Design Document for Project 1

Names: Raphael Cruanes, Dylan Baudhuin, Alex Sanchez Rubio, Eddie Lizaraburu, Leonardo Gil Rojo, Raphael Cruanes, Jeff Winters

Responsibilities: One person per task, including pseudo-code and design document

The Problem

Task 1: Server Access

- Completed

Task 2: KThread.join() (DYLAN)

Solution:
To account for only one Thread being allowed to join another Thread, a condition variable “joinCond” will keep track of whether or not a Thread has joined with the currentThread. If it is recorded, any subsequent attempts to join the currentThread will reference this structure and return an error.
To account for a Thread staying “blocked” until its parent is finished, it will be put to sleep if the joining Thread attempts to join and the parent isn't finished. Once the parent thread runs KThread.finish(), it will check the “joinCond” condition variable for any Thread that attempted to join with it, and then wake that thread up to continue execution.

Code:

```
    public static void finish() {
        Lib.debug(dbgThread, "Finishing thread: " +
currentThread.toString());
        Machine.interrupt().disable();

        Machine.autoGrader().finishingCurrentThread();
        Lib.assertTrue(toBeDestroyed == null);
        toBeDestroyed = currentThread;
        currentThread.status = statusFinished;

        //pseudo code:
        //Check if there is a thread recorded in our "joins" class.
        //If there is, wake it up (put it in the ready queue?)
        //If there isn't, do nothing.
        if(currentThread.joinCond != null){

            joinLock.acquire();
            currentThread.joinCond.wake();  // This is our ability to
wake up the thread that tried to join us.
            joinLock.release();
        }
        //------------
        // is this needed below?
        // boolean interrupt = Machine.interrupt().disable();

        sleep(); // after destruction it takes care of it right?

    }
    //…
    //…



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

```
            Lib.assertTrue(this != currentThread, "ERROR: Threads may not
join themselves."); // this is a thread can't join itself check
      // obtaining the lock prevents 2 threads from attempting to join at
the same time. I mean they can both run join but they will be forced to do
so atomically.
            joinLock.acquire();
            // boolean interupts = Machine.interrupt().disable();
            if(this.status == statusFinished){ //This is us checking if
the thread we are trying to join is already finished. "this" refers to the
thread we are trying to join.
                //do i put it in the ready queue? the instructions say "B
returns immediately from join without waiting if A has already finished."
                //If i disabled interrupts I should enable them before I
return.
                joinLock.release();
                // Machine.interrupt().restore(interupts);
                return;
            }

            Lib.assertTrue(this.joinCond == null, "ERROR: Another thread
has already tried to join this thread."); //This will return an error and
stop the rest from happening

            this.joinCond = new Condition2(joinLock); //record whos
trying to join in the thread we are trying to join's "joins" variable
            // Do i make the thread who is waiting status=blocked? No
sleep does that for me.
            this.joinCond.sleep(); //put the current thread to sleep, is
this the correct operation to make the thread wait inside of
KThread.join()?
            joinLock.release();
            //------------


        }
    //...
  private static Lock joinLock = new Lock();
  private Condition2 joinCond = null;
```

Task 3: Implement Condition Variable Atomicity (RAPH)

Design Document for Condition2 Implementation

The Problem: The goal is to implement a condition variable mechanism using interrupt enable and disable to ensure atomicity, instead of using semaphores. This implementation will provide the same functionality as the existing Condition class, which is based on semaphores. The condition variable allows threads to sleep and be woken up in a controlled manner while ensuring proper synchronization.

Condition Variables A condition variable is a synchronization primitive that allows threads to wait until some condition is met. It is associated with a lock, which must be held when performing operations on the condition variable. The three operations are:

1. **sleep()**: Atomically releases the associated lock and puts the current thread to sleep until another thread wakes it up.
2. **wake()**: Wakes up a single waiting thread, if any exist.
3. **wakeAll()**: Wakes up all waiting threads.

Implementation Details

API The Condition2 class provides the following methods:

- `Condition2(Lock conditionLock)`: Constructor that associates the condition variable with a given lock.
- `void sleep()`: Releases the lock and puts the calling thread to sleep until woken.
- `void wake()`: Wakes up at most one waiting thread.
- `void wakeAll()`: Wakes up all waiting threads.
- `void sleepFor(long timeout)`: Puts the calling thread to sleep for a specified duration or until woken.

Classes

1. **Condition2**
    - Stores a reference to the associated lock.
    - Maintains a queue of waiting threads.

- Implements thread waiting and waking using interrupt disable/enable for atomicity.
- **Header:**

