Niels Joubert CS162-PC
Nadeem Laiwala CS162-PD
Eugene Li CS162-KI
William Wu CS162-KJ
Akhil Dhar CS162-KK
March 4, 2008
Computer Science 162

CS162 Design Doc Phase 1: Build A Thread System

Task One - KThread.join()

Join is a mechanism that allows one thread to suspend its execution and wait for a second thread to complete. When the second thread finishes, the first thread is allowed to resume execution. Said another way, when thread A calls thread B.join(), thread A will only continue execution when thread B finishes. Lecture 4.19 roughly describes this process as: Join() needs to place one thread into the waiting state and associate it with the thread it is waiting on. Upon completion, the joined thread runs its ThreadFinish() routine, where any threads waiting on this thread are woken up and placed on the ready queue.

```
/* This is a reference to the thread that called join on "this" thread. */
private ThreadQueue joinQueue =
ThreadedKernel.scheduler.newThreadedQueue(true);

/* If no other thread is currently joined on this thread, then
join() {
    if (status is statusFinished)
        return
    disable interrupts, save original interrupt status
    joinQueue.waitForAccess(currentThread)
    currentThread.sleep()
    restore original interrupt status
}

finish() {
    ...
        while((get thread using joinQueue.nextThread()) != null)
            thread.ready()
    ...
}
```

Correctness Constraints

- Join will put the calling thread to sleep.
- When a thread finishes, it will wake up all threads joined to it.
- A thread should not be able to call join more than once. This is handled by atomically putting the thread to sleep upon calling join.
- Calling join on a finished thread should have no effect on the calling thread.

Testing Strategy

To test join, we extend the selfTest() method inside KThread. We created a JoinTest class with several testing methods that is run sequentially by the main thread:

testSimpleJoin:

This tests the simple case of one thread joining on another, and that the order of execution between the two is correct. It also checks that joining on a finished thread has no effect.

```
A global variable x is set to 0.
The main thread spawns a new thread, and joins on it.
This new thread asserts that x is zero, increments it by one, and exits.
the main thread tries to join the now-finished thread 2 more times. This
should return immediately.
The main thread checks that x == 1 right after the join statement.
```

testSequentalJoins

This tests several important join features - 4 threads sequentially join on each other, each one checking that the thread that joined on it is in a blocked state. The order of execution is checked through 4 variables that are incremented in a cascaded fashion. This ensures that join changes thread state correctly and that the order of execution is correct.

```
    integers a though d are set to zero.
    thread t2 is spawned by the main thread and joined to.
    thread t2 spawns thread t3 and joins to it.
    thread t3 spawns thread t4 and joins to it.
    thread t4 asserts that t3 is blocked, increments a and finishes.
    thread t3 wakes up, checks that a is increments and that t2 is blocked, and
finishes by incrementing a and b.
    thread t2 wakes up, checks that a and b is incremented, and finished by
incrementing a, b and c.
    the main thread wakes up, checks that a, b and c is correctly incremented. If
they are, the test was successful.
```

Task Two - Condition Variables using Interrupts

Condition variables allow threads to run critical sections only when the state of a protected resource is in accordance with their needs. For a critical section, a lock must be first be acquired to allow the thread to check the state of a resource to be examined. If this state is found to be different from what the thread needs, it can use a condition variable to release the lock on the protected section with the guarantee that it will be placed back on the ready queue and receive the lock when the state of the resource satisfies the condition placed on it by the thread in question.

We need a way to keep track of all of the threads that are waiting on this condition variable, so we use the scheduler to create a thread queue.

```
ThreadQueue waitQueue = ThreadedKernel.scheduler.newThreadQueue(true);

sleep() {
    assert that the current thread holds the lock for this condition variable
    save original interrupt state, disable interrupts for atomicity
    add yourself to the wait queue
    release conditionLock
    Kthread.sleep()
    restore original interrupt state for deadlock prevention
    acquire conditionLock
}

wake() {
    assert that the current thread holds the lock for this condition variable
    disable interrupts for atomicity
    if the wait queue is not empty:
        take the first thread off wait queue
        place that thread on ready queue
    enable interrupts
}

wakeAll() {
     assert that the current thread holds the lock for this condition variable
     disable interrupts for atomicity
     while the wait queue is not empty:
         take the next thread off wait queue
         place that thread on ready queue
     enable interrupts
 }
```

Correctness Constraints

- The thread calling a condition variable must have the lock that is associated with the condition variable.
- Upon waking up, the thread must have a lock to the protected resource again.
- The threads waiting on the condition variable should not consume CPU cycles.

Testing Strategy

To test this condition variable implementation, we extend the selfTest() method inside ThreadedKernel. We create a condition variable (testCondition), an associated lock (testLock) and several threads that act on this condition variable.

This tests interleaved calls of the Condition2 sleep(), wake() and wakeAll() methods and ensures that the order of execution between the threads is correct.

The main thread spawns a new thread, T1.
T1 acquires testLock and sleeps on testCondition.
T2 acquires testLock and sleeps on testCondition.
T3 acquires testLock and sleeps on testCondition.
T4 acquires testLock and wakes one thread sleeping on testCondition.
T5 acquires testLock and wakes all threads sleeping on testCondition.
T6 acquires testLock and sleeps on testCondition.
T7 acquires testLock and sleeps on testCondition. Upon waking up, T7 sleeps on testCondition again. (It should already have the lock required to sleep.)
T8 acquires testLock and wakes all threads sleeping on testCondition.

