CSE 150 Project #1 Group # 3

Group Members: Ryan Hules, Yiying Jie, Chi Hong Kou, Haoxian(Andy) Huang, Jonathan Huang, Nathan Huizar

Task I:

Explanation: For this task, we are suppose to implement the join() function. The method should only be called once, the second call is not guaranteed to return anything.

Variables:
a queue for threads waiting for the join() method.
// private static ThreadQueue waitList;

One of the first things is to disable interrupts. And then we check the waitList queue, if it is null, we instantiate it.

Then, I checked if the status of the thread. If it is finished, return immediately, or else add to the list and put the thread to sleep.
Lastly, restore system interrupts.

I also modified the finish() method just to make sure everything works the desired way. Again, I check the waitList queue, if empty, instantiate it. I also created another thread, which is the threads in the waitList queue, and make sure all of them are executed.

(modified pseudocode)

```
public void join() {
        Lib.debug(dbgThread, "Joining to thread: " + toString());
        Lib.assertTrue(this != currentThread);

        disable interrupts;

        if waitList == null
                Instantiate the waitList queue;

        if this.status == statusFinished
                return;
        else
                put in waitList, wait for access;
```

```
            put thread to sleep;

        enable interrupt;

}

public void finish() {

        if waitList == null
            Instantiate the waitList queue;

        Initiate thread = currentThread.waitList.newThread();

        if thread != null {
            move to ready queue;
            thread = next ready thread;
        }
}
```

Test Case Idea: Run more cases regarding how many threads, run in a loop and see if all the threads execute properly.

Task II:

Implement condition variables in condition2. Implement condition variables directly.

Explanation: To implement the condition variables, the usage of locks are necessary, we decalred a condition lock. We also declared another variable, type KThread. It contains all the threads waiting, therefore called a waitQueue.

For the sleep() function, first released the condition lock. Then disable the interrupts, add the current thread to the waitQueue, make the current thread go to sleep. Lastly, restore interrupts and acquire the lock.

```
public void sleep() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        conditionLock.release();
        disables interrupt
        waitQueue ->>get currentThread
        currentThread -> sleep();
        enable interrupt
        conditionLock.acquire();
```

```
  }

public void wake() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());
        if (!waitQueue.isEmpty())
                disable interrupt
                waitQueue.removeFirst()
                thread.readyQueue? // get thread in a ready state
                enable interrupt
  }

Public void wakeAll() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());
        while(!waitQueue.isEmpty())
                Disable interupt
                waitQueue.remove();
                Enable interupt
}
```

Test case: replace cases where they used variables in the condition class using condition2 class (without using semaphores).

Task III:

Explanation: For waitUntil, we first disable system interrupts.
We actually defined another function outside of the two methods, which I named wakeAlarmThread, and it waits for the alarm thread, it contains the time when a thread is going to be awaken.

After creating a thread of type wakeAlarmThread, we then add the thread to the waitQueue and let it go to sleep next. Lastly, restore interrupt status.

For timeInterrupt, same thing, disable interrupts.

And then an iterator is used to grab the next thread that is ready to be awaken. And if it is ready, then we remove the thread from the waitQueue and put it on the ready queue. This is where the function wakeAlarmThread came in handy, because it returns the wake time of a thread, which can be use to check if the thread is ready to wake up.

Lastly, restore interrupts.

//Implementing Alarm by implementing waitUntil.

```
public void timerInterrupt() {
        long currTime = Machine.timer().getTime();
        //Need to check for ready threads in waiting queue
        //Thread is ready when the wakeTime is <= currTime
        //Check if thread is ready while waitQueue not empty
        //Wake if ready
        KThread.currentThread().yield();
    }

public void waitUntil(long x) {
        long wakeTime = Machine.timer().getTime() + x;
        //Get current thread and have it sleep for length of waketime, which is timer + x ticks
        //Put it into waiting queue then sleep.
        //Woken when interrupted
        while (wakeTime > Machine.timer().getTime()){
                KThread.yield();
        }
        Add this thread to the ready queue
}

// implement waiting for thread to wake up time
class wakeAlarmThread {
        long wake;
        KThread waitThread;

        public wakeAlarmThread (long wTime, KThread wThread) {
                this.wake = wTime;
                this.waitThread = wThread;
        }

}
```

Task IV: Simple Synchronized Messaging

Implement Communicator class with operations, void speak(int word) and int listen().