```
public class Condition2 {
    private Lock conditionLock;
    private LinkedList<KThread> waitQueue;
    public Condition2(Lock conditionLock);
    public void sleep();
    public void wake();
    public void wakeAll();
    public void sleepFor(long timeout);
}
```

2. **Methods Implementation**

   - `sleep()`: Disables interrupts, adds the current thread to the wait queue, releases the lock, and puts the thread to sleep. Upon waking, restores interrupts and reacquires the lock.
   - `wake()`: Disables interrupts, removes a thread from the wait queue, and marks it as ready.
   - `wakeAll()`: Iteratively wakes all threads in the wait queue.
   - `sleepFor(long timeout)`: Uses `ThreadedKernel.alarm.waitUntil(timeout)` to put the thread to sleep for the given duration.
   - **Pseudo-code for sleep():**

```
void sleep() {
    assert conditionLock is held by current thread;
    disable interrupts;
    add current thread to waitQueue;
    release conditionLock;
    put thread to sleep;
    restore interrupts;
    acquire conditionLock;
}
```

Correctness Constraints

- A thread can only call `sleep()`, `wake()`, or `wakeAll()` if it holds the associated lock.

- Interrupts must be disabled during modifications to the wait queue to ensure atomicity.
- The woken thread must re-acquire the lock before resuming execution.
- `sleepFor(timeout)` should ensure that the thread wakes up either by timeout or an explicit `wake()` call.

Code:

```java
package nachos.threads;

import nachos.machine.*;

import java.util.LinkedList;

public class Condition2 {
    private Lock conditionLock;
    private LinkedList<KThread> waitQueue;

    public Condition2(Lock conditionLock) {
        this.conditionLock = conditionLock;
        this.waitQueue = new LinkedList<>();
    }

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

        boolean intStatus = Machine.interrupt().disable();

        KThread currentThread = KThread.currentThread();
        waitQueue.add(currentThread);

        conditionLock.release();
        currentThread.sleep();

        Machine.interrupt().restore(intStatus);
        conditionLock.acquire();
    }
```

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

    boolean intStatus = Machine.interrupt().disable();

    if (!waitQueue.isEmpty()) {
        KThread thread = waitQueue.removeFirst();
        if (thread != null) {
            thread.ready();
        }
    }

    Machine.interrupt().restore(intStatus);
}

public void wakeAll() {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());

    boolean intStatus = Machine.interrupt().disable();

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

    Machine.interrupt().restore(intStatus);
}

public void sleepFor(long timeout) {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());

    boolean intStatus = Machine.interrupt().disable();

    KThread currentThread = KThread.currentThread();
    waitQueue.add(currentThread);

    conditionLock.release();

    ThreadedKernel.alarm.waitUntil(timeout);

    Machine.interrupt().restore(intStatus);
    conditionLock.acquire();
```

```
    }
}
```

Task 4: Alarm (Alex & Jeff)

Initial Notes:

- Complete the implementation of the Alarm class
    - Implementing the waitUntil(LongX)
    - A thread calls waituntil(longX)
        - In order to suspend it's execution until wall clock time has advanced to at least now + x.
            - Method is useful for thread that operate in real time, such as blinking the cursor ounce per second
                - There is no requirement that threads start running immediately after waking up
                - Just put them on the ready queue in the timer interrupt handler after they have waited for at least the right amount of time.
                - **do not need any additional thread to implement waitUntil(); you need only modify waitUntil() and the timer interrupt handler methods.**
                    - waitUntil() itself, is not limited to being called by one thread: any number of threads may call it and be suspended at any one time. If the wait parameter x is 0 or negative, return without waiting (do not assert)

Explanation: This class is created in order for threads to sleep for a certain number of clock ticks. When the current thread calls the method waitUntil(long x), it calculates the int wakeTime and then gets placed into a priority queue imported from Java before sleeping. Within each interrupt from the timer, the class is checking if there's any waiting threads whose wakeTime has expired , if it puts it into a ready queue. This way the threads are scheduled correctly and run when ready.

