University of California Merced CSE150

Design Document For
Nachos Program (Phase 1)

Jonathan Srinivasan
Paulo Alberto Sousa
Erick Uriostegui
Johnathan Vastola
Alexander Woersching
Morgan Trembley

October 2020

1. Overview

Our goal is to implement a completely functional program. We are managing threads through different functions. With 6 main tasks for different functionality in our operating system.

2. Content

Task 1: KThread.join()
- join()
- finish()

Task 2: Condition2
- sleep()
- wake()
- wakeAll()

Task 3: Alarm
- timerInterrupt()
- waitUtil()

Task 4: Communicator
- speak(word)
- listen()

Task 5: Priority Scheduling
- donationUpdate()
- nextThread()
- pickNextThread()
- getThread()
- getEffectivePriority()
- setPriority(int)
- waitForAccess(PriorityQueue)
- acquire(PriorityQueue)
- updatePriority()

Task 6: Boat

3. Design Questions

Task 1: KThread.join()

The KThread.join() function needs to check if the thread being joined is finished, if the thread has already been joined, and if the thread is joining itself. Assuming it isn't, then the parent thread must wait for the child thread to finish before it can resume its operations. In order to do this, interrupts need to be disabled once the function is called, and the parent needs to be added to the child thread's joinQueue and put to sleep. However, in order to reawaken the parent thread when the child has finished, the finish() function needs to be modified to check if the child's joinQueue was empty, and when it was proven to not be empty, ready all parent threads within the queue before putting the child to sleep permanently.

```
join() {
      disable interrupts
      If this thread has already been joined AND this thread isn't
the current thread
            return; //Don't join again
      If this thread is not finished
            add currentThread to joinQueue
            put currentThread to sleep
            mark this thread as joined
      enable interrupts
}

finish() {
      mark this thread to be destroyed
      if(currentThread.joinQueue is not empty){
            ready all threads in the queue
      }
      sleep();
}
```

Correctness:

- Checks to make sure it isn't incorrectly joining.
    - Child isn't equal to the parent.
    - Child isn't already finished.
    - Child isn't already joined.
- Allows child threads to join to other threads.
- Allows parents to properly join to multiple threads.
- Always makes sure child is finished before readying parent.

Testing:

- joinTest1() joins main thread to parent thread, and then joins said parent thread to child thread. The test expects the child to finish first, then parent, then main.
- joinTest2() joins main thread to thread1, then joins main to thread2. The test expects main to wait for thread1 to finish, then join to thread2 and wait for thread2 to finish before running.

Task 2: Condition2

Condition utilizes Sephamore, and is implemented in three main ways. The constructor creates a list of Semaphore waiters and associates the Lock to this condition object, sleep() adds a Semaphore to the waitlist then puts it to sleep using Semaphore.P(), wake() simply goes through the linked list of Semaphore objects and wakes the next one in the list, and wakeAll() carries out the same purpose but wakes all Semaphore objects in the waitlist. Since all Semaphore objects within the Condition class are initialized to a value 0, then Semaphore.P() is almost immediately called, resulting in the P() code when the value is 0, entailing adding the current thread to a waitlist then putting it to sleep. Similarly, the purpose of Semaphore.V() in Condition removes threads from this queue and puts them in a ready state.The default implementation of Condition utilized a LinkedList to implement a queue of Semaphore objects that each contain a list of one thread.
This same implementation was used for Condition2 by disabling interrupts directly, rather than using Semaphores. Using the functionality from the Semaphore class without using an initialized value, and the structure of the original Condition class, Condition2 carries out all the same methods directly. sleep() adds the current running thread to the waitlist and puts it to sleep by calling the KThread sleep() function. wake() and wakeAll() both awaken one of all threads in the waitlist respectively by checking if the waitlist is occupied by any threads, and removing them as necessary based on which function is called.

```
Condition2.sleep() {
	Throw exception if lock isn't held by the current thread
	Release lock
	Disable interrupts
	Add currentthread to waitQueue
	Sleep
		(on awake)
		Enable interrupts
		Reacquire lock
}
Condition 2.wake() {
	Throw exception if lock isn't held by the current thread
	Disable interrupts
	If (waitQueue isn't empty)
		Ready and remove next thread in waitQueue
	Restore interrupts
}
Condition2.wakeAll() {
	Throw exception if lock isn't held by the current thread
	Disable interrupts
```

```
                while (waitQueue isn't empty)
                        Ready and remove next thread in waitQueue
                Restore interrupts
        }
```

