**CSE 150 Operating Systems**

**Design Phase 1: Build a thread system**



**Authors: Team Thimble**

Andrew Yfantis

Isaac Flores

Landon Taylor

Lenny Khodel

**Task I:**

Implementing *join()* in the KThread class is to check if the current thread is finished so it knows to continue running. If the current thread is not finished it should interrupt the parent and put it on a waiting Queue. Our *join()* method was implemented to do two things: have the current thread wait for this thread to finish, and if the current thread is already finished it will return. Join must not be called upon the current thread and it must not be called more than once. At the end of the method the joined thread will have completed execution. The last requirement for join is that this thread must not be the current thread.

**public** **void** join() {

if(currentThread is completed)

{

Return;

}

Lib.debug(dbgThread, "Joining to thread: " + toString());

Lib.assertTrue(this != currentThread);

Join\_count++

disable interrupts

save currentTCB

put current thread into wait queue

Re-enable interrupts

**Task II:**

The Condition2 class is a class that disables interrupts for synchronization. There three methods being implemented in the Condition2 class are *sleep(), wake(),*  and *wakeAll()*.

The *sleep()* method is to automatically release any lock associated with this conditional variable and to have the current thread put to sleep until another thread wake it up. In order for sleep to be called, the current thread must hold the lock and after this method is called the associated lock must be released before sleep returns. Our *sleep()* implementation releases the lock first, puts current thread to sleep, reacquires the lock and then returns.

**public** **void** sleep(){

Disable interrupts;

Add itself to the waitQueue

Release the lock so someone else can access lock

Call sleep from kthread to actually go to sleep

Now another thread will run and when that thread is finished with lock wake the original thread up

Now its current threads turn to acquire the lock

Enable interrupts

}

The *wake()* method will wake one thread that previously called sleep. In order for *wake()*  to be called, the current thread must hold the associated lock. Our *wake()* implementation checks to see if any thread is sleeping and wakes it by putting it on the ready Queue.

**public** **void** wake(){

Current thread must hold the lock

Disable interrupts;

Check waitQueue to see if any threads are waiting on the lock

If there is remove from waitqueue and call current thread’s ready function

Enable interupts

}

The *wakeAll()* method is to wake all threads sleeping on this conditional variable. In order for sleep to be called, the current thread must hold the lock. Our *wakeAll()* implementation checks to see if any thread is sleeping and wakes all threads by putting them all on the ready Queue

**public** **void** wakeall(){

Disable interrupts;

If( anything is sleeping)

{

Wake all up; //put in ready queue=wake

}

Enable interrupts;

}

**Task III:**

3: alarm

we maintain a wait queue which contains sleeping threads sorted by wake time. the thread with the lowest wake time is at [0] of this queue.

alarm contains 2 functions, waitUntil and timerInterrupt

waitUntil calculates the minimum time the current thread should be woken up.

it sets the member variable in the KThread class, which is only accessed by the timerInterrupt. the current thread is added to the wait queue and waitUntil returns void.

waitUntil(waitTime)

machine.timer.getTime() + waitTime -> KThread.currentThread.waitUntil

waitQueue.add(KThread.currentThread)

KThread.currentThread.sleep()

timer interrupt is called by the machine timer and wakes up any threads which are waiting a certain time. if the queue is empty the loop breaks

since threads are sorted by wake time, the while loop checks the first thread and readies and removes it if its wake time is lower than the current time. then the first thread becomes the next thread to be woken up. if the first thread isn’t to be woken up yet, none of the others are either, since the queue is sorted. if the queue is empty, the loop ends and the current thread yields.

timerInterrupt()

while(waitQueue[0].waitunitl < machine.timer.getTime & waitQueue ! empty)

waitqueue[0].ready()

remove waitQueue[0]

kthread.currentThread.yield()

**Task IV:**

The communicator class implements synchronous send and receive of one word messages. It does this by maintaining the following:

A private class ResourceWrapper which encapsulates a condition variable and the integer word

A linked list of Speakers (message senders)

A linked list of Listeners (message receivers)

A static lock object

A speak method

A listen method

The speak(int word) method works as follows: An integer word is passed in as a parameter, representing the word to be spoken. The class-scope lock is immediately acquired and the listen queue is checked. If there is a listener waiting for a message, interrupts are disabled, the first waiting listener is removed and then interrupts are re-enabled. Next, the listener's word is set to the input value, and the next thread waiting in the condition variable is woken up. Finally, the lock is released.

