Project #1 Threads

Hamel Ajay Kothari (de), Maliena Guy (dd), Jack Wilson (dx), Bryan Cote-Chang (ed)

Section 106, TA: Kevin Klues

Task 1: KThread.join()

Correctness Constraints

- When join() is called by a given thread (A) on thread (B). Thread A should be put to sleep and added to thread B's wait queue.
- When thread B has finished execution, thread A (and to be safe any other threads on its wait queue) should be woken up and resume execution.
- PriorityQueues must work (ie. be able to transfer priority) with thread joins.

Declarations

We will only need one field to be added to our KThread class in order to make this implementation work:

- *waitingThreads* - A ThreadQueue created using *scheduler.newThreadQueue(true)* for storing threads waiting for this thread to finish.

Description

To implement our join() call we first assert that the thread is not finished and that we're not calling join() on ourselves. After that we add the thread that has called join (the currentthread) to that queue by calling waitForAccess() and put it to sleep.

In the finish() method of KThread, we iterate over the queue of threads waiting on our thread and place them on the ready queue by calling *ready()*. This can be done as follows:

```
KThread nextThread = waitingThreads.nextThread();
while(nextThread != null) {
    nextThread.ready();
    nextThread = waitingThreads.nextThread();
}
```

We also need to call *waitingThreads.acquire(this)* in the constructor of our KThread in order to ensure that our KThread recieving priority donations from the contents of the ThreadQueue.

Testing Plan

We will test our code by creating a *selfTest()* method in our KThread class which contains some test logic for thread joining. Then we will test our thread joining in the following way:

- We can test this code by creating some threads which each print a series of messages, and call thread.join() from their parents and look at the print statements to ensure that the threads print their messages in the correct order.

Task 2: Condition Variables

Correctness Constraints

- When a KThread calls *sleep()* on the given condition, it must be holding the lock for that condition and it must give it up and be put to sleep.
- When another KThread calls *wake()* on the given condition, it must also be holding the corresponding lock and it must place a sleeping thread on the ready queue.
- When a thread is woken up after calling KThread.sleep(), it must reacquire the lock which it released.
- When *wakeAll()* is called all the threads sleeping on the condition must be woken up.
- The operations should all be completely atomic through the use of interrupts.

Declarations

- *conditionLock* - The lock which we hold on to when we are using our condition.
- *waitingThreads* - A ThreadQueue created using *scheduler.newThreadQueue(true)* for holding on to the threads which are waiting on this condition.

Description

We implement the *sleep()* method as follows:

1. Disable Interrupts (BEGIN CRITICAL SECTION)
2. Release the lock
3. Add our thread to the wait queue
4. Call KThread.sleep() (ENDS/BEGINS CRITICAL SECTION)
5. Acquire the lock.
6. Restore interrupts (ENDS CRITICAL SECTION)

Similarly, in the wake() method we do the following:

1. Disable interrupts (BEGIN CRITICAL SECTION)
2. Take an element off the wait queue
3. If the element is not null, call ready to show that the thread is now ready to run.
4. Restore interrupts (BEGIN CRITICAL SECTION)

For the *wakeAll()* method we perform the same logic as in the wake method but rather than taking only the next thread. We will loop over all the threads while our *nextThread* is not null, we add it to the ready queue.

Testing Plan

In order to test our code we can create test lock and test condition. Then we can create a test thread which acquires the lock, wakes on the test condition,

Task 3: Alarm

Correctness Constraints

- When we call Alarm.waitUntil(x) we want our thread to be put to sleep for at least x ticks.
- On each timerInterrupt we want to make sure that there are no threads sleeping which should have been woken by the given Machine time.

Declarations

- *alarmQueue* - A java.util.PriorityQueue which tracks the threads which need to be woken up and their corresponding times, sorted using a comparator which orders them by their most soon wake time.

Description

In order to implement our Alarm class we need to implement the waitUntil method and the timerInterrupt method. To make our lives easier we will create a container class called AlarmNode which contains each element we put into our *alarmQueue* which simply contains our KThread which we have put to sleep and the time at which we want to wake it up.

Then for our PriorityQueue, we initialize with a comparator which compares based on the wakeTime field of our AlarmNode class so that whenever we retrieve the next element from our queue, it is the earliest one due to be woken up.