Here, we check that:
a) Waking up a thread moves it from the wait queue to the ready queue but does not run it immediately.
b) Upon waking up and returning to "running" state, a thread waiting on a condition variable has the associated lock again.
b) Threads waiting on a condition variable are not "running" (using CPU cycles) until they are pulled off the ready queue.

Moreover, we can run an additional test to ensure that a thread (that should fail on the assertion) cannot call a condition variable without having the associated lock.

T9 acquires testLock and sleeps on testCondition.
T10 wakes all threads sleeping on testCondition, without first acquiring testLock.

Task Three - Alarm

Threads call the waitUntil method to suspend their execution until at least the given amount of clock ticks have elapsed. To accomplish this task, we associate threads with their relative wait time, and store a pointer to each thread. Since multiple threads can be waiting, we create a queue inside the Alarm class to store information about waiting threads. Upon calling Alarm.waitUntil, the caller thread will be placed on this queue and relinquish the CPU.

We define a wrapper class that holds a reference to a thread, and a timer value to indicate when this thread desires to be woken up. To ease the process of waking up threads, we extend this class as a Comparable. This allows us to exploit the natural ordering of time by sorting waiting threads according to the time stamp by which they need to be placed on the ready queue. The wrapper class follows this pseudocode:

```
class KThreadWaiter implements Comparable<KThreadWaiter> {

    KThread myThread; //reference to thread this wrapper manages
    long wakeUpAt;

    constructor (KThread thread, long wakeUpAt)
        saves a reference to the given thread and its respective wakeUp time

    compareTo(KThreadWaiter o) throws ClassCastException
        if this.wakeUpAt > o.wakeUpAt
            return 1
        else if this.wakeUpAt == o.wakeUpAt
            return 0
        return -1;

    getWakeUpAt()
        return wakeUpAt

    getKThread()
        return myThread

}
```

We use a standard Java PriorityQueue data structure to store waiting threads. PriorityQueues sorts objects according to the Comparable interface they publish. The beauty of wrapping threads in custom waiter objects becomes apparent in the waiting and interrupt methods, where the data structure significantly reduces the amount of code needed to find threads that should be placed on the ready queue. Thus, we add the following variable to the alarm class:

```
private PriorityQueue<KThreadWaiter> waitUntilQueue
```

The thread that calls waitUntil gets wrapped inside one of these KThreadWaiter objects and placed in a queue. The thread is then put to sleep, which relinquishes the CPU and changes its state to "Blocked". The waiter object will be sorted against other waiting threads when placed into the queue.

```
waitUntil (long x):
    save original interrupt state, disable interrupts
    wait_time = current_time + x
    create wrapper object KThreadWaiter with currentThread and wait_time
    add new KThreadWaiter to queue
    currentThread.sleep()
    restore original interrupt status
```

Timer Interrupts occur regularly, and allow Alarm to inspect the waiting queue of threads and possibly place threads back on the ready queue. This is easily accomplished through the choice of data structure and a simple looping mechanism:

```
timerInterrupt() {
    find current time through Machine.timer().getTime()
    loop while queue has a non-null KThreadWaiter object with getWakeUpAt <
current time
        pop object off queue
        place KThread wrapped inside object on ready queue using
getKThread.ready()
    yield current thread
}
```

Correctness Constraints

- A thread should not consume CPU cycles until atleast its waitTime has elapsed.
- Any number of threads should be able to wait for any given wakeUp time.
- A thread should not bypass other scheduling routines to get CPU time when it gets woken up by the Alarm.
- When a timer interrupt occurs, all threads whose waitTime has elapsed must be placed on the ready queue.

Testing Strategy

Simple Test (Test1):
    This test is designed to test the simple case where the main thread sleeps for 1500 ticks, wakes up and checks that it has slept for longer than that value.

Multiple Threads sleeping for same amount of time (Test2):
    This test is designed to make sure we handle the value given to waitUntil correctly, and that multiple threads can sleep on alarm.
    The main thread forks another thread, that sleeps for 1500 ticks. The main thread then sleeps for the same value - 1500 ticks.
    Upon waking up, each thread checks that it has been asleep for more than 1500 ticks. Main also checks to make sure the other thread has completed.

Multiple Threads sleeping for different amount of times (Test3):
    This test is designed to glass-test the inner workings of Alarm. It does this by checking several aspects of the Machine's state:

```
    - Tests to make sure the waitUntilQueue has the right ordering (as part of
the correctness constraints)
    - Tests that the right amount of threads are on the waitUntilQueue at all time
    - Tests to make sure all the threads gets woken up
    - Tests that the threads are woken up reasonably early after their time has
expired

    This is achieved by forking 3 threads from the main thread. These 3 sleeps
for different durations, and checks the duration they slept on waking.
    The main thread checks the ordering and size of the queue by popping off
the head of the queue, and checking for the correct size depending
    on which thread is currently at the head of the queue.
    The main thread keeps doing this and yielding while the 3 threads are not
completed.
    If the duration of the longest wait has expired for more than 1000 ticks, we
also raise an error to indicate that the threads should all be done by now.
    To enable all this interaction, we heavily rely on the new assertions written in
Lib, and explained in our section on Testing at the bottom of this document.
```

Task Four - Communicator and one-word messages

The overall summary of a communicator is that speakers may use the communicator to speak a message and listeners may use the communicator to listen to a message. If a speaker wishes to speak but a listener is unavailable, the speaker will wait until a listener is available before speaking. Similarly if a listener wishes to listen but no speaker wishes to speak, the listener will wait for a speaker to arrive before listening.

