Operating Systems
Project 1

Mao Yihuan 2016010459
Ni Bonan 2016012139
Wang Yuanhao 2016012396

April 4, 2019

Task I

In KThread.java: We add a Semaphore called *joined* as member of KThread class, whose value is initialized to be 0. For each thread T, $T.joined$ has value 1 iff. T has finished running and hasn't let a $join()$ call return yet.
In line 163 $joined.V()$ is called after $run()$ returns, meaning that the run finishes.
In line 283 we call $joined.P()$, which simply checks waits for *joined* to be equal to 1, then return (We assume each thread to be joined at most once, therefore don't care what happens after this $P()$).

Tests

In KThread.java: We implement *Runnable* into class $Joiner$ (line 405), which takes in another thread, sequentially calls its $fork()$ and $join()$, then terminates. In $selfTest()$ (line 444) we create two threads, one is a $PingTest$ and the other is a $Joiner$ with the $PingTest$ instance passes into it. Then we $fork()$ the second thread, then $join()$ it.
We can also add an $yield()$ call in $Joiner.run()$ after it forks the new thread to be joined later (line 413), which creates a context switch, and call $join()$ afterwards. This represents the case where a thread finishes first, then its *joined* method is called.

Task II

In Condition2.java: The condition variable class contains a lock and a queue of waiting threads, which are separately a *Lock* instance *conditionLock* and a $ThreadQueue$ instance *waitQueue* specified by the chosen $scheduler$.
In $sleep()$, $wake()$ and $wakeAll()$ we first check whether current thread holds *conditionLock*, which is required by the definition of condition variable. All following operations are wrapped by disabling machine interrupts except the last line of $sleep()$.
In $sleep()$ the current thread puts itself into the waiting queue, then release the lock and call $KThread.sleep()$. After waking from sleep, we first enable machine interruption (line 44), then acquire the condition lock (line 46). This is because the last operation does not require mutual exclusion, typically in the case after a $wakeAll()$ call.
In $wake()$ we check the next thread in the *waitQueue* using its $nextThread()$ method: if such a thread exists (the queue is not empty), we put it to the ready queue. Note that the $nextThread()$ method of $ThreadQueue$ removes the first thread from the queue, therefore we do not need do manually do deletion.
In $wakeAll()$ we simply modify $wake()$ into repeatedly call $waitQueue.nextThread$ and put the returned thread to the ready queue until *waitQueue* becomes empty.

Tests

To test Condition2, we implemented the consumer-producer model, one of the most basic examples for using a monitor. The resutls are in testII.txt. The code are in Condition2.test1() and Condition2.test2(). We used a StringBuilder instance to mimic a thread-unsafe queue. Consumers read chars from the start, while producers append chars to the end. The purpose of the tests is to make sure that Condition2 works in the most basic textbook example.

In the first test, we let a reader reading 5 chars fork() first, and then let a writer fork(). We check whether the reader sleeps when finding the string empty, and whether both threads output correct results.

In the second test, we have a reader reading 6 chars forking first. Then, separated by some time, we have three writers each writing 2 symbols. We check whether the reader sleeps each time it reads all the remaining chars.

Task III

In Alarm.java: We define class $WaitingThread$ which implements Java $Comparable$. A $WaitingThread$ is a (thread, time) pair, representing a thread calling $waitUntil()$ of this $Alarm$ instance ($WaitingThread.thread$) and the time to wake from its $waitUntil$ call ($WaitingThread.wakeTime$). We override its $compareTo$ method into directly comparing $wakeTime$. Then we build a Java $TreeSet$ with $WaitingThread$ elements ($waitingThreadSet$, line 101), which automatically sorts elements in increasing order of $wakeTime$.
Pseudocode for $waitUntil()$:

```
waitUntil(x) {
    wakeTime = current_time + x
    if (wakeTime > current_time) {
        disable machine interrupt
        waitingThreadSet.add(new WaitingThread(current_thread, wakeTime))
        sleep()
        restore machine interrupt status
    }
}
```

Note that we first do the if check, then disable machine interrupt and do critical operations. This avoids unnecessary overhead if a context switch happens after the second line (line 66) in code, or passed-in x is very small or even negative.
In $timerInterrupt$ we wrap everything with disabling machine interruptions. We use Java iterator to iterate through $waitingThreadSet$, which is already sorted, therefore all threads that should be woken are on its head. The pseudocode is here:

```
timerInterrupt() {
    disable machine interrupt
    Iterator iter = waitingThreadSet.iterator()
    while (iter.hasNext()) {
        entry = iter.next()
        thread = entry.thread
        wakeTime = entry.wakeTime
        if (current_time > wakeTime) {
            thread.ready()
            iter.remove()
        } else {
            break
        }
    }
    yield()
    restore machine interrupt status
}
```

Tests

In KThread.java: Implements *Runnable* into class *AlarmThread* (line 421), which calls *Alarm.waitUntil*(). In selfTest() we create 5 threads (line 449), all run *AlarmThread* waiting for 20000 system ticks. Then call *Alarm.waitUntil*() to wait for 10000000 ticks to ensure that all 5 created threads terminate.

