Design Document: Threads

Problem Description:

Currently KThread.yield() when called is repeatable, which means it should do the same thing everytime, however the command nachos -s <#> calls KThread.yield() and inserts it at different places in the code.

Tasks:

1. Implement KThread.join(), which synchronizes the calling thread with the completion of the called thread. (SEE a.) We say that thread B joins with thread A. When B calls join on A, there are two possibilities. If A has already finished, then B returns immediately from join without waiting. If A has not finished, then B waits inside to join until A finishes; when A finishes, it resumes B. Often thread B is called the "parent" and A is called "child" since a common pattern is for a thread that creates child threads to join on them to wait for them to finish. However, note that any thread can call KThread.join() on another (it does not have to be a parent/child relationship).
    a. As an example, if thread B executes the following:
    KThread A = new KThread(...);
    …
    A.join();

2. We need to implement condition variables using interrupt enable and disable to provide atomicity. In condition1.java, we are using the implementation by semaphore, but we also need to provide an equivalent implementation in class Condition2 by manipulating interrupts instead of using semaphores, and reimplement condition.java in condition2.java, such as sleep(), wake(), and wakeall(). So once all is set, there will be two alternative implementations that provide the exact same functionality, one with semaphore and one without it. Finally, we are required to make sure that the thread must have acquired the lock associated with the conditions variable when it invokes methods on the CV.
3. Complete the implementation of the Alarm class by implementing the waitUntil(long x) method. A thread calls waitUntil(long x) to suspend its execution until wall-clock time has advanced to at least now + x. This method is useful for threads that operate in real time, such as blinking the cursor once 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 fork any additional threads to implement waitUntil(); you need only modify waitUntil() and the timer interrupt handler methods. waitUntil() itself, though, 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).

Note that only one instance of Alarm may exist at a time (due to a limitation of Nachos), and Nachos already creates one global alarm that is referenced via ThreadedKernel.alarm.

4. Hhhh
5. Now with the other tasks complete we have a number of Hawaiian adults and children that are trying to get from Oahu to Molokai. There is only one boat and it can only carry two children or one adult. The boat can go back and forth but must have a pilot to do so. Get a solution to get everyone across.

Solutions to the problems:

KThread.join()

The thread join() function is called from a currently executing Thread and ensures the calling thread completes it's entire instructions before continuing to execute the Joined thread. For example, if we create Threads A and B. Suppose we begin executing thread A and within thread A, we call ThreadB.join(), Thread B will begin it's execution and when completed Thread A will join back where it left off. Through this, we can say join() provides a way to recombine two concurrent computations into a single one. Join() is considered also to be a blocking method, which blocks until the thread on which Join has called a die or specified waiting time is over.

- Correctness of Join() function:
    - Suppose you have ThreadA joining ThreadB:
        - If ThreadB has nothing to compute, continue running Thread A
        - Thread A cannot Join with itself
        - ThreadB can't Join if it hasn't been started yet
        - Should only call join once, if more than once not guaranteed to return

In order to solve this problem, we must first check if the Thread we are trying to join is not the current thread

- if(this != currentThread)

Then, knowing this is not the same thread trying to join itself, we can check if the current thread's status is already Finished. If it is, then we can return from the join function immediately

- Status == statusFinished

If both these checks succeed, we should be able to put the current thread in the Ready Queue after putting it to sleep. When the thread wakes it will ensure that the thread we wanted to join is completed so we are ready to start the old thread again

```
waitQueue.waitForAccess(currentThread) // add to readyQueue
currentThread.sleep()
```

This current Thread will simply be added into a LinkedList<Thread>
We also must make sure that interrupts will not happen otherwise the currently executing thread might be unfinished when it interrupts to the parent thread.
Once it is finished we can simply signal all waiting threads in the waiting queue for the child thread

PseudoCode:

```
Public void Join() {
        Disable interrupts
        if(this != currentThread and status != StatusFinished and currentThread.joined){
                Add parent thread to waiting queue
                Put parent thread to sleep
                Enable interrupt;
                return ;

        } else {
        Enable interrupts;
        Return;
        }

}

Void finish() {
// when the child thread has finished execution
        Add thread to dead queue
        Wake up all waiting threads (parents)
}
```

Condition Variables

In order to solve this problem, we must implement the condition variables using monitors in our program. We use variables such as sleep() and wake(). Sleep() is to let the thread wait on a condition and go to sleep. Wake() will check the waiting list; if there is any one waiting, then wake up one waiter. Threads will call sleep(), and will be queued up waiting for a call to wake(). It will pick the first thread that is waiting.