In terms of the algorithm, the overall view is that a speaker transmits a message to a listener via a bounded buffer of size one. Each time a speaker is called, the speaker checks to see if there is an element in the buffer. If there is, then the speaker sleeps. If there isn't, then we add the speaker's word to the buffer. When a listener comes along, it checks to see if there is an element in the buffer. If there is, then it pops the element off, sets a boolean flag to true saying that the speaker has been heard, signals the speaker to awaken, and finally returns with the communicated word. If there is no element in the buffer, the listener sleeps until it is awakened by a speaker that has placed an element into the buffer.

The following psuedocode shows how we accomplish the aforementioned description.

```
class Communicator {

    private Lock commLock;
    private Condition2 canSpeak, canListen, hasRead;
    private LinkedList<Integer> buffer;
    private boolean hasBeenRead;

    Communicator() {
        commLock = new Lock();            //instantiate some lock object
        canSpeak = new Condition2(commLock);
        canListen = new Condition2(commLock);
        hasRead = new Condition2(commLock);
        buffer = new LinkedList<Integer>();
        hasBeenRead = false;
    }

    speak(int word) {
        commLock.acquire();              //lets make this function act atomically
        while (buffer is full)
            canSpeak.sleep();
        buffer.add(word);                //where word is the word that is to be
communicated to the listener
        canListen.wake();
        while (hasBeenRead is false)
            hasRead.sleep();
        hasBeenRead = false;
        canSpeak.wake();
        commLock.release();              //lets make this function not act
atomically anymore
    }

    listen() {
        commLock.acquire();              //lets make this function act atomically
        while (buffer is empty)
            canListen.sleep();
        buffer.pop();
        hasBeenRead = true;
        hasRead.wake();
        commLock.release();              //lets make this function not act
atomically anymore
        return the word you popped off;
    }

}
```

Correctness Constraints

- Both listen() and speak() must act synchronously in an atomic fashion.
- There can be multiple speakers and listeners utilizing the same communicator object.
- There should never be a time when both a speaker and a listener are waiting to act
- A single communicator should use exactly one lock

Testing Strategy

To test join, we extend the selfTest() method inside the ThreadedKernel class to include a Communicator.selfTest() function call. Inside Communicator.selfTest(), we instantiate six Communicator objects, where each one is used for six different testing strategies. Specifically inside Communicator.selfTest(), we use these Communicator objects in conjunction with multiple threads - that we spawn - that each have their own runnable to run. Below are our six Communicator objects with their respective testing strategies:

Test 1 - Standard speaker then listener: This test was designed to test the standard order of speaker/listener function calls; first the speaker is called, then the listener is called.

```
Create Communicator 1
Create new runnable called r1 {
    run() {
        communication1.speak(1)
    }
}
Create new runnable called r2 {
    run() {
        communication1.listen()
    }
}
create new KThread running r1
create new KThread running r2
```

Test 2 - Listener then speaker: This test was designed to test listener being called first, then the speaker.

```
Create Communicator 2
Create new runnable called r3 {
    run() {
            communication2.listen()
    }
}
Create new runnable called r4 {
    run() {
            communication2.speak(2)
    }
}
create new KThread running r3
create new KThread running r4
```

Test 3 - Multiple listeners then speaker: This test was designed to test the fact that multiple listeners would exist in one thread, and from another thread, speak would be called once only - leaving waiting listeners.

```
Create Communicator 3
Create new runnable called r5 {
    run() {
        communication3.listen()
        communication3.listen()
    }
}
```

```
    Create new runnable called r6 {
        run() {
            communication3.speak(3)
        }
    }
    create new KThread running r5
    create new KThread running r6
```

Test 4 - Multiple speakers then listeners: This test was designed to test the fact that multiple speakers would exist in one thread, and from another thread, listen would be called once only - leaving waiting speakers**.**

```
    Create Communicator 4
    Create new runnable called r7 {
        run() {
            communication4.speak(4)
            communication4.speak(5)
        }
    }
    Create new runnable called r8 {
        run() {
            communication4.listen()
        }
    }
    create new KThread running r7
    create new KThread running r8
```

Test 5 - Multiple speakers and multiple listeners: This test was designed to test the fact that multiple speakers would exist in one thread while multiple listeners would exist in another thread.

```
    Create Communicator 5
    Create new runnable called r9 {
        run() {
            communication5.speak(6)
            communication5.speak(7)
            communication5.speak(8)
            communication5.speak(9)
            communication5.speak(10)
        }
    }
    Create new runnable called r10 {
        run() {
            communication5.listen()
            communication5.listen()
            communication5.listen()
            communication5.listen()
        }
    }
    create new KThread running r9
    create new KThread running r10
```