Now, for the implementation of waitUntil we first calculate the time to wake up our thread as the current system time, plus the parameter x. Then we disable interrupts to ensure that the next operations are atomic. From here we create an AlarmNode object with the current thread which is calling waitUntil and the wakeTime we calculated earlier. We add this to the queue, and then put the current thread to sleep. After the KThread.sleep() call we restore interrupts.

For our timerInterrupt we must ensure that all the threads that need to be woken up at this point are woken up. To do so we can peek() the first element off our queue, and if it's not null and it's wakeTime is less than or equal to the current machine time meaning that the time that it was supposed to be woken has passed, we take it off the queue and call ready. We repeat this until the while loop is false as follows:

```
long machineTime = Machine.timer().getTime();
while((alarmQueue.peek() != null) && (alarmQueue.peek().wakeTime <= machineTime)) {
    alarmQueue.poll().thread.ready();
}
```

Testing Plan

We can easily test that our Alarm class works by creating two threads, each which print the time they start at (eg. "Test Thread A: Started at 2000"), and then call Alarm.waitUntil with different times, in our case we sleep for 2000 ticks and 50 ticks, and then after this we print the time they finish at. We then fork off each of these two threads and join them both and look at the output. In the case we defined above, we should see Thread B finishing first somewhere in the order of 50 ticks after it is added to the *alarmQueue* after that we should see Thread A finish somewhere around 2000 ticks after it is added to the queue.

Task 4: Synchronous Send/Recieve

Correctness Constraints

- A speaker will not terminate unless its message has been recieved by a listener.
- A listener will not terminate unless it has recieved a message from a speaker.

- A speaker will communicate a message to one listener, therefore no two listeners can terminate by recieving the same message.

Declarations

- *messageLock* - a lock providing protection for our message and listenerCounts.
- *message* - an integer represent the message we are sending/recieving.
- *messageInUse* - a boolean represent the message is being used or whether it can be overwritten because it has been read by a listener.
- *listenerCount* - a count keeping track of whether there are listeners currently waiting for messages
- *listeningCondition* - a condition for speakers to sleep on if there are no listeners present.
- *speakingCondition* - a condition for listeners to sleep on if there is not currently a message to read.

Description

Our send method will work as follows:

1. Acquire lock
2. Check listenerCount and if listenerCount = 0 or the message is in use, sleep on listeningCondition.
3. Set our message in use vriable to true.
4. Set our message variable to our desired message.
5. Wake on spoken condition.
6. Release our lock

Our receive will work as follows:

1. Acquire lock
2. listenerCount++
3. Wake on listeningCondition
4. sleep on spokenCondition
5. Get our message variable
6. Set our message in use variable to false.
7. listenerCount–
8. wake any sleeping speakers waiting to send a message
9. Release our lock

Testing Plan

We can test our code by having a permutation of speakers and listeners started in different orders and ensuring that messages are received in the right amount and order. To do this we create 4 threads, one which sends two messages, and one which sends one message and two threads which listen for messages in a similar manner. We log when each call to speak() and listen() happens to keep track of when they begin and terminate speaking to ensure that they're terminating in an acceptable manner. We also look at the results of the listen calls to ensure that they are unique like the messages that we sent out.

Task 5: Priority Scheduling

Correctness Constraints

- When a thread waitsForAccess on a given PriorityQueue it should donate its effective priority to the owner of the queue if the given queue has *transferPriority* set to true.
- When a thread acquires a given PriorityQueue it should release the current owner of the thread, forcing it to remove its donations (and calculate its effective priority), and it should gather donations from the contents of the queue.
- Items with higher effective priorities should be chosen over items with lower priorities on the same queue and in the event of equality, the item which has been on the queue longer should be chosen first.
- When you set the priority of a given thread, its effective priority and the effective priority of the threads it donates to should be recalculated.

Declarations

Our declarations for our PriorityQueue involve the following:

- *stateQueue* - a *java.util.PriorityQueue* which holds the ThreadStates of the KThreads which are waiting to acquire the resource protected with this queue. This queue will be backed with a comparator which compares our threads ordering in a descending order of *getEffectivePriority()* and if they match up it compares in an ascending order of the time that they were put into the queue.
- *blockingThread* - the ThreadState of the KThread which is blocking access to this queue (ie. the state for the KThread which owns the resource which this queue protects.).

