Project 1 Initial Design Document

Fang Zhang,
Qiushi Huang,
Zhengyang Song,
Zhaopeng Yan

March 31, 2015

1 Join

1.1 Design

When a thread A calls B.join(), to wait B finishes, it adds itself to B's waitqueue and sleeps. After B finishes, B will wake up all the threads in the waitqueue so that they can continue running.

1.2 Variables and functions

1.KThread::waitqueue: private variable in KThread, save the threads which are waiting the thread to finish.

2.KThread::join(): When the current thread wants to wait another thread B to finish, it will call B.join. The function will add the current thread to the thread's waitqueue and sleep if the thread doesn't finish.

3.KThread::finish(): when a thread finishes, it will call this function to wake up all the threads waiting it. It will let the threads in waitqueue call ready() to add to the ready queue.

1.3 Pseudo-code

Algorithm 1: KThread::join

```
disable interrupt
if this.statues ≠ statusFinished then
    add current thread to waitqueue
    sleep()
restore interrupt
```

1.4 Correctness constraints

1.If A joins B, A will never be scheduled before B finished.

2.If A joins B, A will continue running after B finished.

Algorithm 2: KThread::finish

```
the same as the existing code
if waitqueue isn't empty then
    for every thread i in waitqueue do
        i.ready()
sleep()
```

1.5 test strategy

1. Fork a thread which output when it begins and ends and let the current thread join it. Test whether the join function works.

2. Fork several threads and each thread joins the thread forked before it. Test whether the join works when there are threads join each other.

3. Fork a thread and let the current thread fork it two times. Test whether when a thread joins a thread which is already finished can work.

2 Condition variable

2.1 Design

We need to implement sleep(), wake() and wakeAll() methods for this question. For we are forbidden to use semaphore as what has been done in Condition.java. We need to find another way around.

To do this, we also need a wait queue to record all the threads that are waiting this condition variable by now. Rather than placing a corresponding semaphore in the queue, we put the corresponding thread itself.

2.2 Variables and functions

For waitAll(), it is just all the same as before. We call wake() for all the elements in the wait queue.

For sleep(), we just put the current thread to the wait queue, and make it sleep. In order to guarantee atomic condition, we acquire the lock and disable interrupt before operation while enable interrupt and release the lock after operation.

For wake(), we examine whether the wait queue is empty. If not, just remove the first thread from wait queue and put it into the ready queue. Note that before the operation we disable the interrupt before and enable the interrupt after.

2.3 Pseudo-code

2.4 Correctness Constrains

- For any time, only one thread is running
- When one thread sleeps, it will not wake until some thread wakes it

Algorithm 3: Condition::sleep

```
release the lock
disable interrupt
add current thread to wait queue
make current thread sleep
enable interrupt
acquire the lock
```

Algorithm 4: Condition::wake

```
if wait queue is not empty then
    disable interrupt;
    remove first element;
    put it into ready queue;
    enable interrupt
```

2.5 Test strategy

- init a lock, and init a condition variable using the lock
- fork several thread, call sleep() for all, test whether all threads go to sleep
- call wake() to test whether one of the sleeping threads will awake
- call wakeAll() to test whether all the sleeping threads will awake

3 Alarm

3.1 Design

We need to overwrite the waitUntil(long x) method in the Alarm class with no busy waiting strategy. The aim of this method is to suspend the execution until time has been now+x. When the time is right, we just put it into the ready queue rather than let it run immediately.

So we can add a wait queue to Alarm class, every time waitUntil is called, we just compute the corresponding wake time of that thread, and put the thread together with the wake time into the wait queue. Then let the thread sleep. Whenever there is a timer interrupt, we check the wait queue to see whether there is any thread is due. If so, we just put it into the ready queue.

Algorithm 5: Condition::wakeAll

```
while wait queue not empty do
    wake()
```

Algorithm 6: Alarm::timerInterrupt

```
get current time;
for i in wait queue do
    if wake time of i ¿= current time then
        put i into ready queue
```

Algorithm 7: Alarm::waitUntil(x)

```
get current time;
wake time = current time + x;
put current thread together with wake time into wait queue;
make current thread sleep
```

3.2 Pseudo-code

3.3 Correctness Constrains

- The thread will not run until wake time
- When it beyonds the wake time, the thread will be put into the ready queue

3.4 Test strategy

- fork several threads, and assign different x towards function waitUntil
- test whether all the threads waits at least x time and then wake up

4 Communicator

4.1 Design

We need to implement the Communicator() constructor, speak() for speakers and listen() for listeners.