Test 6 - Multiple speakers, and multiple listeners in independent threads: This test puts multiple speakers in one thread, and has multiple threads that each have an independent listener.

```
    Create Communicator 6
    Create new runnable called r11 {
        run() {
            communication6.speak(11)
            communication6.speak(12)
            communication6.speak(13)
        }
    }
    Create new runnable called r12 {
        run() {
            communication6.listen()
        }
    }
    Create new runnable called r13 {
        run() {
            communication6.listen()
        }
    }
    Create new runnable called r14 {
        run() {
            communication6.listen()
        }
    }
    create new KThread running r11
    create new KThread running r12
    create new KThread running r13
    create new KThread running r14
```

Task Five - Priority Scheduling

The ThreadedKernel has a single scheduler associated with it as a class variable. This scheduler is used to create the queues used throughout the kernel to line up threads waiting for access to some resource (CPU, lock, semaphore). As an example, the first KThread uses the ThreadedKernel's scheduler to create a new thread queue called readyQueue though scheduler.newThreadQueue(). We need to implement PriorityScheduler (and the inner classes PriorityQueue and ThreadState) so that calls to the queue returned by nextThread() will return the highest priority thread for the given resource according to the priority schedule paradigm. The PriorityQueue needs to maintain significant state to return the ideal thread.

Each KThread has a schedulingState reference (which is always dynamic type ThreadState) to keep track of the information needed to compute its priority. This state is changed through the Scheduler itself, and its methods that work on thread priority. Each ThreadQueue, being an inner class of the PriorityScheduler, changes the scheduling state of threads as it moves between queues and resources, and uses this state to calculate the appropriate positions of threads on queues.

Our solution treats Resources (and the PriorityQueue that is connected to them) and Threads (by their ThreadState variable) as nodes in a tree structure. Figure 1 elaborates on this design. Queues are owned by threads and have threads waiting on them. Threads own queues and wait on queues. This arrangement naturally gives rise to the alternation of the type of nodes between consecutive levels in the tree hierarchy. There are two pointers between each level, and maintaining the tree is mainly managing these pointers during all the possible state transitions that queues and threads can undergo. Priority can then be calculated as the maximum of your own priority and the total donation received from all the threads below a specific thread (passed up through the queues owned by and waited on by different threads).

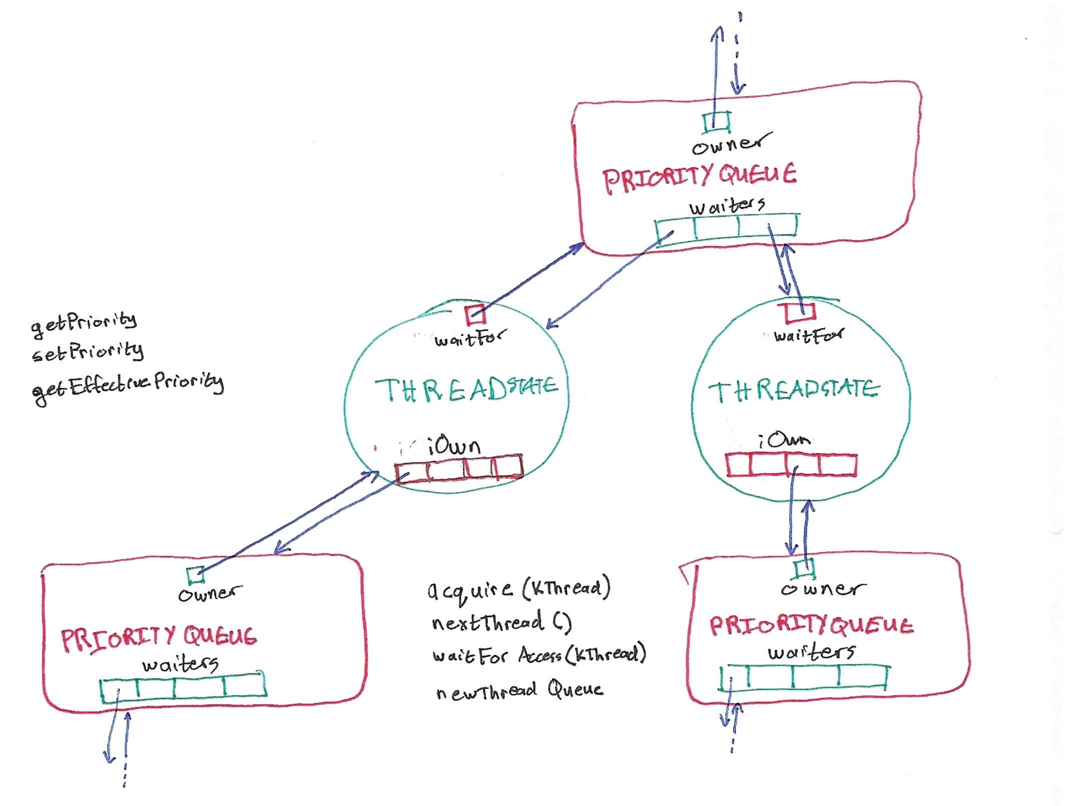

Figure 1 - The Tree Structure our PriorityScheduler builds.
Threads are green, Queues are red

Data structures needed to build this tree:

PriorityQueue:
ThreadState owner;
ArrayList<ThreadState> waiters;
int receivedPriority;

ThreadState:
PriorityQueue waitFor;
ArrayList<PriorityQueue> iOwn;
int receivedPriority;

Priority Scheduling State Changes

We identified 4 possible state changes that a thread can undergo that will change the structure of the tree:

A thread acquires a queue without waiting on the queue's waiters list

This gets triggered through calling priorityQueue.acquire(KThread).
It adds the queue to the thread's list of owned objects (thread.iOwn.add(waitQueue))
It sets the owner of the queue to be the current thread. (queue.owner = getThreadState(thread))

<u>Correctness Constraints:</u>
The queue must not have a current owner.
The thread must not be waiting on a different queue.