To support sleep() function while waiting inside the critical section, we can add wait(&lock), so it will atomically release the lock and go to sleep. Later it will reacquire the lock later before returning. At last, we will use wakeall() to wake up all of the waiting threads.
Here is how a pseudocode example of monitor implementation will be:

```
Public class Condition2{
Setup Lock, Condition, and Queue
sleep()
{
        Acquire lock
        While(the queue is empty)
                wait(&lock) //puts thread on queue and releases lock
        Get the next item
        Release the lock and return the item
}

wake()
{
        Acquire lock
        Add the item in the queue
        Signal the first waiter
        Release the lock
}

wakeall()
{
        Acquire lock
        Add the item in the queue
        Signal all the waiters
        Release the lock
}
}
```

Alarm

To solve the problem, we need two functions WaitUntil(long x) and TimerInterrupt() in the file named Alarm.

Function WaitUntil(long x) takes the desired wait time from the thread, and adds up the current system time and desired wait time and returns an absolute system time that the thread wants to resume.

Function TimerInterrupt() is to put the threads that are ready to run on the ready queue. We can sort the absolute system time of each thread so that the function TimerInterrupt() is able to call them one by one.

Example:

ThreadA called function WaitUntil at system time equal to 2 and wait time equal to 5
ThreadB called function WaitUntil at system time equal to 3 and wait time equal to 1
Function WaitUntil will calculate the absolute system time for Both thread, A:7, B:4
Function TimerInterrupt first sorts the absolute system time, 4(B),7(A). Then wakes them up when the time comes.

Communicator

To solve this problem we must start by creating a communicator class and a communicator within. Then we must create a method that takes an integer named word as a parameter and a method listen that takes no parameters.
The communicator lets threads exchange one word messages. It can have multiple threads waiting to listen and multiple threads waiting to speak.
We must make sure we do a mutex lock so only one thread can be executed at one time.

Public class communicator(){

public communicator() contains mutex lock

- Create Sender and Listener
- Create queue for listen threads
- Create queue for speak threads
- The communicator pairs a speak thread to a listening thread so every listener can only receive one word.

public listen()

- First it must acquire mutex to run the process and access the communicator
- Since it is using mutex, the speak process must be asleep
- Returns word for speak thread
- Releases the mutex for other process and listen sleeps

The speak method takes the word parameter from listen
public speak(int word){

- First it must acquire mutex to run the process and access the communicator
- We then add the speaker to the waiting queue
- Since it is using mutex, the listen process must be asleep
    - If there is no waiting listener, wake up Listener from constructor
    - Place speaker in sending queue
- Executes process and speak the word
    - If the speaker or listeners are still waiting
        - Wake up speaker and listener

- Awake next speaker and listener from waiting queue
- Releases the mutex for other process and speak sleeps

}

Test:

- To test the effectiveness we need to test with different words
- We need to test the time, so we need to see how fast the threads are handled. We can do this by testing with different time intervals for each thread

Boat Problem

1. In order to solve this problem/task we must be able to identify the objective which is Getting everybody on Oahu to Molokai. With this we can initialize the number of children and adults. Since only 1 adult or 2 children can be on the boat at once, the amount of children can't be less than 2 or else we won't be able to get everyone across. Must have a variable to determine if the boat is on Oahu or Molokai.
   Boat.Begin(){
   TotalAdults = (Total Number Of Adults)
   TotalChildren = (Total Number Of Children)
   BoatOnOahu = True
   …
   Since the total number of children and adults will all start on Oahu, the total number of children and adults initially is the same as the total number of children and adults, also the number of children and adults on Molokai are initially zero. We must also keep in mind that there are situations where this problem is absolutely not solvable and that's when the initial number of adults is greater than 0 and the number of children is less than two.
   ...
   AdultsOnOahu = TotalAdults
   ChildrenOnOahu = TotalChildren
   AdultsOnMolokai = 0
   ChildrenOnMolokai = 0
   …
   Now we must create an algorithm that handles certain situations depending on the amount of adults and children since adults can only travel alone. There must be no children on Oahu or at least one child on Oahu and one on Molokai. To handle this, a

while loop can be useful. Also the function ChildRowToMolokai() means that a child is the pilot of the boat and heading to Molokai/Oahu depending on the destination. ChildRideToMolokai() means the child is the passenger of the boat along with a child pilot. AdultRideToMolokai() is just the adult going to molokai.

```
…
   While(AdultsOnOahu > 0){
      ChildRowToMolokai()
      ChildRideToMolokai()
      ChildRowToOahu()
      AdultRowToMolokai()
   }
…
```