If a there are no listeners available when the speaker speaks, then a new ResourceWrapper is allocated for the speaker and its word is set to the input parameter. The ResourceWrapper is then added to the the list of Speakers and the current thread is put to sleep on the condition variable.

The listen() method works as follows: First, the lock is acquired. Next, the word is initialized to 0 and the list of speakers is checked. If there is a speaker waiting, the first speaker is removed and the word is copied to the local word. Then, the next thread waiting on the condition variable is woken up.

If there are no speakers waiting, then a new ResourceWrapper is allocated to store the current listener, and is put into the list of listeners. Then the listener goes to sleep on the condition variable. When the thread wakes up, it sets its word to the value placed there by the speaker.

Finally, the lock is released and the stored word value is returned. Thus, word will only ever be returned once is has been set. Either the listener sleeps while waiting for the message, or it immediately pairs of with a speaker.

The ResourceWrapper class encapsulates an integer word, representing the message to be spoken and a condition variable. The default constructor initializes these to 0 and uses the class-scope lock as the lock around which the condition variable holds a queue. getCondition() and getWord() return the condition variable and word, respectively, and setWord() sets the word.

**Task V:**

The priority scheduler implements functions that manage scheduling of the kernel threads to ensure threads with higher priorities execute first. In a priority based scheduler all thread in waiting Queue are sorted by their effective priority. All threads store a sorted map of queues they own and the highest priority thread stored in the Queue(priority cache). The effective priority of a thread is the highest priority stored in its priority cache. When a thread is selected from the Queue to acquire the resource *nextThread()* that thread is removed from the Queue and the new thread is told to acquire *acquire()* the previous owner of the Queue is told to no longer own the Queue, and the cached priority in the Queue is removed. The new owner of the Queue gets the highest priority in the Queue added to its priority cache and its effective priority updated. When a thread is added to the waiting Queue *waitforAccess()* it may have the highest priority of all threads in the Queue, the owner of the Queue then needs to update its effective priority to be greater than or equal the new waiting thread. When a change occurs to a thread’s effective priority, it must reinsert itself into the correct position in the Queue and may result in affecting the effective priority of that Queues owner.

*nextThread()* will check to see if owner has a value, if it does then remove it. It will also check the list of threads on Queue if empty it will reset everything and exit method. It then checks the list of threads on the Queue , if empty it will reset everything and exit the nextThread method. The owner will the obtain the thread that is chosen by pickNextThread and check if it is null which is important to know if an idle thread was created. The owner will receive all the information, priorities and permissions and be removed from threadlist. The owner will then be returned.

**public** **void** nextThread

disable interrupts

set a kthread object equal to the result of function pickNextThread()

check if it is null

if not null acquire the thread

then return the gotten thread

*pickNextThread()* looks to see what next thread is, if there is nothing on the thread list then exit the method. Check all the priorities of the waitng threads. If there are equivalent prioritiestake the one with the longer waiting time. Need to implement for loop that goes through the threadlist total length. Next check the transfer priority to see if a transfer should be made from waiting thereads.. If so transfer the effective priority, if not 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 who was there the longest. Will return the thread that has the highest priority, but not remove it. This is just to see whats next.

**public** **void** pickNextThread

make a Kthread object

set it to null

have a loop search through the waitQueue and compare all threads effective priorities so we can select the one with the highest priority

return the thread

*GetEffectivePriority()* returns the priority of the thread after taking into account the donations. It checks if the effective property is the recalculated priority, if so the its been properly updated, otherwise skip this portion of code. If not the same start a loop from 0 to size of the list, the waitQueue Queue will obtain the values in the list each time it increments to the next value and also contain threads from the Queues.

Start the next loop and current priority will get the priority of the j value and check to see if it is the maximum value. If it is greater than the maximum it will set the effective priority to that value.After check if the effective priority has been changed, if not the effective priority will just be the priority of the current thread and the effective priority will be returned.