A thread gets placed on a queue as a waiter

This gets triddered through callind priorityQueue.waitForAccess(KThread).
It adds the current thread to the queue's list of waiters (queue.waiters.add(getThreadState(thread)))
It sets the thread's waitFor reference to the queue after asserting that the correctness constraints is upheld. (thread.waitFor = queue)

<u>Correctness Constraints:</u>
The thread must not be currently waiting on a different thread.

Thread's priority gets changed

This has no effect on the structure of the tree, but this will play a big effect when we handle priority donations.

Queue's nextThread gets called to release the current owner and pick a thread from the list of waiting threads, making it the new owner

The current owner gets released, which involves setting the owner pointer to null, and removing the queue from the owner's iOwn list.
We then pick the thread with the highest priority on the waiters list for the current queue, and remove it from the waiters queue.

We now acquire the queue for this new thread, causing the same state change as mentioned above.

Correctness Constraints:
If there are no waiters on the queue, simply release the owner and return null.

It is important that all operations on the three hierarchy maintains the overall structure of the tree.
Loops must be handled gracefully, which is the responsibility of our priority donation scheme.

Finding the next thread to run in the queue

This is a fairly simple problem now - simply select the thread with the highest effective priority off the queue. This will be made a linear operation over the amount of waiters on a queue by caching values appropriately. Since the older threads will be on the list of waiters first, we can traverse the list linearly, selecting the oldest thread with the highest priority.

```
if waiters is empty
    return
biggest = get first element off waiters list
for each waiter on the waiters list
    if current buggest's effective priority < waiter's effective priority
        biggest = waiter
return biggest
```

Calculating Effective Priority

To calculate the effective priority of a thread, we take the maximum value of all the donated priorities of threads waiting on it and its own native priority. The naive implementation of this method will search down the tree structure to find the total priority of all the threads below it. This is terribly inefficient, and we improve on it by caching the total priority donations received from the waiters on the queue or the queues a thread owns. By making your taking the maximum value, we ease the transition from queue member to resource owner. Since you will have the highest effective priority of all the members of a queue when you become the resource owner, you do not need to do any priority donation when the resource you are waiting on becomes one of the resources you own - your effective priority will not change!

Caching donations for efficiency

Rather than traversing the whole tree for every call to getEffectivePriority() we implemented a two-level caching scheme. Each thread keeps a private variable of the priority it has received from all the queues it owns, and it keeps this variable updates as it moves through the above-mentioned state transitions. Each queue also keeps a private variable of the priority that has been donated up to it from all the threads waiting on it. This value is also kept up-to-date as threads move through the above-mentioned state transitions.

Methods needed for caching and donating priorities

ThreadState (Managing ThreadState's receivedPriority)

getEffectivePriority()

Returns the effective priority of a thread, which is the maximum of its received and actual priorities.

```
int getEffectivePriority()
          return Maximum value of receivedPriority and priority
```

pullDonationsUpToMe() *(This function recurses upward as necessary)*

Resets my receivedPriority by recalculating the priority donations from all the resources i own.
This will send up donation to whatever im waiting on if my effective priority changes.

```
 void pullDonationsUpToMe()
          int tempP = 0
          int currentEP = getEffectivePriority()

          for each donator queue in iOwn
             tempP = Maximum of tempP and
donator.getDonation()
          receivedPriority = Minimum of tempP and
priorityMaximum

          if the effective priority changed and we are waiting on
a thread
             sendDonationUpToWaitFor();
```

sendDonationUpToWaitFor() *(This function assists in the upward recursion of pullDonationsUpToMe())*

Triggers the queue this thread is waiting on to recalculate the donation it received from all the threads that waits on it, since the donation of this thread changed.

```
void sendDonationUpToWaitFor()
          if we have a waitFor
                waitFor.recalculateDonation();
```

PriorityQueue (Managing PriorityQueue's receivedPriority)

getDonation()

Returns the total donations received from all threads waiting on this queue.

