Design Document

Project #1
Due Date: 2/3/2012

Group 2
Daeik Kim
Jonathan Child
Fred Knutson
Kevin Caton-Largent

Solution Design

Part 1: Implementation of Join method from KThread class

The `join()` method will check the current thread to see if it is finished. If it is finished, it will continue running normally and skip everything else as it will be able to continue as the parent thread is not waiting on anything. If it is still running, it will interrupt the current thread (parent), and put the parent on the queue.

When originally writing this, I did not see some other things, or made some assumptions about the code. One of the assumptions was that the computer would know to call the thread. In addition, I assumed there was not a separate queue, but rather the general wait queue that would allow it to be checked all the time, each time it went around and got to the top. This idea was, as I found out, very wrong. To fix this, I created a separate list just for the join. In addition, in the `finish()` method, I added in a conditional statement to see if there was anything on the list. If there was, it would wake that instead of using the normal queue.

This is possible because at the end of joining two threads, there will always be something that is waiting, so when `finish()` is called, it means that what the `joinList` is waiting on is ready to be accessed because of the non-global variables.

As I am writing this, after turning it in, I realized that my original naming convention of calling it `threadList` was not as descriptive as `joinList` and have decided to rename it as that. All future updates will include the new naming convention. Reading through this again, I have decided that in the future parts if there are any problems that relate to join, it is probably in how I interrupted the threads and should possibly use more broad implementation of the interrupts. As it is, the code works properly, so this is a note for the future.

```
create a new linked list call joinList
public void join() {
        // included in code
        Lib.debug(dbgThread, "Joining to thread: " + toString());
        // included in code
        Lib.assertTrue(this != currentThread);

        if child is not finished        // if the child is finished, it will skip these lines of
        {
                add current thread to list
                disable interrupts
                put the current thread to sleep
                restore interrupts
        }
}

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

Machine.interrupt().disable();
```

```
        Machine.autoGrader().finishingCurrentThread();

        Lib.assertTrue(toBeDestroyed == null);
        toBeDestroyed = currentThread;

        // Start of code I added
        if (joinList size > 0) {
                disable interrupts
                get first item from joinList
                restore interrupts

                remove first item from joinList
        }
        // End of code I added

        currentThread.status = statusFinished;

        sleep();
  }
```

For testing this thread, we will implement multiple threads and have one parent call to join their child, one time being forced to wait, one time having the child running, and if possible, we will try to see what would happen if the child has not run at all yet, as if they had never been created.

Part 2: Implementation of Condition2 class

Condition2 is an implementation of condition variables that disables interrupt()s for synchronization. In order to keep track of waiting threads, Condition2 will have linked lists of KThreads to queue in threads.

For sleep(), we will need to add the current thread to the linked list so it will wait until it's awoken by wake() function. But before we do this, we have to disable interrupt so that the working threads can finish their tasks. This has to be restored after acquiring the lock at the end. Then we will need to release the lock, and restore the interrupt status and reacquire the thread from lock.

```
public void sleep() {
        //Disable interrupt from machine
        add the current thread to the linked list
        /*release the lock and put it to sleep*/
        /*Acquire the lock again*/
        //Restore machine interrupt using boolean variable
}
```

For wake(), similarly, we will disable interrupt before venturing onwards with any tasks. Then it will check to see if the size of the queue is greater than zero, so that it knows that there is a list to work on. If it is, then turn the status of first queue in the list to ready. Then it will restore the machine status at the end.

```
public void wake() {
        //Disable interrupt from machine
        if (not zero)
        {
```

```
                /*change the status of the threads*/
        }
        //Restore machine interrupt
}
```

For wakeAll(), Similar to wake(), instead of checking the size once, use the while loop and take the threads in linked lists one by one to change the status to ready until the list is empty by calling the wake function. After it is finished changing the status, then it will restore the interrupt status.

```
public void wakeAll() {
        //Disable interrupt from machine
        while (not zero)
        {
                //call wake function
        }
        //Restore machine interrupt
}
```

To test this class, we would have to create a condition2 variable with various amounts of threads, similar to autograder. I would first declare the variable Condition2 and try to run various tasks to give few threads at one time and for another instance, I will add more tasks so that it will contain multiple threads. Then I will check to see how each threads are synchronized, and if my expected output equals the actual output, then my codes are proven for proper synchronization of multiple threads.