Testing:

-Replace Condition to ConditionNew and confirm expected behavior

-Use multiple threads to sleep on one Condition and confirm wakeAll properly wakes all threads, use Kthread.join() statements

Task 3: Alarm

The current/temporary implementation is as follows:

```
 waitUntil(long x) {
//for now,cheat to get something working (busy waiting is bad)
      wakeTime = time + x;
      while (wakeTime > times)
          KThread.yield;
}
```

While this works for now, we need to avoid the busy-waiting caused by constantly yielding the thread.

Solution:

We should create a sleeping queue for any threads that need to go to sleep. We add threads to this queue when they call waitUntil and then use the timerInterrupt function to check if we need to wake them up. This will be a priority queue with the priority based on minimum wake up time. A priority queue with this setup allows us to be certain that the thread closest to its wake up time will always be at the front with easy access.

When a thread wants to go to sleep, we need to add it to the queue with its wake up time. We are passed in a x variable which represents the amount of ticks the thread wants to sleep before being awoken again. This means that our true wake up time for the thread will be calculated by getting the current time + the x requested ticks. With that calculated, all we need to do is add the thread and its wake up time to the queue.

```
//Called when a thread wants to go to sleep for x ticks
waitUntil(long x):
      If x is 0 or negative -> return
      sleepingQueue.add(currentThread, currentTime + x)
```

With our setup, in order to check if we need to wake up a thread we only need to check the front-most object on the sleeping queue. If it's associated wake up time has passed the current time, AKA when wakeUpTime < getTime(), then we wake it up by putting it on the ready queue. In realistic implementation, it may be useful to do the checking of the timers in a while loop.

This way if you get a situation where multiple threads need to be awoken at around the same time, they all get awoken at the same time instead of waiting another 500 ticks.

```
//This func is called every 500 ticks
timerInterrupt:
      Yield Current Thread
      Check the thread on top of the priority queue
            if (getTime() > thread.expiration timer), release it
            If not, continue
```

Correctness:
When a thread calls waitUntil with a parameter of 0 or a negative value, then the thread doesn't wait and is immediately returned. If a thread calls waitUntil with a valid value, then the thread waits for a *minimum* of that time (in ticks) and is then woken up.

Testing:
Test with a variety of waitUntil requests including negative, 0, and both small and large positive numbers.

Task 4: Communicator

The communicator class is implemented through two functions: speak(word) and listen(). Only one of each function should be active at a time, while the other speakers and listeners are waiting in the wait queue. Therefore, we need a lock. Here are the variables needed for

```
communicator(){
      Lock; //for both listener/speaker;
      Waiting queue for listener;
      Receive listener;
      Waiting queue for speaker;
      Send speaker;
}
```

Start the speak(int word) function with a putting on the lock (speakLock.Aquire()), which puts other speakers to sleep, making them wait. Speaker then receives the information by waking up the receiving listener (releaseListener.wake()). The speaker then sleeps and while being sent to the end of the waiting queue. Then if the speaker and listener are not waiting, both the speaker and listener will be woken up (speakerWaitingQueue.wake(); (listenerWaitingQueue.wake()) and will keep going through the same process till none is left. Let go of the lock at the end (releaseSpeaker.lock()).

```
speak(int word){
      speakerLock.Aquire(); //will make other speakers wait
      Put speaker in waiting queue;
      while(speaker is waiting){
            if(no waiting listener){ //no message
                  Wake up receiving listener;
                  Place send speaker to the sending queue;
            }
      }
      while(listener is not waiting or the message not received){
                  Wake up the waiting speaker;
                  Wake up the waiting listener;
            }
      Wake up speaker waiting queue;
      Wake up listener waiting queue;
      releaseSpeaker.lock();
}
```