Code:

```java
package nachos.threads;
import nachos.machine.*;
import java.util.PriorityQueue;
import java.util.Comparator;

/**
 * Uses the hardware timer to provide preemption, and to allow threads to
sleep
 * until a certain time.
 */
public class Alarm {
	/**
	 * Allocate a new Alarm. Set the machine's timer interrupt handler to
this
	 * alarm's callback.
	 *
	 * <p>
	 * <b>Note</b>: Nachos will not function correctly with more than one
alarm.
	 */
	public Alarm() {
		Machine.timer().setInterruptHandler(new Runnable() {
			public void run() {
				timerInterrupt();
			}
		});

		waitQueue = new PriorityQueue<WaitingThread>(new
WaitTimeComparator());
	}

	/**
	 * The timer interrupt handler. This is called by the machine's timer
	 * periodically (approximately every 500 clock ticks). Causes the
current
	 * thread to yield, forcing a context switch if there is another
thread that
	 * should be run.
	 */
	public void timerInterrupt() {
		boolean intStatus = Machine.interrupt().disable();

		long currentTime = Machine.timer().getTime();
```

```
            // Check if any threads need to be woken up
            while (!waitQueue.isEmpty() && waitQueue.peek().wakeTime <=
currentTime) {
                  WaitingThread thread = waitQueue.poll();
                  thread.thread.ready();
            }

            Machine.interrupt().restore(intStatus);

            // KThread.currentThread().yield();


            // same thing just a bit simple
            KThread.yield();
      }

      /**
       * Put the current thread to sleep for at least <i>x</i> ticks,
waking it up
       * in the timer interrupt handler. The thread must be woken up
(placed in
       * the scheduler ready set) during the first timer interrupt where
       *
       * <p>
       * <blockquote> (current time) >= (WaitUntil called time)+(x)
</blockquote>
       *
       * @param x the minimum number of clock ticks to wait.
       *
       * @see nachos.machine.Timer#getTime()
       */
      public void waitUntil(long x) {
            // If x is zero or negative, return immediately
            if (x <= 0)
                  return;

            boolean intStatus = Machine.interrupt().disable();

            long wakeTime = Machine.timer().getTime() + x;

            // Create a new waiting thread entry and add it to the queue
            WaitingThread waitingThread = new
```

```
WaitingThread(KThread.currentThread(), wakeTime);
            waitQueue.add(waitingThread);

            // Put the thread to sleep
            KThread.sleep();

            Machine.interrupt().restore(intStatus);
      }

      /**
       * Class representing a thread waiting to be awakened at a specific
time.
       */
      private class WaitingThread {
            KThread thread;
            long wakeTime;

            WaitingThread(KThread thread, long wakeTime) {
                  this.thread = thread;
                  this.wakeTime = wakeTime;
            }
      }

      /**
       * Comparator for sorting waiting threads by wake time.
       */
      private class WaitTimeComparator implements Comparator<WaitingThread>
{
            public int compare(WaitingThread a, WaitingThread b) {
                  if (a.wakeTime < b.wakeTime) return -1;
                  else if (a.wakeTime > b.wakeTime) return 1;
                  else return 0;
            }
      }

      // Queue of threads waiting to be awakened, sorted by wake time
      private PriorityQueue<WaitingThread> waitQueue;
}
```

Implementation output

```
alarmTest1: waited for 1220 ticks
alarmTest1: waited for 10090 ticks
alarmTest1: waited for 100180 ticks
```

- Implementation of code seems to be working correctly.

Task 5: Communicator (Eddie & Leo)

Requirements:

- Both speak and listen need conditions to wait for the word buffer
  - When the buffer is full speak will sleep
  - When the buffer is empty listen will sleep
- After speak places its word into the buffer it must declare the buffer full, signal a listener to wake up, then release it's lock
- After listen takes a word from the buffer it must declare the buffer empty, signal a speaker to wake up, then release it's lock

Communicator Variables:
Word being passed between speaker and listener (Buffer) = word (int)
Lock (private) = Lock
Boolean to check if buffer is full = wordToBeHeard (bool)
Speak condition = speakReady
Listen condition = listenReady
Wait condition = waitForListen

Explaination:
Speak function starts by acquiring a lock, then checks to see if the buffer is currently filled. While the buffer is filled the speak function goes to sleep, transferring it's lock and awaiting a signal from listen. Once done, the speak function sleeps in waitForListen until the listen() calls wake().

```
public void speak(int word) {

        lock.acquire();

        // while there is a word in the buffer
        while (wordToBeHeard) {

                speakReady.sleep();
        }
```

The buffer is empty when wordToBeHeard is false, in that case the speak function will continue to write it's word into the buffer, declare it full and release its lock:

```
        this.word = word;

        // notes that the buffer is full
        wordToBeHeard = true;

        listenReady.wake();
        waitforListener.sleep();


        lock.release();
```

For the listen function first it will acquire a lock and check to see if buffer is empty. If the buffer is empty then the listener will go to sleep waiting for the buffer to be filled, transferring it's lock.

```
public int listen() {

        lock.acquire();

        // while there is no word in the buffer
        while (!wordToBeHeard) {

        listenReady.sleep();

        }
```