If the queue does not transfer priority, this always returns 0.

```
int getDonation()
          if we do not transfer priority
             return 0;
          return receivedPriority;
```

recalculateDonation() *(This function recurses upward by calling pullDonationsUpToMe() on the owner)*

Resets the queue's total receivedPriority to the maximum effective priority of all the threads waiting on it.
If this value changes in the calculation, the owner of the queue is notified that the total donation changed, and the owner will pull up all donations and recalculate its receivedPriority. On first glance this can be improved by only looking at the new priority of this queue against the current effective priority of the owner, but this case fails if a thread below loses priority and it was the thread that caused the owner's high priority in the first place. Thus, the owner needs to recalculate its receivedPriority, which is a linear operation on the amount of queues (and thus resources) it owns.
If the queue does not transfer priority, this returns immediately without performing any operations.

```
void recalculateDonation() {
          if we do not transfer priority
             return;

          int currentEP = get the current donation we make
          int tempP = 0;
          for each donator on the waiters queue
             tempP = Maximum value of tempP and
donator.getEffectivePriority()
          receivedPriority = take the minimum of tempP and
priorityMaximum

         if the donation changed and we have an owner
            owner.pullDonationsUpToMe();
```

Priority Scheduling State Changes with Priority Donation

We can now write the full implementations of the state changes, taking into account when to

A thread acquires a queue without waiting on the queue's waiters list

You can only acquire a queue when you already have the highest priority on it, or if the queue is empty. Thus no recalculation of your effective priority is needed here!

```
PriorityQueue's acquire(KThread thread)
        Call the thread's threadstate's acquire function.
        owner = thread's threadstate
```

```
ThreadState's acquire(PriorityQueue waitQueue)
        set waitFor to null
        add this queue to your iOwn list of queues you own
```

A thread gets placed on a queue as a waiter

Since adding a new thread as a waiter to a queue can change the total priority donated to the queue this thread is blocking on, we need to call recalculateDonation() after adding the given thread to our list of waiters.

```
PriorityQueue's waitForAccess(KThread thread)
        add this thread to the waiters list of threads blocked on this
queue.
        call recalculateDonation().
        call  waitForAccess(this) on the given thread's threadstate
```

```
ThreadState's waitForAccess(PriorityQueue waitQueue)
        set our wairFor to point to the given waitQueue
```

Thread's priority gets changed

If a thread's priority gets manually changed, we need to recurse upwards and recalculate the donation the queue we're waiting on gets. If this changes, the queue's owner needs to recalculate its receivedPriority, which could possibly change and force a recursion upwards to the queue it waits on, and so forth. Luckily, this is easily accomplished by the methods we've already written, namely recalculateDonation() on the queue we're waiting on, which will recurse with pullDonationsUpToMe() in its owner in the case that effective priorities is affected.

```
ThreadState's setPriority(int priority)
        if the new priority is the same as the old
            return
        set our priority to this new value

        if we are waiting on a queue
          waitFor.recalculateDonation()
```

Queue's nextThread gets called to release the current owner and pick a thread from the list of waiting threads, making it the new owner

Since nextThread is a combination of releasing the current owner, removing the new owner from the waiters list and letting it acquire the lock, we can use those function calls we already designed. All we need to remember is to recalculate the donations the queue receives after we removed the selected thread from the waiting queue. This is a linear time calculation for the amount of waiters on the queue, since we cache each thread's effectivePriority as a constant time operation.

```
PriorititiyQueue's nextThread() {
          call owner.release() to remove the current owner if owner is
not null.
          next = calculate the next thread by calling
pickNextThread();
          remove next from our waiting queue
          recalculateDonation()
          call acquire(next) to make the new thread the owner
```

```
ThreadState's release(PriorityQueue waitQueue)
          remove the waitQueue from our iOwn list
          recalculate our receivedPriority by calling
pullDonationsUpToMe();
```

```
PriorityQueue's pickNextThread()
          biggest = Select the first thread on the waiters list
          for each waiter on the waiters list
             if biggest.getEffectivePriority() <
waiter.getEffectivePriority()
                 biggest = waiter;
         return biggest, the thread with the highest priority that was
first on the list
```

Correctness Constraints

- Only one scheduler should ever exist.
- A thread state should only be changed with calls to the scheduler.
- A thread's priority should be dependent on threads that are waiting on the resources it owns.
- Donations from waiting threads should only be valid as long as the owner thread has a lock on that resource.
- Queues should be resorted each time a members' priority changes.
- A thread can only ever be waiting on a resource or owning that resource.

Testing Strategy

We wrote a significant set of selfTest cases that creates queues and threads, sets up donation trees and steps through joins, locks and moving through a queue with

nextThread.
Here is the test cases we wrote. For more details, please see the PriorityScheduler class.

Unit Test the Priority Scheduler and Thread State:

    * - Tests Initial Blank Values
    * - Tests Adding One Thread to Queue
    * --- Tests Null Owner
    * --- Tests ThreadState's waitFor
    * --- Tests Queue's waiters
    * - Tests Acquiring Queue
    * --- Tests Queue's Owner
    * --- Tests Threadstate's iOwn
    * --- Tests unchanging effective priority
    * - Tests Waiting on Queue that has an owner
    * --- Tests unchanging owner
    * --- Tests ThreadState's waitFor
    * --- Tests Queue's waiters
    * - Tests Queue's nextThread
    * --- Tests old owner's waitFor and iOwn
    * --- Tests new owner's iOwn and waitFor
    * --- Tests queue's owner variable
    * --- Tests queue's waiters list

Test simple donation

    * - Tests Setting Up Thread Priorities
    * - Tests equality of Effective and Normal priorities under initial setup
    * - Tests high priority thread waiting on queue with low priority owner
    * --- Tests effective priority versus real priority of owner
    * --- Tests high priority's priority invariance
    * - Tests nextThread() changing priorities
    * --- Tests old owner losing donated priority
    * --- Tests old owner's real priority invariance

Test multiple waiters on a queue

    * - Tests Correct Effective Priority with multiple threads on queue
    * - Tests Correct nextThread() for ALL 5 THREADS ON QUEUE
    * --- Tests highest priority next thread
    * --- Tests oldest thread with same priority first
    * - Tests Correct order of nextThread() for 4 equal-priority threads

Test join

    * - Tests Correct Effective Priority with joining one thread on another.
    * - Tests that the highest priority thread gets run first!