The listen() function will also start with putting on the lock. Then listen() will go through the exact same process as speak(), except a word will be returned at the end, right after earth lock is released (releaseSpeaker.lock()).

```
listen(){
      listenerlock.Aquire();//will make other listeners wait
      Put listener in waiting queue;
      while(listener is waiting){
            put  listener waiting queue to sleep;
      }
      while(no waiting speaker){
            Put receiving listening in waiting queue; //sleep
      }
      Wake up speaker in queue;
      Release lock;
      Return word;
}
```

Correctness:

Able to share message using communicator()

Only one speaker and listener, but multiple threads can be waiting to speak() or listen()

Testing:

-Test with only one speaker and one listener

-Test with multiple speakers and multiple listeners

Task 5: Priority Scheduling

Goal:

A scheduler that chooses threads based on their priorities. For Priority Scheduler, we will have to implement priority scheduling, priority inversion, and priority donation. For priority scheduling, we will have three different threads and our task is to make the program choose the thread with the highest priority. For priority inversion, we will make our program choose the thread with the lowest priority first. For priority donation, the thread with the highest priority will donate its priority to the thread with the lowest priority.

Outline:

```
PriorityQueue
      donationUpdate()update resource owner from donations/depending threads
      waitForAccess(KThread)     call ThreadState function
      acquire(KThread)           call ThreadState function
      nextThread()               call pickNextThread
      pickNextThread()           get highest priority element
      Bool transferPriority;     if queue should transferPriority from waiting threads
      Treeset Waiting;           The main treeset of waiting threads
      ThreadState owner;         Threadstate indicating which thread acquires priority
ThreadState
      getThread()                return the current thread related to threadstate
      getPriority()              return priority of threadstate
      getEffectivePriority()     return effective priority of threadstate
      setPriority(int)           set priority of associated thread to value
      waitForAccess(PriorityQueue)
      acquire(PriorityQueue)     acquire what's guarded by PriorityQueue
      updatePriority()           update effective priority for this threadstate
      KThread thread;            The thread with which this object is associated.
      int priority;              The priority of the associated thread.
      int ePriority;             The effective priority of the associated thread.
      int waittime;              The time thread began waiting.
      PriorityQueue waitingFor;  The ThreadQueue the associated thread waiting for.
      HashSet owned;             The hashset sorting donated priorities
```

Implementation:

The PriorityScheduler class contains PriorityQueue with a sorted TreeSet.
If a queue allows for priority donations, the thread currently with access to the resource receives the highest effective priority in that queue. In calculating effective priority a thread must update and check all threads donating to it.

```
updatePriority(ThreadState) {
      for (every resource owned by this thread)
            For (every thread waiting)
                  effectivepriority = max(effectivepriority,
                  oldhighestpriority)
            Update priority of owner
      }
}
```

We need to ensure we update the resource owner from donations. We do this by iterating through owned queues and updating waiting threads priority and then finding the highest priority for the owning thread.

```
donationUpdate(){
            if (owner){
                  Update owner priority
                  For each (owned resource queue)
            for(waiting threads)
                  Update depending threads priority
                  Owner effective priority= max(oldpriority,
                  threadpriority)
}
```

As long as the effective priority of a thread can be correctly determined, the TreeSet should place any thread in correct order. Then picking the next thread is simply picking the last element in the set with highest priority.