Once the buffer is full, listen will take the word from the buffer, declare the buffer empty, signal any sleeping speakers, release it's lock and return the word to it's caller. It then wakes the speaker so it can finish.

```
    int wordToHear = word;


        wordToBeHeard = false;


        speakReady.wake();
        waitforListener.wake();


        lock.release();

        return wordToHear;
```

Task 6: Boat (Leo)

- The problems we face are the following:
    - Someone needs to bring the boat back from Molokai when it is brought to Oahu
    - It isn't useful to have the same group of people ride the boat to Molokai and then bring it back.
    - It probably isn't useful to have more people come back to Oahu than left.
    - There are two possible two-way trips that make sense then:
        - Two children go to Molokai and one returns to Oahu.
        - An adult goes to Molokai and then has a child pilot the boat back to Oahu.
    - We will definitely need to use both cases, since the second is the only one that gets an adult to Molokai, but the first is the only one that gets a child there.
- So a solution could be:
    1. Have two children travel to Molokai, then one return the boat to Oahu. This brings one child to Molokai.
    2. Repeat step 1 until there is exactly one child, and the adults, left on Oahu.
    3. An adult gets on the boat and travels to Molokai, where one of the children that was dropped off then takes the boat back to Oahu. There is now two children on Oahu. The children take the boat to Molokai, one child is dropped off, and the other returns to Oahu. This effectively transports one adult to Molokai.
    4. Repeat step 3 until all the adults are on Molokai.
    5. There is now one child left on Oahu, who can return to Molokai, completing the solution.

Let's take note of how different people should react to different situations given our above solution.

First, the people on Oahu:

- If there is more than one child on Oahu (step 1 and 3), we have to transport the children there:
    - Children try to board the boat. If there are two children on the boat, it sets off.
    - Adults do not try to board the boat.
- If this isn't the case, but there are adults on Oahu:
    - There would only be one child on Oahu. This child does not try to board the boat.
    - All adults try to board the boat and set sail immediately upon entering.
- If there is only one child and no adults on Oahu:
    - That child will try to board the boat and set sail.

Now, the people on Molokai:

- No matter what, one child pilots the boat back to Oahu, so:
    - Adults do not try to board the boat.

        - Children try to board the boat. If there is already a child in the boat, they do not try to board the boat.
    - **ONE Exception:**
        - The boat arriving could be carrying the last child, in which case it isn't necessary to send it back to Oahu.
            - This is the only case where exactly one child arrives in Molokai instead of two. We can observe who is in the boat, and have all the people in Molokai halt upon seeing exactly one child in the boat.

"Trying to board a boat:"

- To be clear, this would entail:
    - Checking whether the boat is on the same island as the person.
    - Check whether the boat can accommodate the person.
        - If either check fails, we can use a condition variable to have that thread sleep until the boat returns empty to the island that person is on. For example:

```
lock.acquire();
if(boat in Molokai || cannotFit || shouldNotBoard):
        cv.sleep()
…….
lock.release()

Void sendToOahu():
        ….. // Update variables here, not shown
        cv.wakeAll()
```

Let's identify critical sections that would require using a synchronization primitive.
In general, this would be anything that updates shared information such as the number of people on a given island, or the number of people on a boat.

- **Boarding the boat or the boat queue**
    - It may be the case that two people try to board the boat at the same time and end up breaking the conditions of the boat (that there can only be at most two children or one adult).
        - For example, with one child in the boat, another child could find there is enough space on the boat and begin boarding. The CPU could switch to a 3rd child, who also finds enough space on the boat (the 2nd child has still not boarded) and boards at the same time. The CPU goes back to the 3rd child, who finishes boarding.
    - However, it turns out that our use of a condition variable and lock above solves this, since only one thread can check its conditions and board at a time.

- **Sending the boat between Oahu and Molokai**

- The boat must only be sent once per trip. Since we need to indicate which child is the pilot to the grader if there are multiple children anyways, we can make it so that the pilot is the one who sends the boat between islands.
- “Sending the boat” means changing the information to indicate that the passengers have moved. This means updating their internal locations, changing the number of adults and children on both islands, and changing the location of the boat. The scheduler could interrupt between any of these, and give unexpected behavior.
    - To fix this, we can put a lock on all of these variables:

```
void sendToOahu(){
        varLock.acquire();
        // Change variables here (not shown)
        varLock.release();
}
```

    - This can be the same lock as used with the condition variable. In fact, we can simply not release the lock before calling the sendTo() function, and that should be enough.
- The thread that calls sendToOahu()/Molokai() would of course be forced to wait until the method finishes running to do anything else. However a child not piloting the boat may cause unintended problems if they try to do something while on the boat. However, the lock being acquired by the thread running the sending code should fix this.