4.2 Variables and functions

For Communicator(), we just need to initialize all the class members.

For listen(), we first acquire a lock, then check whether there is a word ready, if not, wake up a speaker and goes to sleep. When there is a word ready, it receive the word and mark it unready. Finally we release the lock.

For speaker(), we also acquire a lock, check whether there is no listener or whether there is a ready word, if so, we go to sleep. When there is listener and no word ready, we say a word and mark the corresponding flag, wake up all the listener. Finally we release the lock.

Notice that we use a word_ready flag to indicate whether there is a ready word, use num_listener and num_speaker to make the check easier. The wake and sleep operations are realized by two condition variables using the same lock.

4.3 Variables and functions

- num_listener: number of listeners
- num_speaker: number of speakers
- word_ready: indicate whether word is ready
- cond_listener: condition variable for listeners
- cond_speaker: condition variable for speakers
- word: the current word can be heared

4.4 Pseudo-code

Algorithm 8: Communicator::Communicator

```
num_listener ← 0;
num_speaker ← 0;
word_ready ← false;
word ← "";
init a lock;
init cond_listener and cond_speaker with the same lock;
```

Algorithm 9: Communicator::speak

```
acquire the lock;
num_speaker ← num_speaker + 1;
while num_listener is 0 or word_ready is true do
    go to sleep
word ← word of current thread;
word_ready ← true;
wake all the listeners;
num_speaker ← num_speaker − 1;;
release the lock;
```

Algorithm 10: Communicator::listen

```
acquire the lock;
num_listener ← num_listener + 1;
while word_ready is false do
    wake up all the speakers;
    go to sleep
get the word;
word_ready ← false;
num_listener ← num_listener − 1;
release the lock
```

4.5 Correctness Constrains

- If there are more than one listener and one speaker, then there will be a thread that are not blocked.
- If a speaker thread finished, then his word is heard and only head once
- If a listener thread finished, then he heard and only heard a word once

4.6 Test Strategy

- less listener than speaker
- less speaker than listener

For all the above cases, see whether the result is normal

5 Priority scheduling

5.1 Design

To implement the priorityscheduler, we only need to implement the threadstate class and priorityqueue class in priorityscheduler. We set two variables in threadstate: priority and effective.When we want to get priority or effectivepriority, we will return priority and effective. And we set a variable in priorityqueue to record the thread for which the queue is waiting. Then we create an interface setEffectivePriority() to set effective. Every time a new thread add to the queue, it will donate it's priority to it using setEffectivePriority(). Since when a thread's effectivepriority changes, it will affect the priority of the thread it waits, which means it must also call setEffectivePriority() from the threads it waits until the thread doesn't wait. To know what threads it is waiting, we can set a variable in KThread to record the waitqueue the thread is in. Notice that in task 1, if the waitqueue is priorityqueue, it can also donate priority.

5.2 Variables and Functions

- effecitve: protected variable in threadstate, save the effective priority of the associated thread.
- waitedthread: public variable in priorityqueue, save the thread that the threads in the queue is waiting for.
- waitqueue: private variable in priorityqueue, save the waiting thread in priority queue.
- waitingqueue: public variable in KThread, save the queue the thread is in. If the thread does't wait for any thread, the waitingqueue should be null.
- threadstate::setEffectivePriority(): When a thread wants to donate priority, it will call this function. We first compare the setting value to the original effective, if it's smaller, do nothing; else, set effective to be the setting value and call seteffectivepriority() in the thread it is waiting for.

- threadstate::getEffectivePriority(): It will return the effective priority of the associated thread by returning effective.
- threadstate::waitForAccess(): When a thread is added to the queue to wait, it will call this function. It will enqueue the thread to waitqueue and donate priority to the waitedthread by calling seteffectivepriority and it will set it's waitingqueue the the priority queue since it's in it.
- threadstate::acquire(): If a thread doesn't need to wait for access in the queue, it will call this function. It will set the waitedthread to the current thread. After waitedthread is not null, it will not call this function.
- priorityqueue::nextThread(): It will return one thread in the queue which has the highest priority. If the queue is empty, it will return null. Since the thread in the queue is now waiting for the return thread, the waitedthread is set to the return thread. And the return thread is no longer waiting, the waitingqueue is set to null and the effective priority of the previous thread is reset to priority.
- threadstate::resetpriority(): reset effective priority.