Part 3: Implementation of WaitUntil and TimerInterrupt from Alarm class.

The waitUntil() method puts the current thread to sleep for at least x ticks and wakes it up in the timer interrupt handler. Since waitUntil is not limited to one thread, a priority queue can be made to keep track of the threads that are still waiting to be awakened. The priority queue would implement a separate comparator class to keep the threads ordered in the proper ordering. A separate class would also be needed to store the current thread and its number of ticks required for waiting.

```
PriorityQueue of SleepingThreads
        /* The comparator would order the threads according to their wait times */

class SleepingThread {
        /* Stores current thread and the amount of time for waiting */
}

class ThreadComparator {
        /* would compare time required for waiting amongst two threads */
        /* The order is based off of the time the threads require for waiting */
        /* Threads that have shorter wait times would be placed in front of the queue whereas threads
        /*      with longer wait times would be placed in back of the queue. */
}
```

In the waitUntil method a new sleeping thread will be created that stores the current thread and the amount time for it to wait. It then stores the sleeping thread onto a priority queue that will then keep track of threads waiting to be awakened.

```
waitUntil(long x)
```

```
{

        // variable to store the amount of time to wait until waking thread.
        // it stores the current time + the time (x) to add for waiting
        wakeTime = current time + x
        // ensures that the time for waking has not passed and that there is a valid value in wakeTime
        if (wakeTime > currentTime)
        {
                // disable interrupts
                status = Machine.interrupt().disable()

                /* Create new sleeping thread that contains the current thread and the amount of time
        required for waiting */
                SleepingThread = new
                SleepingThread (current thread, current time + x)
                /* place sleeping thread on the priority queue */
                PriorityQueue insert SleepingThread

                /* put current thread to sleep */
                currentThread sleep
                Machine.interrupt().restore(status)
        }

}
```

The timer interrupt handler handles the threads that wake and places them onto the ready queue. It first causes the current thread to yield, which forces a context switch if necessary. It then loops through the priority queue taking the most ready thread and adding them to the ready queue. The head of the queue would be the most ready thread from the priority queue based upon the comparator's ordering. The loop would check to see if the head is null and if it is less than the current time. This is to guard against putting a thread before the time has passed even though it is the next to go on the priority queue.

```
timerInterrupt ()
{
        // initial code
        Kthread.currentThread().yield()
        /**/

        /* store head of priority queue for later use */
        head = PriorityQueue head

        /* checks to see if head is not null, and if time has passed for waiting
        * If the time has passed it loops through the priority queue until it reaches a thread that
        still has time leftover to wait.
        */

        while (head != null && head <= current time)
        {

                // disable interrupts
```

```
                status = Machine.interrupt().disable()
                /* place head of priority queue onto ready queue */
                readyQueue insert head
                /* remove the ready thread from the priority queue because it is no longer waiting */
                PriorityQueue remove head
                /* update the head variable to point to the new head */
                head = PriorityQueue head
                // restore interrupts
                Machine.interrupt().restore(status)

        }

}
```

To test this class we created a number of threads in a test method. Each thread would have their own different wait times that they would pass in to the waitUntil method. We would then check to see how much time passed for them to wake and see if the times were consistent with the times passed in + the current time at the time they were passed in.

Part 4: Implementation of Communicator class.

The first thing we need to do is create the constructor class for Communicator. In order to accomplish this task efficiently we have decided it is best we use Condition2 class for our wake /sleep functions. We will create two boolean objects to keep track of our listener and speaker. Finally we will create a Integer type variable to store the word spoken by the speaker.

