Project 1 - Initial Design Document

Chen Lijie Fan Haoqiang Bi Ke

Contents

1 Our git Repository

`https://github.com/wjmzbmr/nachos`

2 Implementation of KThread.join()

2.1 Correctness Invariants

- A thread should not join to itself and a finished thread should not join to other threads.
- The method need to be made atomic, by disabling interrupting at first, and restore it when the method returns.
- Whether being joined or not, a thread must finish executing normally.

2.2 Declaration

- In class KThread, add a member variable waiterQueue(a queue of Thread), which stores the joined threads.
- Modification in KThread.join() and Thread.finish().

2.3 Description

The pseudocodes for modifications of both methods are listed below.

```
procedure JOIN()
    Disable Interruption
    if this != currentThread and this.status != statusFinished then
        add currentThread to waiterQueue
        Let the currentThread sleeps
    end if
    Restore Interruption
end procedure
```

```
procedure FINISH()
    ...
    currentThread.status = statusFinished
    Wake up threads in waiterQueue.
    sleep()
end procedure
```

2.4 Testing strategy

We plan to make the following tests.

1. Standard Case Testing

Make a thread, joined it to another one, and check whether it running order is the same as our expectation.

2. A thread joined to many other threads

Make a thread, joined it to several other threads and check whether the result is the same as our expectation.

3. A thread be joined by many other threads

Make a thread, let it be joined by several other threads and check whether the result is the same as our expectation.

4. Corner Case Testing

Make some threads be joined to itself, and join some finished threads to other threads to see whether or not those corner cases are correctly handled.

3 Another Implementation of Condition Variable

3.1 Correctness Invariants

sleep()

- The current thread must hold the lock before the method, and get the lock again after the method.
- The operation that releases the lock and put the current thread into the waiting queue must be atomic.

wake()

- The current thread must hold the lock before the method.
- The operation that wake up a thread which called sleep() before must be atomic.

wakeAll()

- The current thread must hold the lock before the method.
- The operation that wake up all the threads which called sleep() before must be atomic.

3.2 Declaration

- In class Condition2, add a member variable waiterQueue(a queue of Thread), which stores the waiting threads.
- a method sleep(), same functionality as in the class Condition.
- a method wake(), same functionality as in the class Condition.
- a method wakAll(), same functionality as in the class Condition.

3.3 Description

Following are the pseudocodes for all the methods above.

```
procedure SLEEP()
    Lib.assertTrue(conditionLock.isHeldByCurrentThread())
    Disable Interruption
    Add currentThread to waiterQueue
    Release the lock
    let currentThread sleep
    Acquire the lock
    Restore Interruption
end procedure
```

```
procedure WAKE()
    Lib.assertTrue(conditionLock.isHeldByCurrentThread())
    Disable Interruption
    if WaiterQueue is not empty then
        Wake up and remove one thread in the waiterQueue.
    end if
    Restore Interruption
end procedure
```

```
procedure WAKEALL()
    Lib.assertTrue(conditionLock.isHeldByCurrentThread())
    Disable Interruption
    while WaiterQueue is not empty do
        Wake up and remove one thread in the waiterQueue.
    end while
    Restore Interruption
end procedure
```

3.4 Testing strategy

1. Using Condition2 to implement the producer and consumer problem

Write a simple program which use Condition2 to implement the producer and consumer problem and check whether or not the results are meeting our expectation.

4 Implementation of the Alarm

4.1 Correctness Invariants

waitUntil()

- The operation that moving the currentThread into the waiting queue and sleep it must be atomic.

timerInterrupt()

- Every threads whose waiting time is over must be waken up.
- The operation that wakes up all those threads must be atomic.

4.2 Declaration

- A new class WaitingThread, which records a thread which are waiting together with its designated waking up time. It should be comparable by its waking up time.
- A new member priority queue waiterQueue in the class Alarm, which stores all the WaitingThread according to their waking up time, so we can retrieve the thread with minimum waking up time quickly.
- Modification in timerInterrupt().
- Modification in waitUntil().