**public** **int** getEffectivePriority() {

PriorityDonation donation = priorityCache.last();

**return** donation.priority;

}

**protected** **class** PriorityDonation{

**public** **int** priority;

**public** PriorityQueue queue;

}

*setPriority()* will set the priority of the associated thread to the specified value. In the base code need a check to make sure its not greater than 7 or less than 1, and update the effective priority.

**public** **void** setPriority(**int** priority) {

**if** (**this**.priority == priority)

**return**;

revokeDonation(**null**);

registerDonation(priority,**null**);

}

*waitforAccess()* passes in the waitqueue which does nothing as we just save the current time into the waiting time

**public** **void** waitForAccess(PriorityQueue waitQueue) {

**assert**(waitQueue != **null**);

//System.out.println(thread+" started waiting for "+waitQueue);

waiting = waitQueue;

requeue(waiting);

}

*Acquire()* add a thread to the acquired list. Reset the effective priority

**public** **void** acquire(PriorityQueue waitQueue) {

**assert**(waitQueue != **null**);

//Make sure this thread is completely removed from the queue:

//(before any donation changes are made)

dequeue(waitQueue);

//If there was a previous owner, perform handover:

//(only if there was a donation)

**if**(waitQueue.currentOwner != **null** && waitQueue.transferPriority) {

waitQueue.currentOwner.revokeDonation(waitQueue);

waitQueue.currentOwner = **null**;

}

//Declare ownership of this resource

waitQueue.currentOwner = **this**;

//If we were waiting on this resource then stop:

**if**(waiting == waitQueue){

waiting = **null**;

}

//Perform donation if necessary:

**if**(waitQueue.transferPriority){

registerDonation(waitQueue);

}

}

**Task VI:**

1. The boat problem. Many people are stuck on the island of Oahu (at least two children), and need to go to the island Molokai on a single boat. This boat can only hold up to two children, or one adult. Example 3 people waiting:
2. Two children row to Molokai
3. One child rows back to Oahu
4. One adult rows to Molokai
5. The child that was left rows back to Oahu
6. Both children go to Molokai (done)

We can see there needs to be a loop where if an adult crosses, there must be a child waiting at Molokai to bring the boat back, if the adult was not the last person on Oahu.

Our test cases must result in the same amount of adults and children that start and end, and they move across within the defined parameters. Having an initial 2 children makes the variable of other threads a fairly straightforward.

static void AdultItinerary()

{

If(child at Molokai)

{

//get in boat as adult

//row across

//get out of boat

}

//countAdultMolokai++

//countAdultOahu--

}

static void ChildItinerary()

{

//one child keeps bringing other children across

//brings back for adult(s) near end of completion.

While( people at oahu)

{

If( at oahu)

{

If( boatempty)

//join as passenger || pilot //(is one more preferable?)

If( boatpassenger)

{

//Join as pilot

//row to molokai

}

If( pilot)

{

//Join as passenger

//signal (pilot)

/sleep

}

Sleep()

}

Else //at Molokai

{

If(boatempty)

//join as pilot //must be alone

If(boatpassenger)

{

//row to molokai

countChildOahu--;

countChildMolokai++;

}

If(amPilot)

{

//make sure there are no passengers

//pilot to oahu

}

}

}

**Task VII:**

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

Semaphores are a powerful tool. Sometimes they are not the right tool for a specific job. It is unfortunate to implement priority donation for semaphores because of the way semaphores “access” the resource.

Currently, we can see when a high-priority thread is waiting on another thread with lower priority. With semaphores, that would go away; we would just know that a thread is waiting on another thread. This would create a problem with any thread gaining priority from the waiting thread, probably leading to starvation.

We would need to track any and all threads that would use the semaphore, and have a much more complex routines of donating priorities (i.e if a lower priority thread needs a donation from a thread that had gotten a donation already).

8. A student proposes to solve the boats problem by use of a counter, AdultsOnOahu. Since this number isn’t known initially, it will be started at zero, and incremented by each adult thread before they do anything else. Is this solution likely to work? Why or why not?

The solution is not likely to work with the example given. Most prominently is the fact that when the program is done, we will want to see AdultsOnOahu be equal to 0, but this is also the case when the boats problem starts. It is a useful idea, but alone will not solve the boat problem. Even if we do not know the starting number, the starting number exists, and should be used to avoid such a starting/termination conditions problem.