                    while boatLocation != Molokai {
                        sleep;
                        countNumberOfPeopleOnIsland();
                    }
                    if(numChildOnMolokai==0) {
                        AdultRowToOahu();
                        updateVariables();
                    }
                }
               sleep;
           }
           sleep;
    }
```

Child thread should also check in to Oahu the first time it is run, and keep running until the simulation is finished. It should consider two main cases; either it is on Oahu or Molokai. If it is on Oahu, then it must consider 3 cases:

(1) There are other children on Oahu. Child either needs to be a pilot or a rider on the boat to Molokai. If it's a pilot, it should wait for a rider to get on. Otherwise, will be a rider.
(2) There are adults on Oahu. → Child should wake an adult and let the adult go to Molokai.
(3) It is the only person on Oahu → Child should go to Molokai and signal `begin()`, then continue looping.

```
ChildItinerary() {
    Check in to island, instantiate variables, and acquire pilotLock;
    while(not finished):
        if currentIsland = Oahu:
            countNumberOfPeopleOnIsland();
            if boatLocation = Oahu:
                Count people on island;
                if (numChildOnOahu>1):
                    if child is not pilot:
                        childIsPilot = true;
                        while !boatHasChildAsRider: sleep;
                        childRowToMolokai(); updateVariables();
                        childIsPilot = false;
                        wakeUpRider()
                    else if(!boatHasChildAsRider):
                        boatHasChildAsRider = true;
                        wakeUpPilot();
                        childRideToMolokai();  updateVariables();
                        boatHasChildAsRider=false;
                else if(numAdultOnOahu>0): Count people on island; wake adult;
                else:
                    childRowToMolokai();
                    Update variables and signal begin();
        Else:
            Count number of people on island;
            if nobody on island: then signal begin();
            else if boatLoation = Molokai and !boatHasChildAsRider:
                childRowToOahu();  updateVariables();
        Signal begin() and sleep;
    Signal begin() and sleep;
}
```

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. Should not be 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()`.
- Test multiple cases to see if it finishes running. 2 children with varying numbers of adults, stress tests with hundreds of threads, varying number of children, etc.

Additional Questions

Why is it fortunate that we did not ask you to implement priority donation using semaphores?

Semaphores can represent multiple resources while condition variables represent only a single resource – so each semaphore could have multiple threads that allow it to "pass through". There are two problems:

(i) If a thread is waiting on a semaphore, we have to update priorities of all threads currently running while holding semaphore. This means a lot more priority updating than condition variables chained together.

(ii) Assume thread A (priority 5) acquires a semaphore with 2 resources, and thread B (priority 10) acquires same semaphore. There should be no effective priority updating because both threads can be run (so no priority inversion). Consider what happens when B acquires before A – still no priority inversion. But if A is also holding a condition variable and C wants to acquire it, C would donate priority to A, so A has a higher effective priority than B, but because A and B are on same semaphore, A won't donate priority to B, so we get priority inversion. This is complicated!

A student proposes to solve the boats problem by use of a counter, `AdultsOnOahu`. Since this number isn't known initially, it will be started at zero, and incremented by each adult thread before they do anything else. Is this solution likely to work? Why or why not?

Yes -- if we give priority to the children to go to Molokai. Thus, we would move all the children to Molokai first, and send a child back ONLY to allow the adults on Oahu to get across to Molokai. This means that we can maintain as an invariant that, when the first adult begins to cross to Molokai, that there will always be at most two children on Oahu. Thus, we can end one round after `AdultsOnOahu` = 0 -- to allow the up to two remaining children on Oahu cross to Molokai.