The main objective for this class is to implement synchronous send and receive a one word message between the speaker and the listener objects. The speaker will wait until the listener is called on the same communicator object. So to do that we created a speaker method that will check if the listener method has been called. If the listener method had not been called yet, the speaker class will put the condition object to sleep. If it does have a listener, it will change the status of the threads in the condition variable to ready. And at the end it releases the thread.

```
public void speak(int word) {
        acquire thread;
        while(speaker speaking){
        sleep;}
        speaker speaking = true;
        word = word;
        word_is_ready=true;
        there_is_a_word.wake;
        while(word ready){
        word gone sleep:}
        speaker=false;
        wake speaker;
        release thread;


}
```

The next part to this class is the listener method. It retrieves the word that was saved in the speaker class.

Listener will wait on the speaker until it has been called. So to accomplish this we will create blocks of conditional statements, to check if the speaker has been called. If the speaker has not been called we set the boolean value related to the listener calls to true. We then put the condition to sleep. Then pass the word spoken by the speaker to the listener. You then you reinitialized the boolean variables, the listener and the speaker, back to false, then release the lock and return the word. But if speaker has been called, you first wake up all the threads in the condition variable. Then you pass the word from the word spoken by the speaker to the listener. Then you reinitialized the boolean variables, speaker and listener, to false. Then you release the lock and return the word.

```
public int listen() {
        acquire thread;

        while(listener busy){
        sleep;}
        listener busy = true;
        while(word not ready){
        there is a word .sleep;}
        word = word;
        word gone.wake;
        listener busy = false;
        listener ready wake;
        release thread;
        return word;

}
```

How we plan on testing communicator is we will pass in several different threads. First one that will pass in a listener before speaker, the two speakers and two listeners and finally two listeners then two speakers. This will test several different things. The first one if the code can handle a listener before a speaker, seeing if it will try to return something before the speaker can pass anything into word. The second thing will test if the code can handle multiple speakers, if the second speaker replaces the word the first speaker passed in before the listener has a chance to return the word. And the last case will see if the listeners will return both words the speaker will pass in or if it returns one because the second listener replaces the first listener.

Part 5: Implementation of Priority Scheduler class

For priority scheduling, each thread needs a name, a start time, threads needed (or threads that are being waited for), and a priority. Once a thread has been given a name and a priority, it can be placed on the queue based on its priority, an int that ranges from 0-7, with the default being 1. If there is nothing in the queue and a free thread, it will automatically run. Each time a thread opens up, the thread with the highest priority will be chosen. If there are multiple threads with the same priority, that priority being the highest on the queue, the one that has been waiting the longest will be chosen, as taken by the current time minus the start time. The ThreadState class will manage the basic needs of each thread such as storing its state, donations, and priorities and as well as updating those upon necessity.

Note: After finishing `priorityScheduler`, I can look back on this and see only one small difference in way of thinking: to make it easier to check the time, we check to see whether or not one `waitingTime` for a thread is less than the `waitingTime` for another thread. The lower the `waitingTime`, the longer it has been waiting.

As far as the second part of `priorityScheduler`, implementing it was a lot harder than we had assumed it originally to be. As can be seen by our psuedo-code and writeup, our ideas were only half right. Because of that, we soon encountered trouble implementing this class. In the end, we had to deepen our understanding of the

class itself to determine what was the correct thing to do.