Task IV

In Communicator.java: A Communicator class contains a passed-in *Lock* instance *conditionLock*, four condition variables *speakerQueue*, *listenerQueue*, *speaking*, *listening* (implemented using *Condition*2), two integers *numSpeaker*, *numListener* both initialized to be 0, and an int value *word*.
We first give pseudocode, then explain:

```
speak(word) {
    conditionLock.acquire()
    if (numListener == 0) {
        numSpeaker++
        speakerQueue.sleep()
        numSpeaker−−
    } else {
        listenerQueue.wake()
    }
    while (transferring) {
        speaking.sleep()
    }
    this.word = word
    transferring = true
    listening.wake()
    conditionLock.release()
}

public int listen() {
    conditionLock.acquire()
    if (numSpeaker == 0) {
        numListener++
        listenerQueue.sleep()
        numListener−−
    } else {
        speakerQueue.wake()
    }
    while (!transferring) {
        listening.sleep()
    }
    heard = word
    transferring = false
    speaking.wake()
    conditionLock.release()
    return heard
}
```

At any time, assuming no more speakers and listeners will come, the number of existing speakers and listeners which will eventually pass the first $if-else$ clause are the same. We call such speakers and listeners **active**. This means we can pair the existing active speakers and active listeners.
$transfer$ represents whether *word* is holding unread message. If so, a speaker cannot overwrite that, and has to wait in *speaking* until a listener finishes its reading. Afterwards, a speaker can modify *word* and

$transfer$, wakes a sleeping active listener in $listening$ if there's one, then return. There is already another active listener in the system by our argument above, so the message will be found by an active listener.
An active listener, after passing the $if - elseclause$, first check whether $transfer$ is true: If so, it can directly read $word$ because of mutual exclusion guaranteed by condition lock, then put $transfer$ back to false. Otherwise, it has to wait in $listening$ for $transfer$ to become true, then read and modify $transfer$. Then call $speaking.wake()$ to announce that $transfer$ has become false.

Tests

We test our implementation by testing the behavior of $Communicator$ in three scenarios. The results are in testIV.txt.

In the first test (code in Communicator.test1()), we test the scenario where speakers wait for listeners. We let two threads speak first, and then let two threads listen. The two speakers speak different words. We do so by putting a ThreadedKernel.alarm.waitUntil() between creating the speaker threads and the listener threads. We check if all threads output correct information and exited in correct order and whether the listeners heard different words.

In the first test (code in Communicator.test2()), we test the scenario where listeners wait for speakers. We let two threads call listen() first, and then let two threads speak. The two speakers speak different words. We do so by putting a ThreadedKernel.alarm.waitUntil() between creating the listener threads and the speaker threads. We check if all threads output correct information and exited in correct order and whether the listeners heard different words.

In the third test (code in Communicator.test3()), we test the scenario threads may join each other. We have a listener $L1$ and two speakers, $S1$ and $S2$ where $S1$ joins $L1$ at the start. Therefore, $L1$ should always hear the word from $S2$ when working correctly.

Task V

As a quick overview, for every $ThreadState$ we maintain a collection $acquiredResource$ (HashSet) of acquired queues and a waiting queue $waitingResource$ (if any). For every $PriorityQueue$, we maintain a linkedlist $waitingThreads$ of threads waiting on the queue, and a ThreadState $holder$ (if any). We maintain an up-to-date effective priority for every thread state. It can be calculated by taking the max from the effective priority of all threads waiting for a queue in $acquiredResource$. Whenever a thread T's effective priority changes, we attempt to update the effective priority for the holder of the queue that T is waiting on. Whenever a new thread waits for access to a queue and has higher effective priority than the queue holder, we update the effective priority of the queue holder.

More specifically, a $PriorityQueue$ has three important methods that may be called by a user, namely $waitForAccess(KThread)$, $acquire(KThread)$ and $nextThread()$. This is how we implement them. For simplicity, we assume $transferPriority$ is $true$ when describing our implementation (the $false$ case is much simpler).

```
void PriorityQueue::acquire(KThread thread) {
    getThreadState(thread).acquire(this)
    this.holder = getThreadState(thread)
}

void PriorityQueue::waitForAccess(KThread thread) {
    ts = getThreadState(thread)
    ts.waitForAccess(this)
    if ((this.holder != null) & (ts.effectivePriority > this.getThreadPriority())){
        append ts to waitingThreads
        holder.calcEffectivePriority()
    }else{
        append ts to waitingThreads
    }
}
```

Here, getThreadPriority() is a new procedure. It is used to compute the maximum of effective priority among all threads in *waitingThreads*. We do this via a iteration of *waitingThreads*.

```
KThread PriorityQueue::nextThread() {
    ThreadState nextT = ThreadState in waitingThreads with largest effectivePriority, pick the first
one when tied
    if (holder != null){
        this.holder.relinquish(this)
    }
    if (nextT==null){
        this.holder = null
        return null
    }
    nextT.acquire(this)
    this.holder = nextT
    return nextT.thread
}
```