For our ThreadState class, our declarations are as follows:

- *thread* - The KThread which this state object tracks.
- *priority* - The vanilla priority of our KThread which can be gotten/set by the user.
- *effectivePriority* - an integer representing the calculated effective priority which is recalculated when our thread's *priority* changes, or when new threads wait for access to the resources which our thread owns, or alternatively any of the threads within a queue of our owned resources change priority.
- *waitingQueue* - A PriorityQueue (not java.util.PriorityQueue) which is a reference to the queue which our KThread is currently waiting on.
- *waitingTime* - A long representing the Machine time at the point which our KThread was added to the *waitingQueue*.
- *ownedQueue* - A *HashSet¡PriorityQueue¿* which contains references to all the PriorityQueues which our KThread has gained ownership of.

Description

For PriorityQueue we need to implement a few methods to get our solution to work. The first method we would logically implement would be the *pickNextThread()* method. This one is fairly simple using our java PriorityQueue by calling *stateQueue.peek()* to return the next element in our queue sorted by priority or null if there are no elements in our queue.

Our next method we must implement is *nextThread()* which is also reasonably simple. We first ensure that our *!stateQueue.isEmpty()* if it is empty, we return null, otherwise we call *stateQueue.poll()* to get the next ThreadState. We then call acquire on that next thread state and return the corresponding thread for that thread state.

The bulk of our logic for our PriorityQueue actually resides in the ThreadStates of our threads. The methods which we have implemented are *setPriority()*, *getPriority()*, *getEffectivePriority()*, *waitForAccess()*, *acquire()*, *release()* and *recalculateEffectivePriority()*.

The *getPriority()* and *getEffectivePriority()* methods each return the corresponding fields, *priority* and *effectivePriority*. The *setPriority()* method is reasonably simple as well, it just sets the *priority* field and then calls *recalculateEffectivePriority()* method delegating most of its logic there.

Our *waitForAccess()* it a bit more complicated. We want to place our thread on the *stateQueue* and keep track of the time that it was put on the queue as well as recalculate the effective priority of the owner of the queue if there is on. We do this as follows:

```
waitForAccess(PriorityQueue waitQueue)
    waitingQueue = waitQueue
    waitingTime = Machine.timer().getTime()
    waitingQueue.stateQueue.add(this)
    if waitingQueue.blockingThread != null
        waitingQueue.blockingThread.recalculateEffectivePriority()
```

When we acquire ownership of the resource (either through *nextThread()* or *acquire()* in PriorityQueue) the *acquire()* method of our ThreadState gets called with the PriorityQueue we are acquiring. To handle this we release the previous owner from the queue, remove ourself from the *stateQueue* if we're in it, and add this queue to our owned queues. We also set our thread to be the blockingThread of the queue and zero out our *waitingQueue* field and waitingTime field. Finally we recalculate our effective priority. The looks like the following:

```
acquire(PriorityQueue waitQueue)
    if waitQueue.blockingThread != null
        waitQueue.blockingThread.release(waitQueue)
    waitQueue.stateQueue.remove(this)
    waitQueue.blockingThread = this
    ownedQueues.add(waitQueue)
    if waitingQueue = waitQueue
        waitingQueue = null
        waitingTime = 0
    recalculateEffectivePriority()
```

The *release()* method is a bit more straight forward. All we have to do is remove queue that is passed in from our *ownedQueues*, set the *blockingThread* of that queue to null because we no longer own it, and then recalculate our effective priority to factor in that we no longer have donations from that queue.

Finally, our *recalculateEffectivePriority()* method is where a large amount of our logic takes place. Here we must update our *effectivePriority* as well as propigate the changes to wherever we may have been donating our priority. The method will function similar to the following:

```
recalculateEffectivePriority()
    if waitingQueue != null
        waitingQueue.stateQueue.remove(this)
    currPriority = priority
    for PriorityQueue owned in ownedQueues
        TheadState ts = owned.pickNextThread()
        if ts != null
            int tsEffective = ts.getEffectivePriority()
            if tsEffective > currPriority
                currPriority = tsEffective
    bool mustPropigate = ((currPriority != effectivePriority) && (waitingQueue != null))
    effectivePriority = currPriority
    if waitingQueue != null
        waitingQueue.stateQueue.add(this)
    if mustPropigate
```

```
if (waitingQueue.transferPriority && waitingQueue.blockingThread != null)
    waitingQueue.blockingThread.recalculateEffectivePriority()
```

Testing Plan

In order to test our code we can set up a simple scenario of priority donation. We will have 3 priority queues: *pq1*, *pq2*, *pq3* and 5 threads, *t1...t5*. Then we will have *t1...t3* waitForAccess on the three corresponding queues. We will also call acquire on the three queues using *t2...t4*. Now if we set the priority of *t3* to 6 we will expect the effective priority of *t4* to be 6 as well. If we now set the priority of *t5* to 7 and have it waitForAccess on *pq1* we will expect *t4* to now have an effective priority of 7 as well. Finally if we call *nextThread()* on *pq1*, *t4* will no longer be the owner of this resource and no longer be getting donations from the queue, so we should assert that its effective priority is now 1.