Task Six - Boating between Oahu and Molokai

In this task, we simulate a group of Hawaiian children and adults that wish to travel from one island to the next. We need to create independent threads containing their appropriate itineraries, which are dependent on their adult or child status. We can do this by creating child and adult Runnable objects:

```
for (the number of desired children):
    create child Runnable object:
        run() method calls childItinerary()
    create a new thread with child Runnable
    fork the thread
for (the number of desired adults):
    create adult Runnable object:
        run() method calls adultItinerary()
    create a new thread with adult Runnable
    fork the thread
```

A variety of data structures are required for us to fulfill this specification. These condition variables are all associated to one Lock that we create at the beginning of the begin() method. We will need the following objects:

```
Lock:
A general lock for the islanders to use.
A lock for the begin() method to check if the simulation is complete.

Condition Variables:
passengerWanted - this is used if the pilot of the boat is a child and wants to
wait for a second child to come on board
passengerReady - this is used for the second child passenger to signal to the
child pilot
adultOnOahu - this is used to wake or sleep adults that are on Oahu
childOnOahu - this is used to wake or sleep children that are on Oahu
childOnMolokai - this is used to wake or sleep children that are on Molokai
finished - this is used for the begin() method to check if the simulation is
complete

Integers:
numAdultsOnMolokai
numAdultsOnOahu
numChildrenOnMolokai
numChildrenOnOahu
numChildrenDone

Enumeration:
Location containing OAHU and MOLOKAI

Location:
boatLocation

Booleans:
boatHasPilot
boatHasPassenger
```

To successfully move everyone from Oahu to Molokai, the adults and children will need different itineraries. Our general goal is to move all the children to Molokai before moving any of the adults. After doing so, we then appropriately use children to bring the boat back and forth such that adults can reach Molokai. Upon reaching Molokai, we allow the the adult threads to completely finish.

The AdultItinerary can be as follows:

```
AdultItinerary() {
   Acquire the lock
   Increment number of adults on Oahu by 1
   Sleep on adultOnOahu condition variable

   Row to Molokai
   Wake childOnMolokai condition variable
   Release the lock
}
```

Note that the adult goes to sleep upon arriving at Oahu. Upon waking up, it rows to Molokai - therefore, we make sure that child only wakes an adult on the adultOnOahu condition variable when it arrives at Oahu from Molokai. We can safely assume that there will be a child at Molokai when an adult arrives, given the constraints that are placed on when adultOnOahu is called. We can see how the constraints are created in childItinerary.

The ChildItinerary can be as follows:

```
ChildItinerary() {
   Acquire the lock

   Create a boolean called done, and set it to false
   Set my location to be Oahu
   Increment number of children on Oahu by one.

   If there are two children on Oahu:
      Wake childOnOahu condition variable

   Sleep on childOnOahu condition variable

   While the boolean done is still false:
      While the location of boat is not at my location:
         If my location is at Oahu:
            Sleep on childOnOahu condition variable
         Else my location is at Molokai:
            Sleep on childOnMolokai condition variable

      If my location is at Oahu:
         If number of children on Oahu is greater or equal to two:
            If the boat does not have a pilot:
               Set boolean boatHasPilot to true
               Wake childOnOahu condition variable
               Sleep on passengerWanted condition variable

               Wake passengerReady condition variable
               Row to Molokai
               Change my location to Molokai
```

```
                    Set boolean boatHasPilot to false
                    Sleep on childOnMolokai condition variable
                Else if the boat does not have a passenger:
                    Set boolean boatHasPassenger to true
                    Wake passengerWanted condition variable
                    Sleep on passengerReady condition variable

                    Ride to Molokai
                    Change my location to Molokai
                    Set boolean boatHasPassenger to false
                    Wake childOnMolokai condition variable
                    If the difference between number of children on Molokai and
                        the number of children that are done:

                        Set boolean done to true
                        Increment the number of children done by one
                    Else:
                        Sleep on childOnMolokai condition variable

                Else the child should go back to sleep:
                    Sleep on childOnOahu condition variable


            Else child is the only child on Oahu:
                Row to Molokai
                Change my location to Molokai

                // Since we think the simulation is done, we will do the following
three lines:
                Acquire the finish lock for the finish condition variable
                Wake the finish condition variable for the begin() method
                Release the finish lock for the finish condition variable

                Set an alarm for the current child to wake up after 1000 ticks

        Else child is on Molokai:
            Row to Oahu
            Change my location to Oahu
            If the number of children on Oahu is greater than 1:
                Do not do anything.  We'll keep holding the lock, and while loop will
continue
            Else if number of adults on Oahu is greater than 0:
                Wake adultOnOahu condition variable
                Sleep childOnOahu condition variable

    Release the lock
}
```

We break down the child itinerary algorithm into different conditions based on the location of the child.