- **The pilot must call ..RowTo.. before the rider calls ..RideTo.. when two children are riding**
    - We can use a condition variable to ensure that the rider goes last. This also takes care of waiting for a second rider when we have children.
    - However, there is a second thing to consider: the pilot releases the lock and gives it to the rider. It isn't guaranteed to reacquire the lock, so we need to finish the code in the rider's thread, not the pilot's. Therefore, when we call the transport function when we have two children, we transport them in the rider's thread. However, sometimes we only have a pilot, and thus must finish in the pilot's thread when we only have one person:

```
lock = new Lock()
cv = new Condition()

boardChild(bool waitForAnotherRider):
        lock.acquire()

        // Waits for a second rider if necessary
        // The first child to board would be the rider, not the pilot
        if(boat.empty() && waitForAnotherRider):
```

```
        cv.sleep()
        ChildRideToMolokai()/Oahu()
        transportBoat()
else :
        ChildRowToMolokai()/Oahu()
        cv.wake()
        If(!waitForAnotherRider): transportBoat()
lock.release()
```

With all these constraints in mind, here we have finalized pseudocode for our itineraries:

```
bool done = false
boatLocation = “Oahu”
lock = new Lock()
oahuCv = new Condition_var(lock)    // Wakes those on Oahu when the boat arrives there
molokaiCv  = new Condition_var(lock) // Wakes those on Molokai when the boat arrives there
boatCv = new Condition_var(lock)     // Wakes children on a boat when ready to ride

void AdultItinerary():
        location = “Oahu” // Initialize the location
        while(true):
                if(location == “Molokai”): finish() // Adults on Molokai have nothing to do.

                lock.acquire()
                // Sleeps if the boat isn’t in oahu, the boat can’t fit an adult, or if there are 2
                // children in oahu and an adult shouldn’t board.
                if(boatLocation = “Molokai” || !boat.empty() || childrenInOahu > 1):
                        oahuCv.sleep()
                else:
                        boardAdult(location)

                lock.release()

void boardAdult(location):
        // Check the location to figure out which one to use (not shown)
        AdultRideToMolokai/Oahu()
        transportBoat(adults = 1, children = 0, location)
        // once an adult arrives in Molokai, they have nothing to do. If they came from Oahu,
they are now in Molokai.
        if(location = “Oahu”): finish()

void childItinerary():
        location = “Oahu”
        while(true):
                if(done) finish()
                lock.acquire()

                if(location == “Molokai”):
                        if(boatLocation = “Oahu” || boat.hasOneRider()):
                                molokaiCv.sleep()
                        else:
                                // Only one child may board from Molokai
                                boardChild(waitForAnotherRider = false)
```

```
            else:
                if(boatLocation = “Molokai” || boat.isFull()):
                    oahuCv.sleep()
                else:
                    // Two children have to board from Oahu, unless there is only one
                    // child left, in which case this child doesn’t wait for another.

                    boardChild(waitForAnotherRider = (childrenInOahu > 1), location)
            lock.release()

void boardChild(bool waitForAnotherRider, location):

    // Waits for a second rider if necessary
    // The first child to board would be the rider, not the pilot
    // This cv is only for these two children. We don’t want to wake other islanders who are
    // only waiting for the boat to return and for them to board.
    if(boat.empty() && waitForAnotherRider):
        boatCv.sleep();
        // We can check which one to use based on the current location of the children
        // (not shown)
        ChildRideToMolokai()/Oahu()
        // We have to call this here, otherwise we cannot guarantee transportBoat will be
        // called before the other child reacquires the lock.
        transportBoat(adults = 0, children = 2, location)
        location = Molokai/Oahu
    else :
        ChildRowToMolokai()/Oahu()
        boatCv.wake()
        If waitForAnotherRider:
            // If we are taking two riders, we simply let the rider call transportBoat and
            // relinquish control.
            location = Molokai/Oahu
            return

        // This encapsulates that one child traveling to Molokai indicates that we are
done.
        if(location == “Oahu”) done = true;
        transportBoat(adults = 0, children = 1, location)
        location = Molokai/Oahu

void transportBoat(int adults, int children, location):
    newLocation = the opposite of location
    numChildrenIn(newLocation) += children
    numChildrenIn(location) -= children
```

```
numAdultsIn(newLocation) += adults
numAdultsIn(location) -= adults

boatLocation = newLocation
if(newLocation = oahu): oahuCv.wakeAll()
else:                   molokaiCv.wakeAll()
```