Task 6: Boat Problem

Correctness Constraints

- Our *begin()* method must terminate with the correct sequence of rowing children/adults to/from Oahu in order to get everyone from Oahu to Molokai.
- We can only row one adult to the other shore in a boat.
- We can row either one or two children to the other shore in a boat but no combination of adults and children.
- Our threads must function pretty much independently (as individual people) and be able to follow the preprogrammed logic to get everyone to the other side.

Declarations

All of the members defined below will be defined in our Boat class as static members and initialized in the *begin* method to ensure they are reset after each run.

- *bg* - Our BoatGrader which we will be calling to trigger the boat to leave an island.
- *boat* - A RowBoat object, which represents our boat and the operations which make sense for a boat in the real world. This class will have to be implemented to make our solution work.
- *numberOfAdultsOnOahu* - an int representing the number of Adults on Oahu, inititalized to 0
- *numberOfChildrenOnOahu* - an int representing the number of Children on Oahu, initialized to 0
- *numberOfAdultsOnMolokai* - an int representing the number of Adults on Molokai, initialized to 0
- *numberOfChildrenOnMolokai* - an int representing the the number of Children on Molokai, initialized to 0
- *boatLock* - a Lock which ensures atomicity to all of our operations through ownership of the boat.
- *boatOnOahu* - a condition to sleep on until the boat arrives on Oahu.
- *boatOnMolokai* - a condition to sleep on until the boat arrives on Molokai.
- *okForAdultToLeaveOahu* - a condition to sleep on until an adult is allowed to leave Oahu, in our case we will let this be the case that there is only one child remaining on Oahu.
- *okForTwoChildrenToLeaveOahu*
- *okForOneChildToLeaveOahu*

- *okForChildToLeaveMolokai*
- *childDeliveredToOtherSide* - a condition which a Child sleeps on when they are riding to another shore with another child rowing.
- *simulationStart* - a condition which all of our Child/Adult threads sleep on until all the threads have been forked and initialized.
- *simulationOver* - a condition which our main thread waits on until it is time to exit.

Description

To make our lives easier we will create a few helper classes which will represent the objects in our simulations, namely a RowBoat, a Child and an Adult.

Our RowBoat will represent the one boat we have to use to row back and forth from Oahu and will be protected by our boatLock. It will have counters for the number of children and adults in the boat as well as an array tracking the occupants of the boat and a boolean tracking whether the boat is on Oahu. Then our boat will have a couple of logic methods. The first method is *rowToOtherSide()* which will check the location of the boat, and row to the other shore and wake all the threads sleeping on the condition that the boat is on the side it just rowed to. During the rowing process, the boat will also update the counters of people on either side by subtracting its child/adult count from the shore it was on and adding it to the shore it was going to. It will also set the location of the occupants of the boat to the new location of the boat and make the appropriate calls to our boat grader depending on if an adult/child is rowing the boat and if someone is riding in the boat. We also will have a *load* method which takes a person and puts them in the boat assuming they can fit. It will throw an exception if they can't but we can ensure they can always fit by checking the contents of the boat before calling *load*.

We also will have representations of our people with a Child and Adult class which track the location of that person with an *onOahu* boolean, and each of which implement Runnable so they can be run as threads. In the Adult *run()* method we will make a call to the AdultItinerary passing *this* to say we want to run with this Adult. We do the same with Child and ChildItinerary.

Now with these structures/tools in place we just need to come up with the actual logic for the Adult and Child itineraries and the logic of the begin method.