We now describe how we implemente the ThreadState class. Important methods include $acquire()$, $calcEffectivePriority()$, $getEffectivePriority()$, $setPriority()$, $waitForAccess()$ and $relinquish()$. Important new member variables include *acquiredResource* (stated above), *waitingResource* (stated above) and $effectivePriority$, the up-to-date effective priority.

```
void ThreadState::acquire(PriorityQueue waitQueue) {
    Add waitQueue to acquiredResource
    waitingResource = null
    if (waitQueue.holder!=null)
        this.calcEffectivePriority()
}

int ThreadState::calcEffectivePriority() {
    int maxPriority = priority;
    for (PriorityQueue pq in acquiredResource){
        maxPriority = max{maxPriority, pq.getThreadPriority()}
    }
    if ((maxPriority != this.effectivePriority) && (this.waitingResource != null)){
        this.effectivePriority = maxPriority
        if (waitingResource.holder!=null)
            waitingResource.holder.calcEffectivePriority(this)
    }else{
        this.effectivePriority = maxPriority
    }
}
```

In $getEffectivePriority()$, we simply return the cached $effectivePriority$. In $waitForAccess(queue)$, we simply set *waitingResource* to *queue*.

```
void ThreadState::setPriority(int priority) {
    if (this.priority == priority) return
    this.priority = priority
    this.effectivePriority = calcEffectivePriority()
}

void ThreadState::relinquish(PriorityQueue acqQueue){
    this.acquiredResource.remove(acqQueue);
    this.calcEffectivePriority();
}
```

Tests

To help testing, we implemented the $PriorityQueue.print()$ method, which prints all ThreadState in $waitingThreads$. Each ThreadState will be printed as $threadName < priority, effectivePriority >$. In addition, it prints the current holder of the queue (in the same format). The test results are in testV.txt.

Using the $print()$ method, we test whether features are implemented correctly in the following steps:

- PriorityQueue picks a thread with highest (effective) priority;
- PriorityQueue breaks tie by choosing the thread waiting longest;
- effective priority is correctly calculated and transmitted, when a thread holds at most one queue;
- $setPriority()$ sets priority and effective priority correctly;
- effective priority is correctly calculated and transmitted, when a thread holds many queues;
- PriorityQueue functions without priority donation when $transferPriority$ is $false$.

In the first test (PriorityScheduler.test1()), we test whether a single PriorityQueue picks waiting threads in the same order as priority, and whether when two threads have the same priority, the thread waiting longer will be chosen. We do so by letting four threads with priority wait to access an acquired PriorityQueue, and check if they access the queue in the right order.

In the second test (PriorityScheduler.test2()), we test whether effective priority is appropriately updated and transmitted between queues. We do so by setting up two acqruied PriorityQueues, and let the holder of the first queue waits for the second queue. We then append queues of different priority to the two queues, and observe if the effective priorities are updated correctly.

In the third test (PriorityScheduler.test3()), we test whether when $setPriority()$ is used, effective priority is still correctly updated. Similar to the previous test, we use three PriorityQueues $Q1$, $Q2$ and $Q3$. Initially $t1$ acquires $Q1$, $t2$ acquires $Q2$, $t3$ acquires $Q3$; $t1$ waits for $Q2$, $t2$ waits for $Q3$. We then perform a series of $setPriority()$ calls and other operations, and observe whether the effective priorities are updated correctly.

In the forth test (PriorityScheduler.test4()), we consider the case where a thread holds multiple queues and may receive priority donation from them. Initially $t1$ holds $Q1$, $Q2$ and $Q3$; it then waits for $Q4$, which is held by $t5$. We then append one threads to $Q1$-$Q3$ each, with increasing priority. We observe whether the effective priority of $t1$ is updated appropriately.

In the fifth test (PriorityScheduler.test5()), we simply test whether the code of test2 works appropriately when $transferPriority$ is set to $false$.

Task VI

In this task we have two kinds of people, and children and adults can't take one boat. Since two children can take one boat, it's easy to transport children to Molokai. However, if there's no children on Molokai, we can't simply transport an adult to Molokai, because he'll have to return Oahu himself, and those are useless operations.

Therefore, to design a method for everyone to arrive at Molokai, we'll always need two children to transport adults one by one. In our algorithm, those two children are decided at the beginning of the process.

1. Find two children $qd1, qd2$. They're assigned important jobs to transport adults.
2. Transport all the other children to Molokai. The method is to repeatedly do $qd1 : bg.ChildRowToMolokai(), child : bg.ChildRideToMolokai(), qd1 : bg.ChildRowToOahu()$. (Two children goes, one child returns)
3. Transport all adults to Molokai. The method is to repeatedly do $qd1 : bg.ChildRowToMolokai()$, $qd2 : bg.ChildRideToMolokai()$, $qd1 : bg.ChildRowToOahu()$, $adult : bg.AdultRowToMolokai()$, $qd2 : bg.ChildRowToOahu()$. (Two children goes, one child returns, one adult go, one child returns)
4. At last the only two remaining children $qd1, qd2$ go to Molokai.

This algorithm will terminate because in both step 2 and 3, the number of children and adults decreases, and it'll eventually goes to step 4.