First of all, for pickNextThread(), we needed to change how it worked. Our original idea was that it would be sorted previously, not understanding that we had to do that ourselves. Because of this, I can not speak for how efficient the code is, but it works.

```
PriorityQueue class

/* contains methods for updating priorities of the threads in the queue */
/* keeps tracks of the threads priorities */

        protected ThreadState pickNextThread() {// Looks to see what the next thread is, a peek
        function
        Checks to see if there is anything on the thread list, if not, then exit method

        Checks the priorities of all waiting threads. If equivalent priorities, takes longest
        time. Will do this by implementing a for-loop that goes through the length of the
        threadList.

        Check the transfer priority, aka check to see if there should be a transfer from waiting
        threads. If there is a need, transfer the effective priority. Otherwise, just pass in
        the priority.

        Check the priority of the current thread to the max priority on the list, if it is
        greater than, set the new max to the current priority, change the position of the max,
        and update the time. If they have the same priority, check to see what was there the
        longest.
         Will return the thread that has the highest priority, but not remove it. This is just
        to see what is next.

        nextThread()
        Selects the next thread in the queue. Will first check to see if the owner has a value,
        if it does, it will be removed.

        Next, it will check the list of threads on the queue. If it is empty, it will reset
        everything and exit the nextThread() method.

        Next, the owner will obtain the thread that is chosen by pickNextThread. As a check, it
        will see if it is null, which could reflect an idle thread that was created, but overall
        should be useless. All of the information, priorities, and permissions will be given to
        owner and it will be removed from the threadList.

        The owner will then be returned.



getEffectivePriority()
Returns the priority of the thread after taking into account the donations.

Check to see if the effectivePriority is the recalc priority, if it is, then it has been updated
properly. Otherwise, skip this section.
```

// If not the same, start skip here //
starts a loop from 0 to the size of the list
the waitQueue queue will obtain the value in the list each time it increments to the next value. Will also obtain threads from the queues.

Start next for loop from 0 to the size of the threads. Current priority will get the priority of the j value and check to see if it is the max value, if it is greater than the max, it will set the effective priority to that highest max, incrememting through the entire series of queues and lists. The one possible way to check for this would be to add an if statement checking to see if effectivePriority was 7 it would automatically exit the loop.

// End of loops //

Checks to see if the effective priority has been changed, if it has not, the effective priority will just be the priority of the current thread, and the effective priority will be returned.

setPriority(int priority)
Will set the priority of the associated thread to the specified value .

```
// included in base code
if (this.priority == priority)
        return

this.priority = priority;
// end of base code
```

Inside of the base code, have a check to make sure it is not more than the max and not less than the min (7, 1, respectively). Also, update effective priority.

waitForAccess (KThread thread)
Passes in the waitQueue, which does nothing for our code as we just save the current time into waitingTime.

acquire(KThread thread)
Add thread to the acquiredList. Reset effectivePriority.

increasePriority()
Will increase the priority of the thread by one and check if its priority is not already maxed before increasing.

```
if priority = max
        exit
thread priority + 1
```

decreasePriority()
Will decrease the priority of the thread by one and check if its priority is not already at minimum priority before decreasing.

```
if priority = min
```

```
        exit
thread priority - 1
```

Out of what we originally thought we had to do, this was it, but due to unforseen problems and lack of understanding, we had to add in many different methods to make it work properly. Most were not too difficult, the hardest method overall being `nextThread()`, but the realization that we needed our own queues that were not originally provided helped us to finish the problem.

As for testing this, we will pass in multiple threads with various priorities ranging from 0-7 and check to see how the code interacts with them. For instance, one case would be to pass in 10 threads, 7 of which are dependent on the lowest priority, the rest randing upwards to the max of 7, and the other three threads containing lower numbers, for instance 1, 3, 5 and seeing what would happen. In addition, to test the priorities when two threads had been created separately with the same priority, we would give two threads the priority of 7 and have one made before and one made after and make sure they run in the proper order. Besides that, any and all testing that would happen would be variations of this.