As I said above, in our *begin()* method we will perform the initializations to all of our static members. All of our counters should be initialized to 0 and our Conditions should be initialized on our one boat lock. After this our logic for begin is rather simple. Initialize all our threads, fork them off and then sleep until the simulation is over:

```
for(int i = 0; i < adults; i++)
    new KThread(new Adult()).fork();
for(int i = 0; i < children; i++)
    new KThread(new Child()).fork();
// Ensure our threads get to increment their counters
// This yield shouldn't last long
while((numberOfAdultsOnOahu + numberOfChildrenOnOahu) < (adults + children))
    KThread.yield()
boatLock.acquire();
simulationStart.wakeAll();
simulationOver.sleep();
while(numberOfAdultsOnOahu + numberOfChildrenOnOahu > 0)
    simulationOver.sleep()
boatLock.release()
```

As for our AdultItinerary logic, our job is actually pretty simple. The one thing to keep in mind is that our Adults should now row across to the other shore unless there is only one child left on Oahu with them. This is so that we can ensure that there is at least one child on the other side, who can row back and pick up the remaining child. If they left when there were no children on the other side the Adult would row across and have to row back to bring the boat back. Therefore our Adult itinerary looks like this:

```
boatLock.acquire();
numberOfAdultsOnOahu++; //Increment our Adults on Oahu to start
simulationStart.sleep();
while(numberOfChildrenOnOahu != 1 || !boat.isOnOahu())
    okForAdultToLeaveOahu.sleep()
boat.load(adult);
boat.rowToOtherSide()
if(numberOfChildrenOnOahu + numberOfAdultsOnOahu > 0)
    okForChildToLeaveMolokai.wake()
boatLock.release()
```

Since we just left Oahu, it's acceptable for us to query the remaining number of people on Oahu as we do above for the adults because this is information they could easily obtain as they are leaving.

Our ChildItinerary logic is slightly more complicated. We have to be mindful of 3 cases while we are on Oahu: The case that we are the only child with no adults, the case we are the only child with remaining adults, and the case that there are multiple children remaining on Oahu. If we are the only child on Oahu, with no adults, we get in the boat and row to the other side. If we are the only child with adults still there, we wake an adult telling them it's okToLeaveOahu. We then sleep until the boat arrives back on Oahu and reevaluate the situation. Finally, in the other case we load the boat with the two children (the first getting in and then sleeping on *childDeliveredToOtherSide* and the other getting in and rowing away and then waking on that condition) and send them over. In this same case, once we arrive on Molokai, we check the number of people on Oahu and if there were people when we left, continue, otherwise we wake on simulation over and break out of our loop. On the other hand of all this if we are on Molokai, we check to see if there are people on the other shore (information which can easily be ascertained from the last people to arrive on Molokai) and if so we load a child into the boat and row across to the other side. Our child logic looks like the following:

```
boatLock.acquire()
while(true)
    if(child.isOnOahu)
        while(!boat.isOnOahu)
            boatOnOahu.sleep()
        if(numChildrenOnOahu == 1 && numAdultsOnOahu == 0)
            boat.load(child)
            boat.rowToOtherSide()
            simulationOver.wake()
            break
        else if(numChildrenOnOahu == 1 && numAdultsOnOahu > 0)
            okForAdultToLeaveOahu.wake()
            boatOnOahu.sleep()
            continue
        else
            if(boat.getChildCount == 1)
                boat.load(child)
                boat.rowToOtherSide()
                childDeliveredToOtherSide.wake()
            else
                boat.load(child)
                childDeliveredToOtherSide.sleep()
            if(numChildrenOnOahu + numAdultsOnOahu == 0)
                simulationOver.wake()
                break;
    else
        while(boat.isOnOahu)
            boatOnMolokai.sleep()
```

```
        if(numberOfChildrenOnOahu + numOfAdultsOnOahu > 0)
            boat.load(child)
            boat.rowToOtherSide()
        else
            simulationOver.wake()
            break;
boatLock.release()
```

Testing Plan

In order to test that our logic actually works we can write a *selfTest()* method which calls *begin()* with various permutations of adults and children and watch the output of the methods to ensure that they follow a valid combination of BoatGrader calls to row our children/adults back and forth until completion and that it terminates when there are no more people on Oahu.

Design Doc Questions

1. Priority donation for Semaphores is a tricky subject because for Semaphores the waiting takes place until someone calls Semaphore.V() and at that point threads are woken up off the queue. The problem with this in order to give the donations to the thread which calls V() we have to know in advance which thread will call V() and then give it the donation and remove it when the actual call to V() happens, but we don't really have a way of knowing in advance which thread will call it, so donations in this sense will be tricky.

2. This solution can work, as long as you make sure that you wait for all the threads to be initialized and their counters to be incremented before you proceed with any logic. There is a potential for flaws in this approach if you immediately proceed with logic after you increment because it is possible that not all the threads will increment the counter before you begin performing conditional operations based off the values of these counters which early on include insufficient data.