Now that the adults are handled, it will depend on the amount of children that are left, so after the while loop is done and there are still children then there should be another while to finish getting the chosen across. Also there may be cases were the boat is on molokai and not on Oahu

```
…
   While(ChildrenOnOahu > 2){
      If(BoatOnOahu=False){
         ChildRowToOahu()
      }
      ChildRowToMolokai()
      ChildRideToMolokai()
      ChildRowToOahu()
      ChildRowToMolokai()
      ChildRideToMolokai()
   }
…
```

From this, there is still a chance for one child to be left on Oahu so must address that situation.

```
…
   While(ChildrenOnOahu = 1){
      ChildRowToOahu()
      ChildRowToMolokai()
      ChildRideToMolokai()
   }
}
```

This should handle getting everyone across with these both while loops, but now we must fix both of these loops with changing the number of adults/children are on oahu to know if the boat should go back from molokai to oahu to get everyone across. Everytime

ChildRideToMolokai() is called the number of children on Oahu is minus 2, the number of children on Molokai is plus 2 and BoatOnOahu is False.

```
ChildrenOnOahu = ChildrenOnOahu - 2
ChildrenOnMolokai = ChildrenOnMolokai + 2
BoatOnOahu = False
```

Everytime AdultRowToMolokai() is called the number of adults on Oahu is minus 1 and the number of adults on Molokai is plus 1.

```
AdultsOnOahu = AdultsOnOahu - 1
AdultsOnMolokai = AdultsOnMolokai + 1
BoatOnOahu = False
```

We also need to keep track for when the boat goes from Molokai to Oahu. Everytime ChildRowToOahu() is called, only one child needs to go back, so the number of children on Molokai is minus 1 and the number of children on Oahu is plus 1.

```
ChildrenOnMolokai = ChildrenOnMolokai - 1
ChildrenOnOahu = ChildrenOnOahu + 1
BoatOnOahu = True
```

There will be no need for an adult to row to Oahu so there is no reason to address that situation with the counters.

We must also address that there will be times where while loops will begin but the boat will be on Molokai and not Oahu, so we must make an if statement and insert it at the beginning of the while loop to get the boat back to Oahu.

```
…
        If (BoatOnOahu = False){
                        ChildRowToOahu()
                                ChildrenOnMolokai = ChildrenOnMolokai - 1
                                ChildrenOnOahu = ChildrenOnOahu + 1
                                BoatOnOahu = True
                }
...
```