5.3 Pseudo-code

Algorithm 11: threadstate::setEffectivePriority()

Input: priority

if $priority > effective$ **then**
    effective = priority
    if $thread.waitingqueue.waitedthread \neq null$ **then**
        getThreadState(thread.waitingqueue.waitedthread).seteffectivepriority(priority)

Algorithm 12: threadstate::getEffectivePriority()

return *effective*

Algorithm 13: threadstate::waitForAccess(waitQueue)

waitQueue.add(this)
getThreadState(waitQueue.waitedthread).setEffectivePriority(getPriority(this))
this− >waitingqueue = waitQueue

5.4 Correctness constraints

1. The effective priority of a thread is larger than or equal to the effective priority of all the threads which is waiting for it directly or indirectly.

Algorithm 14: priorityqueue::nextThread()

```
if waitqueue.isempty() then
    getThreadState(waitedthread).resetpriority()
    waitedthread = null
    return null
else
    Find the thread A in waitqueue such that the effective priority is
    highest.(If multiple threads with the same highest priority are
    waiting, choose the one that has been waiting in the queue the
    longest) A.waitingqueue = null
    getThreadState(waitedthread).resetpriority()
    waitedthread = A
```

Algorithm 15: threadstate::acquire(waitQueue)

```
waitQueue.waitedthread = corrent thread
```

5.5 test strategy

1. fork several threads with locks, each thread holds a lock and waits the previous thread's lock, test whether the donation works.

2. fork several threads, set some of them with high priority and some of them with low priority and all waiting for some locks, test whether the threads with high priority will finish first.

6 Boat Grader

6.1 Design

Notice that if we ensure 3 properpties below, then all the people will finally get to Molokai:

1. every time, boat to Molokai is full: it will catch 1 adult or 2 children, not only one child.
2. every time, on boat back Oahu, there is only one child.
3. the boat won't stuck: there always appropriate people to put on boat for above two situations.

Brief proof is that: suppose 1 adult is 100kg and 1 child is 50kg, then by porperties 1 and 2, there is 50kg weight-down of total weight of Oahu after every round trip, since the boat never stuck, total weight of Oahu will down to 0, i.e. they will finally all get to Molokai.

Algorithm 16: threadstate::resetpriority()

```
effective = priority
```

For ensuring above properties, we assume there are at least two children. From above analysis, we find that children play a role piloting boat back Oahu, and adult never return, so we decide to send children first, which is the idea our algorithm design originated.

6.2 Variables and functions

- BoatLocation: Oahu or Molokai;(global variable)
- Location: for each thread, keep in track where the person is;(local variable)
- BoatLock: a lock for protecting action to boat;
- WaitOahu: condition variable based BoatLock, for people waiting on Oahu;
- WaitMolokai: condition variable based BoatLock, for people waiting on Molokai;
- WaitFull: condition variable based BoatLock, for to get two children on boat;
- Weight: weight of people on boat;(50kg for 1 child, 100kg for 1 adult)
- Total: number of all people;(variable for main thread)
- OnMolokai: number of people on Molokai;(variable for main thread)
- ChOahu: number of children on Oahu;(global variable for Oahu people)

6.3 Pseudo-code

Algorithm 17: Boat::begin

```
fork adult and child threads
listen OnMolokai until which equal to Total
```

6.4 Testing

test(0,2,b), (0,10,b) , (10,2,b) , (10,10,b).

Algorithm 18: Boat::ChildItinerary

```
acquire BoatLock
if Location=Oahu then
    while BoatLocation≠Oahu || Weight= 100 || ChOahu= 1 do
        WaitOahu.sleep
    waitOahu.wakeall
    if Weight= 0 then
        Weight+ = 50
        WaitFull.sleep
        bg.ChildRideToMolokai
        Location:=Molokai;
    else
        Weight+ = 50
        bg.ChildRowToMolokai
        WaitFull.wake
        ChOahu− = 2
        BoatLocation:=Molokai
        Location:=Molokai
        OnMolokai+ = 2
        Weight− = 100
        WaitMolokai.sleep
else
    Weight+ = 50
    OnMolokai− = 1
    BoatLocation:=Oahu
    Location:=Oahu
    Weight− = 50
    ChOahu+ = 1
    WaitOahu.wakeall
release BoatLock
```

Algorithm 19: Boat::AdultItinerary

```
acquire BoatLock
if Location=Oahu then
    while BoatLocation≠Oahu || Weight= 100 || ChOahu> 1 do
        WaitOahu.sleep
    Weight+ = 100
    bg.AdultRowToMolokai
    BoatLocation:=Molokai
    Location:=Molokai
    OnMolokai+ = 1
    Weight− = 100
    WaitMolokai.wakeall
    WaitMolokai.sleep
else
    WaitMolokai.sleep
release BoatLock
```