4.3 Description

Following are the pseudocodes for all the methods above.

```
procedure WAITINGTHREAD(WAKETIME,THREAD)
    return a WaitingThread object with the given wakeTime and Thread
end procedure
```

```
procedure WAITUNTIL(X)
    Disable Interruption
    wakeTime ← currentSystemTime + x
    Add WaitingThread(wakeTime,currentThread) to waiterQueue
    let the currentThread sleep.
    Restore Interruption
end procedure
```

```
procedure TIMERINTERRUPT(X)
    Disable Interruption
    while The waiterQueue is not empty do
        t ← waiterQueue.peek()
        if t.wakeTime > currentSystemTime then
            break
        end if
        Wake t up
        waiterQueue.poll()
    end while
    Restore Interruption
end procedure
```

4.4 Testing strategy

1. Specified Ordering

Make many threads(around 50), and let the i-th thread waitUntil with a corresponding with x=$2000 \cdot i$.

Then we check whether their are waken up at the same order.

2. Specified Ordering with small x

Change the x in the previous test to 200 to see what happens, note 200 is smaller than the timer's clock ticks.

3. Randomized Ordering

Make many threads(with varying multitude like 10, 100,1000) with randomized x.
Then we check whether their waken up order meets our expectation.

5 Implementation of the Communicator

5.1 Correctness Invariants

speak()

- The speaker will wait if its word are not listener by a listener.
- The operation that setting the spoken word must be atomic.
- The speaker can not setting the spoken word if it has not been taken by a listener.

listen()

- The listener will wait if there is no set word.
- The operation that taking the spoken word must be atomic.

5.2 Declaration

- A state variable temp, to temporarily store the spoken word.
- A lock mutex, which ensure the operation involving the word must be atomic.
- 4 variables, AS,AL,WS,WL, indicate the current number of active speaker, active listener, waiting speaker, waiting listener.
- 4 conditions variables with lock mutex, waitS for waiting speaker, waitL for waiting listener, waitToTake for the active listener who are waiting to take the word, waitTaken for the active speaker waiting for its word to be taken.
- Function speak(word), and procedure listen().

5.3 Description

```
procedure INITIALIZE()
    AS,AL,WS,WL ← 0.
    Initialize mutex, and all conditions with mutex.
end procedure
```

```
procedure SPEAK(WORD)
    mutex.acquire()
    while NOT (AS == 0 AND AL > 0) do
        WS+=1
        waitS.sleep()
        WS-=1
    end while
    AS+=1
    temp ← word.
    waitToTake.wake()
    waitTaken.sleep() //waiting for word to be taken
    AS-=1
    if WS > 0 then
        waitS.wake()
    end if
    mutex.release()
end procedure
```

```
procedure LISTEN()
    mutex.acquire()
    while NOT (AL == 0) do
        WL+=1;
        waitL.sleep()
        WL-=1
    end while
    AL+=1
    if WS > 0 then
        waitS.wake();
    end if
    waitToTake.sleep();
    take temp;
    waitTaken.wake();
    AL-=1;
    if WL > 0 then
        waitL.wake()
    end if
    mutex.release();
end procedure
```

5.4 Testing strategy

Randomized test

Generate some speakers and listeners on the same communicator randomly, and check whether the correctness conditions are met.

6 Implementation of the PriorityScheduler

6.1 Correctness Invariants

- For the threads waiting for the same resource, the one with higher priority get the resource first, in case of a tie, the one has been waiting for longest time get it first.

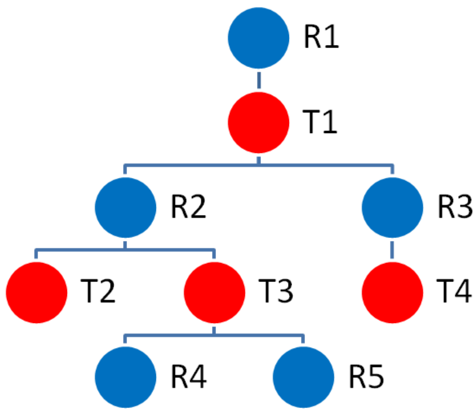
- Every methods do the scheduling must be atomic.
- If a thread is waiting for a particular resource, its priority will be donated to the thread which holds that resource.

6.2 A simple illustration

We know:

- A thread can only wait for one resource, while it may holds access to several resources.
- A resource can only be hold with one thread.

So the relation graph looks like:



And it has a tree structure, so when updating one thread's priority, we can just update the effective priority from bottom to top.

This can be done with a sophisticated structure like Link-cut tree or Splay tree to maintain the DFS order of all the nodes. But since, the tree-depth are normally not very large, I think a simple brute-force update from leaf to root is sufficient.