PriorityScheduler allows a thread to place itself on a queue as well as acquire a resource but there should not be duplicates. When entering a queue, a thread must update wait times and check for donation priority to the thread owner if needed.

```
waitForAccess(PriorityQueue) {
      if already waiting remove(currentthread,waitingQ)
      loop(other waiting threads, waittime++)
      add thread to priorityqueue
      waitingFor = PriorityQueue
      if (owned already), remove
      PriorityQueue.donationCheck since new thread is queued
}
```

When a thread gains access to a resource it must update others that may be affected by the change, checking if previously owned resource and removing it from the queue as well as repositioning previous owner. The thread adds a resource queue to the list of owned resources, removes resources as desired and updates priority.

```
acquire(PriorityQueue) {
      if (no change) stop waiting
      remove from waiting
```

```
            add to owned resources
            if (priorityqueue has an owner)
                  remove this queue from owned resource list of previous
            owner
                  PriorityQueue.updatePriority, reposition thread based
            on loss of resource queue
            add this queue as owned resource
            owner.updatePriority, reposition thread based on new
      priority
      }
```

With these methods properly implemented, priority scheduler can handle any number of independent queues and threads with varying priority levels, implementing donation or not. Finding the next thread is simple, we just pull the last element in the Treeset for the PriorityQueue, assuming we sorted everything properly.

```
      nextThread() {
            check if interrupts disabled
            picknextThread()
            if (no thread return null);
                  acquire nextThread
                  return thread;
            }
      pickNextThread() {
            return waiting.last() ? null
      }
```

Testing:

- ➢ It's important to consider different priority levels without donation to ensure proper selection within each level and sorting of threads properly
- ➢ Thread donating to other threads in a chained ownership to ensure donation propagation through multiple tiers of ownership
- ➢ Threads changing priority between resource owners first and second execution to ensure effective priority recalculated properly
- ➢ Ensure threads are not waiting for more than one resource at a time
- ➢ Ensure threads do not add themselves to a queue more than once
- ➢ Ensure calculating the correct effective priority for multiple queues

Task 6: Boat

Constraints:

1 adult can row the boat while no children are on
1 child can row the boat while no Adults are on
1 child can ride the boat while 1 child is rowing
Boat must be rowed from one island to the other then back
Boat must always have a pilot before departing

General Pseudocode:

```
While there are adults on Oahu
    Send 2 children to Molokai
    Send 1st child back to Oahu
    Send an adult to Molokai
    Send 2nd child back to Oahu
While there are more than 2 children on Oahu
    Send 2 children to Molokai
    Send 1st child back to Oahu
Send last 2 children to Molokai and end simulation
```

Global Variables:

Int: Adults on Oahu, Children on Oahu, Adults on Molokai, Children on Molokai
Boolean: Passengers, Boat location
Condition: Child waiting on Oahu, Child waiting on Molokai, Child waiting for partner, Adult waiting on Oahu
Semaphore: terminate(0) - .P() at end of begin method to avoid early termination

```
Adult Itinerary:
      lock
      Increment counter for adults on Oaho
      unlock
      Lock Oahu
      Sleep and wait for child to tell adult they are OK to travel
      Get on boat and row to Molokai
      Update counters as needed
      Unlock Oahu
      Lock Molokai
      Wake up child so they can row back to Oahu
      Unlock Molokai
```

Correctness:

All threads must be counted in the population before starting

This method is only called when there is at least 1 adult on Oahu

Adult Itinerary only runs when everything is set for an adult to travel to Molokai

