Project 1 - Initial Design Document

Chen Lijie	Fan Haoqiang	Bi Ke
2013011313	2011012357	2011012360

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