6.3 Declaration

6.3.1 Scheduler

- Implementation of getPriority(), setPriority() and getEffectivePriority

6.3.2 Kthread

- Notice that the original Kthread Object has a member schedulingState, which can be used to record its scheduling state.

6.3.3 PriorityThreadQueue

- Make a subclass of ThreadQueue named PriorityThreadQueue. Which is supposed to maintain the threads waiting for this resource.
- A member variable resourceHolder, which points to the thread which holds the resource.
- A member variable maxPriority, which denoting the maximum effective priority in the waiterQueue, set as minimum if waiters is empty.
- A binary search tree of SchedulingState waiters contains all the waiting threads, and compare them with the priority and the enqueue Time.
- implementation of nextThread(), acquire(), waitForAccess().

6.3.4 SchedulingState

- member variables thread, priority, enqueueTime, effectivePriority, waitingResource which corresponding to the thread it represents, the priority of that thread, the enqueueTime of that thread, the effective priority of that thread, and the resource this thread is waiting for.
- A member variable resources implemented by a binary search tree, which holds all the resources acquired by this thread.

6.4 Description

6.4.1 Scheduler

```
procedure getPriority(thread)
    return thread.schedulingState.priority
end procedure
```

```
procedure getEffectivePriority(thread)
    return thread.schedulingState.effectivePriority
end procedure
```

```
procedure setPriority(thread, p)
    if p < priorityMinimum OR p > priorityMaximum then
        return
    end if
    thread.schedulingState.setPriority(p)
end procedure
```

6.4.2 PriorityThreadQueue

```
procedure Initialize()
    resourceHolder ← null;
    waiters ← new empty TreeSet.
end procedure
```

```
procedure getMaxPriority()
    if waiters.empty() then
        return priorityMinimum
    end ifreturn waiters.first().effectivePriority
end procedure
```

```
procedure UPDATE()
    tmp ← getMaxPriority()
    if tmp != maxPriority then
        if resourceHolder then resourceHolder.updateResource(this,maxPriority)
        elsemaxPriority ← tmp
        end if
    end if
end procedure
```

```
procedure UPDATEWAITER(STATE, EP)
    waiters.remove(state)
    state.effectivePriority ← EP
    watiers.add(state)
    update()
end procedure
```

```
procedure WAITFORACCESS(THREAD)
    state ← thread.schedulingState
    state.enqueueTime ← currentTime
    state.waitingResource ← this
    waiterQueue.add(state)
    update()
end procedure
```

```
procedure ACQUIRE(THREAD)
    state ← thread.schedulingState
    resourceHolder ← state
    state.addResource(this)
end procedure
```

```
procedure NEXTTHREAD()
    if resourceHolder != null then
        resourceHolder.removeResource(this)
        resourceHolder ← null
    end if
    if waiterQueue.empty() then
        return null
    end if
    state ← waiterQueue.poll()
    thread ← state.thread
    update()
    state.waitingResource = null
    state.addResource(this); return thread
end procedure
```

6.4.3 SchedulingState

```
procedure INITIALIZE()
    priority, effectivePriority ← priorityDefault
    resources ← empty TreeSet
    waitingResource ← null
end procedure
```

```
procedure UPDATE()
    tmp ← priority
    if !resources.empty() then
        tmp ← max(tmp, resources.first().maxPriority)
    end if
    if tmp != effectivePriority then
        if waitingResource != null then
            waitingResource.updateWaiter(this,tmp)
        else
            effectivePriority ← tmp
        end if
    end if
end procedure
```

```
procedure SETPRIORITY(P)
    priority ← p
    update()
end procedure
procedure UPDATERESOURCE(RES, MAXP)
    resources.remove(res)
    res.maxPriority ← maxP
    resources.add(res)
    update()
end procedure
procedure ADDRESOURCE(RES)
    resources.add(res)
    update()
end procedure
procedure REMOVERESOURCE(RES)
    resources.remove(res)
    update()
end procedure
```

6.5 Testing strategy

Single queue

Set up a single resource, and fork many threads which waits for it, check whether they are executed in the correct order.

Many conditions

Set up a couple of condition variables, and many threads waiting on them, print the thread denoting process and the queue content, to ensure that the priority is donated correctly and the thread are executed in the correct order of the effective priority.

7 the Boat Problem

7.1 Correctness Invariants

- Every adults and children should ended up at Molokai when begin() ends.
- During the boat move, there must be at most one adults or two children on the boat, but not one adult and one children, and not empty as well.