Updating the pseudocode with all variables counters, situations, while loops, and if statements in full effect, the complete pseudocode should look like the following:

```
Boat.Begin(){
TotalAdults = (Total Number Of Adults)
TotalChildren = (Total Number Of Children)
AdultsOnOahu = TotalAdults
ChildrenOnOahu = TotalChildren
AdultsOnMolokai = 0
ChildrenOnMolokai = 0
        While(AdultsOnOahu > 0){
                If (BoatOnOahu = False){
```

```
                ChildRowToOahu()
                        ChildrenOnMolokai = ChildrenOnMolokai - 1
                        ChildrenOnOahu = ChildrenOnOahu + 1
                        BoatOnOahu = True
        }
        ChildRowToMolokai()
        ChildRideToMolokai()
                ChildrenOnOahu = ChildrenOnOahu - 2
                ChildrenOnMolokai = ChildrenOnMolokai + 2
                BoatOnOahu = False
        ChildRowToOahu()
                ChildrenOnMolokai = ChildrenOnMolokai - 1
                ChildrenOnOahu = ChildrenOnOahu + 1
                BoatOnOahu = True
        AdultRowToMolokai()
                AdultsOnOahu = AdultsOnOahu - 1
                AdultsOnMolokai = AdultsOnMolokai + 1
                BoatOnOahu = False
}
While(ChildrenOnOahu >= 2){
        If (BoatOnOahu = False){
                ChildRowToOahu()
                        ChildrenOnMolokai = ChildrenOnMolokai - 1
                        ChildrenOnOahu = ChildrenOnOahu + 1
                        BoatOnOahu = True
        }
        ChildRowToMolokai()
        ChildRideToMolokai()
                ChildrenOnOahu = ChildrenOnOahu - 2
                ChildrenOnMolokai = ChildrenOnMolokai + 2
                BoatOnOahu = False
        ChildRowToOahu()
                ChildrenOnMolokai = ChildrenOnMolokai - 1
                ChildrenOnOahu = ChildrenOnOahu + 1
                BoatOnOahu = True
        ChildRowToMolokai()
        ChildRideToMolokai()
                ChildrenOnOahu = ChildrenOnOahu - 2
                ChildrenOnMolokai = ChildrenOnMolokai + 2
                BoatOnOahu = False
```

```
        }
        While(ChildrenOnOahu = 1){
                If (BoatOnOahu = False){
                        ChildRowToOahu()
                                ChildrenOnMolokai = ChildrenOnMolokai - 1
                                ChildrenOnOahu = ChildrenOnOahu + 1
                                BoatOnOahu = True
                }
                ChildRowToOahu()
                        ChildrenOnMolokai = ChildrenOnMolokai - 1
                        ChildrenOnOahu = ChildrenOnOahu + 1
                        BoatOnOahu = True
                ChildRowToMolokai()
                ChildRideToMolokai()
                        ChildrenOnOahu = ChildrenOnOahu - 2
                        ChildrenOnMolokai = ChildrenOnMolokai + 2
                        BoatOnOahu = True
        }
}
```

Now with the main pseudocode idea to complete Task 6 done, its should have some Test cases to prove the algorithm works. In order to make these test cases we need to initialize random numbers with each other so many ways we can do this is by having odd numbers for both adults and children, we will call that Test Case 1. Even numbers for both adults and children, we will call that Test Case 2. Odd number for adults and even number for children will be Test Case 3. Even numbers for adults and odd numbers for children will be Test Case 4.

Test Case 1: TotalChildren = 3
TotalAdults = 1
Then: ChildrenOnOahu = 3
AdultsOnOahu = 1
ChildrenOnMolokai = 0
AdultsOnMolokai = 0

Using our algorithm, since number of adults is greater than 0 the first while loop will run which will do the following: first 2 children will go across making ChildrenOnOahu = 1, ChildrenOnMolokai = 2 and BoatOnOahu = False, then 1 child will go back to Oahu making ChildrenOnOahu = 2, ChildrenOnMolokai = 1 and BoatOnOahu = True. Second the adult will go to Molokai making AdultsOnOahu = 0, AdultsOnMolokai = 1 and BoatOnOahu = False. Since there is no more adults and still two children on Oahu the 2nd while loop will execute, inside there is a if statement to determine the location of the boat since the boat is at molokai we must send 1 child back making it

ChildrenOnMolokai = 0 , ChildrenOnOahu = 3 and BoatOnOahu = True, then the rest of the while loop will execute, 2 children will go to Molokai making ChildrenOnOahu = 1, ChildrenOnMolokai = 2 and BoatOnOahu = False. Then again one child is sent back to Oahu making ChildrenOnOahu = 2, ChildrenOnMolokai = 1 and BoatOnOahu = True. Finally the last children are sent to Molokai making ChildrenOnOahu = 0, ChildrenOnMolokai = 3, and BoatOnOahu = False. At this point everyone should be across, to double check everything is correct, TotalChildren should be equal to ChildrenOnMolokai, as well as TotalAdults = AdultsOnMolokai. These Both checkout, so Test Case 1 is good.

Test Case 2: TotalChildren = 2
TotalAdults = 2
Then: ChildrenOnOahu = 2
AdultsOnOahu = 2
ChildrenOnMolokai = 0
AdultsOnMolokai = 0