Explanation: While there is no word to be sent, there should not be an active speaker that is awake. Otherwise, the listener should be awake once word has been obtained. Likewise, if there is no word ready to be heard, there should not be an active listener awake. Otherwise, once the word has been received/heard, a speaker should be awoken and be ready to transmit another word.

```
public Communicator() {
```

```
        lock = new Lock();
        //Create lock conditions for speaker/listener in waiting queue
        Speaker = new Condition(lock)
        Listener = new Condition(lock)
    }

public void speak(int word) {
        Speaker gets lock, lock.acquired();
        Speaker ++  ← Speaker is available
        while(no word or no speaker is ready)
                Speaker.sleep();
        Word = true  ← flag to show that word is ready
        listen.wake(); ← wake up all the listeners
        Speaker.sleep();
        Speaker --  ← Speaker not available
        lock.release();


    }
public int listen() {
        listener gets lock, lock.acquired();
        while(no word is ready)
                Speaker.wake();
                listener.sleep();
        Word = false ← flag to show that word is unavailable after listener receives word
        Speaker.wake(); ← wake up speaker after receiving word
        lock.release();

        return word;
    }
```

Test Case Ideas: Make sure code works with multiple listeners and speakers, test whether or not listener waits for speaker and vice-versa. A combination of multiple speakers to one listener or the other way around.

Task V:

Implement priority scheduling;
need to change nachos.threads.RoundRobinScheduler to nachos.threads.PriorityScheduler;
Implement methods getPriority() and getEffectivePriority() and setPriority(); */

Explanation: Priority Scheduler will pick threads with the highest priority to run. In case of multiple threads with highest priority, the scheduler will pick a thread that has been there longest via FCFS basis. If there are more than one thread that has the same priority, the one would be chosen through FCFS basis.

An issue that may occur called priority inversion, where a high priority thread needs to wait for a lower priority thread. This is fixed through priority donation and in our code through getting the effective priority of a thread as opposed to regular get priority.
In get effective priority, we look for threads in our queue that have transferred priority, in which case a donation has occurred and so we must update effective priority of that thread. Pick next thread should always pick the thread in queue that has the highest priority and on FCFS for any ties.

// priority can decided based on memory requirements, time requirements or any other resource requirement

//The function nextThread() and pickNextThread() are within the PriorityQueue class and the rest is within the ThreadState class.

//picks the next thread after calling the pickNextThread function after checking certain conditions of our LinkedList

```
 public KThread nextThread() {
        Lib.assertTrue(Machine.interrupt().disabled());

        // Pick the next thread
        ThreadState nThread = this.pickNextThread();

        //check to see if the thread is empty
        if (nThread == null) return null;

        // Take out the next thread from the waiting queue
        this.waitThread.remove(nThread);

        // Call acquire to get thread
        this.acquire(nThread.getThread());

        //return the thread we just got
        return nThread.getThread();
    }


protected ThreadState pickNextThread() {

        // assure the smallest priority
        int nextPriority = priorityMinimum;

        // ensure the thread state is null
```

```
        ThreadState nextTS = null;


        for (ThreadState currThread : this.waitThread) {

                int currPriority = currThread.getEffectivePriority();

                // this will ensure threads have correct priority
                if (nextTS == null || (currPriority > nextPriority)) {

                        nextTS = currThread;

                        nextPriority = currPriority;
                }
        }

        return nextTS;
}


Int getPriority() {
// should simply return priority of thread
        return priority;   //return priority of a thread
}

public int getEffectivePriority() {

                        // check to see if the list is empty
                        if (this.have.isEmpty()) {

                                return this.getPriority();

                        }

                        // if it is not empty, change priority
                        else if (this.priorityChange) {

                                this.effectivePriority = this.getPriority();

                                for (PriorityQueue pq : this.have) {

                                        this.effectivePriority = Math.max(this.effectivePriority,
pq.getEffectivePriority());
```

```
                        }

                        this.priorityChange = false;
                    }

                    return this.effectivePriority;
                }


public void setPriority(int priority) {

        if (this.priority == priority)

                return;

        this.priority = priority;

        // force priority invalidation
        for (PriorityQueue pq : want) {

                pq.checkPriority();

        }

}


public void waitForAccess(PriorityQueue waitQueue) {

        this.want.add(waitQueue);

        this.have.remove(waitQueue);

        waitQueue.checkPriority();

}


Void acquire(PriorityQueue waitQueue){
//helps us remove threadstate in queue if called
        waitQueue.item = this;
        getEffectivePriority();
```

}

Test Case Ideas: Ensure priority schedule works with large amount of threads and that it doesn't take too long to pick one.