- A child is ready to return the boat to Oahu when an adult is woken

```
Child Itinerary:
      lock
      Increment counter for children on Oahu
      unlock
      Wait to synchronize populations of Oahu
      While unfinished
            While (adults on Oahu > 0)
                  Lock Oahu
                  If (passengers > 1) then, sleep
                  If (passengers == 0)
                        Get on boat (increment passengers)
                        Wake another child
                        Wait for them to get on boat by sleeping
                        Ride to Molokai
                  Else
                        Get on boat (increment passengers)
                        Wake waiting partner
                        Row to Molokai
                        Change boat location to Molokai
                  Update all counters
                  Unlock Oahu

                  Lock Molokai
```

```
                        Get off boat (update counters for boat and Molokai
populations)
                        If (passengers == 0) //should always be the case
                              Get on boat and prepare to row back
                              Row back to Oahu
                              Change boat location to Oahu
                              Update counters
                              Unlock Molokai
                              Lock Oahu
                              Get off boat and update Oahu population
                              Wake adult to travel to Molokai
                              Unlock Oahu
                        Else
                              Sleep //will be woken by adult so prepare
                              Get on boat and prepare to row back
                              Row back to Oahu
                              Change boat location to Oahu
                              Update counters
                              Unlock Molokai
                              Lock Oahu
                              Get off boat and update Oahu population
                              Unlock Oahu
                        Wait some time to keep threads synced

                  While (children on Oahu > 2)
                        Lock Oahu
                        If (passengers > 1) then, sleep
                        If (passengers == 0)
                              Get on boat (increment passengers)
                              Wake another child
                              Wait for them to get on boat by sleeping
                              Ride to Molokai
                        Else
                              Get on boat (increment passengers)
                              Wake waiting partner
                              Row to Molokai
                              Change boat location to Molokai
                        Update all counters
                        Unlock Oahu

                        Lock Molokai
```

```
                        Get off boat (update counters for boat and Molokai
populations)
                        If (passengers == 0) //should always be the case
                              Get on boat and prepare to row back
                        Else
                              Sleep
                        Row back to Oahu
                        Change boat location to Oahu
                        Update counters
                        Unlock Molokai
                        Lock Oahu
                        Get off boat and update Oahu population
                        Unlock Oahu
                  //send last 2 children to molokai
                  Lock Oahu
                        If (passengers > 1) then, sleep
                        If (passengers == 0)
                              Get on boat (increment passengers)
                              Wake another child
                              Wait for them to get on boat by sleeping
                              Ride to Molokai
                        Else
                              Get on boat (increment passengers)
                              Wake waiting partner
                              Row to Molokai
                              Change boat location to Molokai
                        Update all counters
                        Unlock Oahu

                        Lock Molokai
                        Get off boat (update counters for boat and Molokai
populations)
                        Unlock Molokai
                  Mark sim as finished
            Terminate simulation
```

Correctness:

This thread controls the synchronization for both threads, ensuring the populations are tallied before starting and waiting for threads to switch and process without timing issues.

Each portion of the child itinerary only allows 2 threads to process at a time.

There needs to be a wait after the population counter to ensure everyone is accounted for before doing anything else.

There needs to be synchronization between 2 children when they go to Molokai together.

There also needs to be synchronization between the adult thread and child threads so one doesn't overtake the other and give the incorrect instruction order.

The child portion runs only after the adults have all been moved to Molokai.

Testing:

First we check that the methods gathered the right total populations correctly before starting to process. We used the extremes of 246 adults/2 children and 248 children to verify this. Then we wrote the child itinerary portion that deals only with children and tested with 2 children alone, then 248 children, then a few random numbers to ensure it was flexible. Then we wrote the adult itinerary and child itinerary portion that dealt with adults on Oahu. We tested this first with 1 adult and 2 children, then 246 adults and 2 children, then 124 of each. Once we had the extreme cases working we picked a few random odd/even/prime numbers of each to ensure the code worked properly.

Design Questions

---

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

> Semaphores don’t have a thread that has acquired the resource. It doesn’t make sense to donate priority. The priority would be donated by default to the last thread woken by Sephamore.V()

2.A student proposes to solve the boat problem by using a counter, `AdultsOnOahu`. Since this number is not 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?

> The answer depends on its implementation. It should function normally. First we make sure the incrementing counter is atomic, also there must be synchronization so children don’t execute when there are adults that haven't added themselves to the counter.