The algorithm will execute the first while loop because AdultsOnOahu is greater than 0, this will send 2 children across resulting in ChildrenOnOahu=0, ChildrenOnMolokai=2 and BoatOnOahu = False. Then, 1 child will return to Oahu resulting in ChildrenOnOahu=1, ChildrenOnMolokai=1 and BoatOnOahu = True. Next, an adult will go across to Molokai AdultsOnOahu=1, AdultsOnMolokai=1 and BoatOnOahu = False. Now the while loop will run again but since BoatOnOahu=False the if statement will run which will send a child to go back to Oahu leaving ChildrenOnMolokai=0, ChildrenOnOahu=2 and BoatOnOahu=True. Then 2 children will ride to Molokai making ChildrenOnOahu = 0, ChildrenOnMolokai=2 and BoatOnOahu=False, then 1 child will go back to Oahu making ChildrenOnOahu=1, ChildrenOnMolokai=1 and BoatOnOahu=True. Next, the adult will go across updating the variables to AdultsOnOahu=0, AdultsOnMolokai=2. Now all adults are on Molokai, so we need to get the one remaining child across, which will run the final while loop and the if statement because the boat is on molokai. 1 child will go to Oahu making ChildrenOnOahu=2, ChildrenOnMolokai=0 and BoatOnOahu=True. Finally the 2 children will both go to Molokai making ChildrenOnOahu=0, ChildrenOnMolokai=2 and BoatOnOahu=False. TotalChildren and TotalAdults are equal to ChildrenOnMolokai and AdultsOnMolokai respectively. Therefore Test Case 2 passes.

Test Case 3: TotalChildren = 2
TotalAdults = 1
Then: ChildrenOnOahu = 2
AdultsOnOahu = 1
ChildrenOnMolokai = 0
AdultsOnMolokai = 0

This test case is simple enough for our algorithm. Firstly, the first while loop will run because AdultsOnOahu is greater than 0, so 2 children will cross to molokai so that ChildrenOnOahu=0, ChildrenOnMolokai=2 and BoatOnOahu=False. Then a pilot child will return to Oahu, ChildrenOnOahu=1, ChildrenOnMolokai=1 and BoatOnOahu=True. Then the adult will cross resulting in AdultsOnOahu=0, AdultsOnMolokai=1 and BoatOnOahu=False. The child on Molokai goes back to Oahu, ChildrenOnOahu=2, ChildrenOnMolokai=0 and BoatOnOahu=True. Then that child picks up the final child to have ChildrenOnOahu=0, ChildrenOnMolokai=2 and BoatOnOahu=False.The children and adults on Molokai are the same as TotalChildren and TotalAdults. Therefore Test Case 3 passes.

Test Case 4: TotalChildren = 3
TotalAdults = 2
Then: ChildrenOnOahu = 3
AdultsOnOahu = 2
ChildrenOnMolokai = 0
AdultsOnMolokai = 0

The algorithm will initiate the first while loop to send Adults over because AdultsOnOahu is greater than 0. Firstly, 2 children will cross so that ChildrenOnOahu=1, ChildrenOnMolokai=2 and BoatOnOahu=False. Then a pilot child will return to Oahu, ChildrenOnOahu=2, ChildrenOnMolokai=1 and BoatOnOahu=True. Then an adult will cross: AdultsOnOahu=1, AdultsOnMolokai=1 BoatOnOahu=False.There is still one Adult left so the while loop will run again, but this time with the if statement because BoatOnOahu=False. A child will row back to Oahu: ChildrenOnOahu=3, ChildrenOnMolokai=0 and BoatOnOahu=True. Then 2 children will row back again to Molokai: ChildrenOnOahu=1, ChildrenOnMolokai=2 and BoatOnOahu=False. Then a solo pilot child will row back to Oahu: ChildrenOnOahu=2, ChildrenOnMolokai=1 and BoatOnOahu=True. Then the last adult will ride over: AdultsOnOahu=0, AdultsOnMolokai=2 and BoatOnOahu=False. Now since all adults are across and ChildrenOnOahu=2, the second while loop will run with the if statement because of BoatOnOahu=False. Finally the child takes the boat from Molokai to Oahu: ChildrenOnOahu=3, ChildrenOnMolokai=0 and BoatOnOahu=True. Then 2 children will go across:

ChildrenOnOahu=2, ChildrenOnMolokai=1 and BoatOnOahu=False. One child will go back to Oahu: ChildrenOnOahu=2, ChildrenOnMolokai=1 and BoatOnOahu=True. Then to finish the 2 children will go across: ChildrenOnOahu=0, ChildrenOnMolokai=3 and BoatOnOahu=False. TotalChildren and TotalAdults are equal to ChildrenOnMolokai and AdultsOnMolokai respectively. Therefore Test Case 4 passes.