Task VI:

Explanation: This portion of the program in concerned with solving the boat problem (i.e. getting all the islanders from one island to the other). It operates on the following logic:

- End immediately if there are less than two children in the system, because the problem is unsolvable below that threshold, outside of edge cases in which there is one or less people
- Prioritize sending children to the destination island, unless there is only one child on the source island. In that case, prioritize sending adults to avoid infinite loops
- Prioritize sending children back to the source island

```
//now with 30% less child labor!
static boolean boatOnOahu;
static int cKnownOnOahu;
static int cKnownOnMolokai;
static int aKnownOnOahu;
static int aKnownOnMolokai;
static Lock l;
static Condition2 cWaitingInBoat;
static Vector<KThread> waiting;
static Condition2 cWaitingOnOahu;
static Condition2 cWaitingOnMolokai;
static Condition2 aWaitingOnOahu;
static Condition2 aWaitingOnMolokai;
public static void begin( int adults, int children, BoatGrader b )
   {
          // Store the externally generated autograder in a class
          // variable to be accessible by children.
          bg = b;
         l = new Lock();
          cWaitingInBoat = new Condition2(l);
          cWaitingOnOahu = new Condition2(l);
          cWaitingOnMolokai = new Condition2(l);
          aWaitingOnOahu = new Condition2(l);
          aWaitingOnMolokai = new Condition2(l);
          waiting = new Vector<KThread>();
```

```
        // Instantiate global variables here
        Int cKnownOnOahu = children;
        Int aKnownOnOahu = adults;
        Int cKnownOnMolokai = 0;
        Int aKnownOnMolokai = 0;
        Bool boatOnOahu = true;


        // Create threads here. See section 3.4 of the Nachos for Java
        // Walkthrough linked from the projects page.
        //create adult threads
        vector<KThread> AThreads = new vector<KThread>;
        Runnable a = new Runnable(){
                Public void run(){
                        adultItinerary();
                }
        };
        for (int i = adults; i > 0; i--) {
                KThread temp = new KThread(a);
                temp.setName("Adult on Oahu");
                aKnownOnOahu++;
                temp.fork();
        }
        //create child threads
        Runnable c = new Runnable() {
                public void run() {
                        childItinerary();
                }
        };
        for (int i = children; i > 0; i--) {
                KThread temp = new KThread(c);
                temp.setName("Child on Oahu");
                cKnownOnOahu++;
                temp.fork();
        }

  }
static void AdultItinerary()
  {
        l.acquire()
        While (not done)
                If !boatOnOahu
```

```
                    If there are no children on Molokai
                        if there's only one child in the system
                            Break to avoid infinite looping
                        If this thread isn't where the boat is
                            wake one that is
                        Otherwise
                            Row to oahu
                            aKnownOnOahu++;
                            aKnownOnMolokai--;
                        aWaitingOnOahu.sleep();
                    Otherwise
                        Wake up a child on molokai
                    aWaitingOnMolokai.sleep();
                Otherwise, if this thread isn't where the boat is
                    Wake one that is
                    aWaitingOnMolokai.sleep();
                Otherwise, if there are children
                    Send them first
                    aWaitingOnOahu.sleep();
                Otherwise
                    Row to molokai
                    aKnownOnMolokai++;
                    aKnownOnOahu--;
                    aWaitingOnMolokai.sleep();
        l.release();

}

static void ChildItinerary()
{
        l.acquire();
        while(not done)
                If !boatOnOahu
                    If this thread is on molokai
                        Row to Oahu
                        cKnownOnOahu++;
                        cKnownOnMolokai--;
                    Otherwise
                        Wake a thread that is
                    cWaitingOnOahu.sleep();
                Otherwise, if the boat is on Oahu and this thread isn't
                    Wake one that is
                    cWaitingOnMolokai.sleep();
```

```
            Otherwise, if there's nobody in the boat
                If this is the only child left on oahu
                    If there are any adults
                        Send them first
                        cWaitingOnOahu.sleep();
                    Otherwise, row to Molokai alone
                Otherwise
                    Get in the boat and wait for another child
                    cKnownOnOahu--;
                    cWaitingInBoat.sleep();
            Otherwise //there's already a child in the boat
                Get in boat and ride to Molokai
                cKnownOnOahu--;
                cKnownOnMolokai+=2;
                cWaitingOnMolokai.sleep();
        l.release();

    }
```

Test Case Ideas: run with only one child, to see if the program loops infinitely

Design Questions

#1 Why is it fortunate that we did not ask you to implement priority donation for Semaphores?

- Because of semaphores, we are unable to determine whether or not the last thread that acquires the semaphore is of the lowest priority or if another thread needs a "donation" based on their priority status. As of now, we can determine which thread is using what resources based on whose acquiring the lock.

#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?

- One counter alone wouldn't be sufficient to solve the problem, because you need to keep track of, at the very least, the number of children on Oahu as well.

Keeping track of only the adults on Oahu would not give enough information to be able to tell whether the problem has been solved or not.