Part 6: Solution to boat Problem

For the boat problem, where you can only carry two children or one adult at a time, the solution will be to carry the adult only if there is at least one child at Molokai to pilot the boat back to Oahu. To make the boat.java problem more entertaining, we assumed that the boat was actually a ***nuclear submarine***. We also assumed that the children are super kids that can operate a submarine by themselves. For the given problem, we decided that it will be most efficient to keep track of number of children in Molokai using lock, condition, and semaphores. We would also need to utilize boolean type variables to help us identify various states such as status of the nuclear submarine, whether or not it is being occupied by someone, and whether or not the entire process is complete. We will also use another condition type variable to allow for synchronization while we try to ferry two children on the nuclear submarine. Furthermore, with a for loop, we will change the number of adults and child passed in as argument to number of threads using Kthread; this is given to us in example for sample itinerary with runnable interface. For each thread, adult or child, we will set the proper name and fork the thread to get the process going.

```
//Pseudocode//

public static void begin(int adults,int children,nuclear submarineGrader b)
{
        //Create the required number of Adult threads for (number of adults passed in)
        {
                Runnable r = new Runnable()
                {
                        public void run()
                        {
                                AdultItinerary();
                        }
                };
                Set as Kthread.
        }

        //Create the required number of Child threads for (number of children passed in)
        {
                Runnable r = new Runnable()
                {
```

```
                        public void run()
                        {
                                ChildItinerary();
                        }
};
Set as Kthread.
}

while (process is not complete)
{
/*Check to see if crossed Adults equal to the total number of adults and crossed */
/* children equals to the total children minus one. */
Yield main thread to ensure other threads finish.
}
transport last child over
}
```

Both itineraries for adult and child will call upon the various nuclear submarine grader functions to make it more visibly understandable by the reader as to what each lines of codes will be doing and how it is done. AdultItinerary() will first wait until there is at least one child at Molokai with semaphores. Then to ensure the submarine doesn't leave without anyone on board, we will make sure the adult is in the submarine with Lock before we leave to Molokai. Then once he transports, we will wake up a child from Molokai to ferry the nuclear submarine back to Oahu. Then we will increment the crossed Adult value by one and decrement crossed Child by one. Then we will release the lock to end our Itinerary for the Adult.

```
static void AdultItinerary()
{
//Wait until there is at least one child at Molokai
//get in the submarine
//transport to Molokai
Wake up the child to send the nuclear submarine back to Oahu
Increment counters for Adult to signify that the adult is at Molokai
Decrement counters for Child to signify that the child, who will be pilot the nuclear submarine back
toOahu, is not at Molokai.
}
```

On the other hand, ChildItinerary would need have more complex system compared to Adult planned route because they will need to do more work compared to Adult under given circumstances. However since complexity is woven out of simplicity, we will break it down step by step to make our approach more understandable.

So the first thing we need to make sure before we do anything is that there are still people are on Oahu. Once you check that we have people Oahu then we can set the boolean type variable , that the child starts on Molokai, to false because the child has to start at Oahu. The next the task is to set the condition if they are at Molokai. If they are at Molokai then we need the child to pilot the nuclear submarine back from Molokai to Oahu. So we need the child to get in the submarine in and then travel back to Oahu, then get out of the submarine and release the boat to allow the adult to get in the submarine. But if there is no child at Molokai we need to transport at least one child to the island. So what we will do to accomplish this task is get we have to get two children in the submarine going from Oahu to Molokai. So we have to reserve spots for the two children on the sub. Then we need to get the children in the sub one at a time. We have to create a checking system most likely using if and else statements. So the if statement will check if there is at least one child on the sub, and the

else statements will change the conditional statement saying there is one on the sub to true, and putting a child on the sub and putting it to sleep. The if statement will then put the other child on the sub as the pilot and wake the first child up and transport them to Molokai. Once there the passenger child will get off then the pilot child will pilot the sub back to Oahu to give the boat to the next adult to transport him/her to Molokai.

```
static void ChildItinerary()
{
        start with no children at
        Molokai, children at Molokai = false;
        while (!complete)
        {
                if (at Molokai)
                {
                        child travels to Oahu;
                }
                else
                {
                        reserve seat for two children;
                        if (child on submarine)
                        {
                                child on submarine = false;
                                child get on sub;
                                wake first child up;
                                sub travels to Molokai;
                        }
                        else
                        {
                                child on submarine = true;
                                child get on sub, sleep and
                                wait for second rider;
                                child rides to Molokai;
                                child gets off at Molokai
                        }
                        children off the sub;
                }
        }
}
```

For self test cases, we will pass in various combinations of adults and children on the island. The main thing we are looking for in the self test cases are whether or not the number of crossed adults and children on the destination island are equal to the number of the initial number of people we passed into the argument, and also whether or not we deleted or added any travelers in the process of crossing them over.