If the child is on Oahu, the child checks the number of children that are on Oahu. If there are two or more children on the island, the child will attempt to become a pilot or passenger. Otherwise, the child will go back to sleep on Oahu. There are a few condition variables that come into play for the pilot and passenger. Once a child become a pilot, the child will sleep for a passenger and change the boatHasPilot boolean to true. Thus, once it sleeps, the next child will see that a pilot is present and try to become a passenger instead. If that is successful, then the passenger will wake the pilot and the passenger himself will go to sleep. The pilot will then wake up, wake the passenger, and row to Molokai. This particular use of condition variables make it such that the pilot will row first before the passenger rides to Molokai.

If the child is on Molokai, then it rows back to Oahu. If there are other children on Oahu, the child goes through the algorithm checks for Oahu. Otherwise, if there are adults present on Oahu, wake one up and the child will go to sleep.

If only one child rows from Oahu to Molokai, we will check to see if the simulation is complete. We will acquire the finish lock, and wake the begin() method on the finish condition variable. Then we will set the child to sleep on an alarm for 1000 ticks. If we are indeed complete, the begin() method will return and the simulation will be over. If not, the begin() method will sleep again on the finish condition variable, and the child will eventually wake up after a minimum of 1000 ticks and resume the simulation.

When a child or adult row, we use a few helper methods that helps us keep track of the number of people on the different islands as well as the location of the boat. We define the helper methods as follows:

```
adultRowsToMolokai() {
    Decrement number of adults on Oahu by 1
    Increment number of adults on Oahu by 1
    Change boat location to Molokai
    Call AdultRowToMolokai method
}
adultRowsToOahu() {
    Decrement number of adults on Molokai by 1
    Increment number of adults on Oahu by 1
    Change boat location to Oahu
    Call AdultRowToOahu method
}
childRowsToMolokai() {
    Decrement number of children on Oahu by 1
    Increment number of children on Molokai by 1
    Change boat location to Molokai
    Call ChildRowToMolokai method
}
childRidesToMolokai() {
    Decrement number of children on Oahu by 1
    Increment number of children on Molokai by 1
    Call ChildRideToMolokai method
}
childRowsToOahu() {
    Decrement number of children on Molokai by 1
    Increment number of children on Oahu by 1
    Change boat location to Oahu
```

```
   Call ChildRowToOahu method
}
```

Correctness Constraints:

- In order for the boat to move between islands, exactly one child or adult must pilot the boat
- The boat can carry a maximum of two children or one adult at any given time
- Each person can only know the number of people on the island they are currently on
- No person is privileged to know the total number of people at the beginning
- No thread may busy wait at any time

General Testing Strategy

1) One approach to testing the robustness of the code is to use a JUNIT test suite for each component/class. By employing JUNIT test cases that primarily test each code segment independently by using assertions, we can compare the expected output of the code with the actual output. In the event that the expected output does not match the actual output, we can signal an error to the designer that will allow us to pinpoint the broken component. In effect, the interdependency of code segments can lead to obscure errors that can be corrected by employing this "divide and conquer" testing strategy. Assertions follow the following format:

```
...
assertTrue(<expected output>, <actual output>)
assertFalse(<expected output>, <actual output>)
...
```

How to ensure that some test did complete

We extended Lib to give us the functionality to ensure that certain pieces of code gets completed. This is done by keeping a list of known places, represented by the MustReachAssertion inner class in Lib.

Two static methods in Lib gives you the ability to assert that a certain block of code gets reached.

At the start of the code block, call getAssertReachedStart(name) to add a place in the list of places. When the code block has completed, call the reached() method on the object returned by the first call. If this method is not called, an assertion will fail on completion of TeadedKernel's selftest method. This is achieved by calling Lib.assertAllHasBeenReached(), which runs through the list of places and checks that each's state is "reached".

Design Questions:

1) *Why is it fortunate that we did not ask you to implement priority donation for semaphores?*

Implementing priority donation for semaphores would be difficult because a semaphore can have multiple owners and multiple waiters. Thus, there is a greater web of interaction between threads when working with semaphores, which makes the task of managing donations much more difficult. Specifically in our case, multiple owners would destroy the current abstraction we have for priority donations. The current tree abstraction we have will need to be changed into a graph, and it would be unfortunate to lose the clean abstraction.

Also, semaphores don't maintain a queue of who has initially incremented or decremented it. Assuming we couldn't change the underlying way that semaphores are currently implemented, it would be difficult to determine to whom the priorities should be donated to. Donations would also develop terrible run times due to all the graph traversals that must be made.

2) *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?*

It is not a good idea to only use an AdultsOnOahu counter as a metric for when to assume the boat problem is complete. The question poses the idea that the AdultsOnOahu counter will increase to the total number of adults before anything else happens. Then as adults move to Molokai and the counter reaches zero, we can use the communicator to indicate that the simulation is over. However, this makes the false assumption that every adult will have the initial opportunity to increment the counter. In the extreme case, let's imagine we have 10 adults and each is represented by a thread on the thread queue. They will increment the AdultsOnOahu counter when they first have access to some CPU time. But due to an ineffective scheduler, it could be the case that only one adult is ever given CPU time, while the other nine are starved. Thus, AdultsOnOahu will only ever get as high as 1 because the other nine threads never had a chance to increment the counter. And once that sole adult reaches Molokai and AdultsOnOahu is equal to zero, we will falsely proclaim that the simulation is over.