7.2 A simple illustration of the strategy

- Note that it clearly makes no sense for an adult or two children to move from Molokai to Oahu. And if there are only one child and more than one adults at the start, the task is impossible.
- On Oahu, if there are more than two children, let two travel to Molokai. Otherwise if there are adults, let one travel to Molokai. Otherwise let the only child travel to Molokai if exists.
- On Molokai, if there are more than one child here and the schedule is not over, one child travel to Oahu.

7.3 Declaration

- A class Information, has 4 member variables adult, children, waitingChildren and hasBoat. Recording the corresponding information on one location. Which are initially zero. And two instance of it, oahu for Oahu, molokai for Molokai.
- A Communicator communicator, which are previously implemented and used to send one-way message from threads to the begin()
- A Lock boatMutex, which ensure that the mutual access to the boat and the information. And three conditions waitMolokai, waitOahu and waitToGo, for the people waiting on Molokai, the people waiting on Oahu, and the first people waiting for the travel partner on Oahu.
- Modification in begin(), AdultItinerary() and ChildItinerary().
- Constant integers ADULT=0 and CHILD=1.

7.4 Description

```
procedure BEGIN(ADULTS, CHILDREN)
    oahu.hasBoat ← true
    Initialize boatMutex, and conditions by boatMutex.
    Initialize communicator
    Create adults Adult threads.
    Create children Children threads.
    while communicator.listen() != adults + children do
    end while
end procedure
procedure CAN(TYPE, SIDE)
    if !side.hasBoat then
        return false
    end if
    if type == ADULT then
        return side.waitingChildren == 0
    else
        return side.waitingChildren <= 1
    end if
end procedure
procedure REPORT()
    communicator.speaker(molokai.adult + molokai.children)
end procedure
```

```
procedure TRAVEL(TYPE, ROW, RIDE, FROM, TO)
    //row = True, it pilots, otherwise it is a passenger.
    base on type, row and to, call the specific method of bg.(like ChildRowToMolokai()).
    if type == ADULT then
        from.adult -= 1
        to.adult +=1
    else
        from.children -= 1
        if from == oahu then
            from.waitingChildren -= 1
        end if
        to.children +=1
    end if
    if row then
        from.hasBoat=false
    end if
    //this will enforces that between the pilot's arrives and ride's arrive, the boat is at nowhere.
    if ride then
        to.hasBoat=true
        if to == molokai then
            report()
            waitMolokai.wakeAll()
        else
            waitOahu.wakeAll()
        end if
    end if
end procedure
```

```
procedure ADULTITINERARY()
    //Come to live
    oahu.adult += 1
    boatMutex.acquire()
    while !can(ADULT,oahu) OR oahu.children >= 2 do
        if oahu.hasBoat then waitOahu.wakeAll()
        end if
        waitOahu.sleep()
    end while
    travel(ADULT,true,true,oahu,molokai)
    boatMutex.release()
end procedure
```

```
procedure ChildItinerary()
    //Come to live
    oahu.children += 1
    where ← oahu
    while  do
        boatMutex.acquire()
        if where == oahu then
            while !can(CHILD,oahu) OR (oahu.adult > 0 AND oahu.child = 1) do
                if oahu.hasBoat then waitOahu.wakeAll()
                end if
                waitOahu.sleep()
            end while
            oahu.waitingChildren +=1
            if oahu.child >= 2 then
                if oahu.waitingChildren == 1 then
                    waitToGo.sleep()
                    travel(CHILD,false,true,oahu,molokai)
                else
                    waitToGo.wake()
                    travel(CHILD,true,false,oahu,molokai)
                end if
            else
                travel(CHILD,true,true,oahu,molokai)
            end if
        else
            while !can(CHILD,molokai) do
                if molokai.hasBoaat then
                    waitMolokai.wakeAll()
                end if
                waitMolokai.sleep()
            end while
            travel(CHILD,true,true,molokai,oahu)
        end if
        boatMutex.release()
    end while
end procedure
```

7.5 Testing strategy

Randomized General Test

Start with 3 to 5 adults and children and output the resulting plan and check the output manually. To increase the confidence, we can check it many times.

A special Test

One concern is that a person may appear while somebody else are viewing the information, this might cause trouble, so we fork many children(around 20), and to see the results is whether what we expected or not.