Design Document for Nachos
Phase 1: Building a Thread System

Abdulaziz Awliyaa, Christopher Connors, Brian Giroux
COSC 3407 - Operating Systems
Group 1

February 11, 2017

Task I: Implement `KThread` Joining

In `KThread`, we implement the `join()` method. When called, this method will cause the the calling thread to wait on another thread before continuing.

Implementing the `join()` Method

The `join()` method requests access to the CPU, it checks to make sure nothing is running; if it is, it will wait on the queue.

If `join()` is called a second time, the behaviour is undefined

See *Algorithm* 1.

Algorithm 1: The `join()` method

```
if this thread is the current thread then
    abandon;
end
disable interrupts;
if this thread is running then
    put this thread on the wait queue;
    put this thread to sleep;
end
enable interrupts;
```

Test Cases for Task I

1. We will create a thread
2. Start a second thread
3. We will try to join one of the threads to itself to make sure it can't join itself
4. We will join one thread to the other thread
5. We will verify that the thread waits for the first thread

Task II: Implementing the `Condition2` Class

In the `Condition2` class, we implement three methods: `sleep()`, `wake()` and `wakeAll()`.

Implementing the `sleep()` Method

First we have to make sure that the current thread holds the lock. Then we atomically add the thread to the queue and put it to sleep.

We must release the lock before putting the thread to sleep otherwise no other thread can acquire the lock.

See *Algorithm* 2.

Algorithm 2: The `sleep()` method

```
if the current thread does not hold the lock then
    abandon;
end
disable interrupts;
add the current thread to the wait queue;
release the lock;
put the thread to sleep;
acquire the lock;
enable the interrupts;
```

Implementing the `wake()` Method

We check to make sure that the current thread has the lock. Then, we atomically check the wait queue and if there is a thread on the wait queue, wake it up.

See *Algorithm* 3 on the following page.

Implementing the `wakeAll()` Method

We simply call `wake()` on the first thread on the wait queue until the wait queue is empty.

See *Algorithm* 4 on the next page.

Algorithm 3: The `wake()` method

```
if the current thread does not hold the lock then
    abandon;
end
disable interrupt;
if the first item on the readyQueue is a KThread then
    call KThread.ready();
end
enable interrupts;
```

Algorithm 4: The `wakeAll()` method

```
disable interrupts;
if readyQueue is not empty then
    foreach KThread in the readyQueue do
        wake the KThread;
    end
end
enable interrupts
```

Test Cases for Task II

1. We will create some threads (5 or more) putting them to sleep as we go along
2. We will call `wake()` to test that one thread wakes up
3. We will call `wakeAll()` to make sure that the rest of the threads wake up.

Task III: Implementing `Alarm`

In this section we implement the `timerInterrupt()` and `waitUntil()` methods.

Implementing the `timerInterrupt()` Method

The `timerInterrupt()` method will wake the first item on the wait queue if its time is up. The wait queue will be a priority queue and the items on the wait queue will contain a thread and a wake time.

See *Algorithm* 5.

Algorithm 5: The `timerInterrupt()` method

```
disable interrupts;
if the readyQueue is not empty then
    check the wake time for the thread at the head of the queue;
    if it’s time for him to wake up the wake him up;
end
enable interrupts;
yield to the next thread;
```

Implementing the `waitUntil()` Method

This method will calculate the threads wake time, then put the thread along with its wake time on the waitQueue.

See *Algorithm* 6.

Algorithm 6: The `waitUntil()` method

```
input      : the amout of time ($x$) to wait
disable interrupts;
set wake time to current time + $x$;
put the thread and wake time onto the waitQueue (priority queue);
go to sleep;
enable interrupts;
```

Test Cases for Task III

We will make some threads and put them to sleep for various amounts of time. We will watch as they are removed from the wait queue.

Task IV: Implementing `Communicator`

The `Communicator` class allows threads to communicate. We will implement two methods: the `speak()` and the `listen()` methods.

The `speak()` Method

Speakers wait until there are no listeners or other speakers before writing to the variable. Then, they wake up a waiting listener.

See *Algorithm* 7.

Algorithm 7: The `speak()` method

```
input      : an integer value to speak
acquire lock;
if the listen queue is empty then
    create a new speaker in the speaker queue;
    set its word to the spoken word;
    put the speaker to sleep;
else
    get the first listener off the listen queue;
    set the listener word to the spoken word;
    wake up the listener;
end
release lock;
```

The `listen()` Method

Listeners wait until there are no speakers to read the variable. Then, they wake up a waiting speaker.

See *Algorithm* 8 on the next page.

Test Cases for Task IV

1. Speak some words (4 or 5)

2. Call some listeners (2 or 3)

Algorithm 8: the `listen()` method

```
output      : the integer word spoken by the speaker
acquire lock;
if the speaker queue is empty then
    create a new listener in the listener queue;
    put the listener to sleep;
    get the sleeper's word;
else
    remove a speaker from the queue;
    get the speaker's word;
    wake up the speaker;
end
release the lock;
return the word;
```

3. Make sure that each listener heard what was spoken

4. Check the speaker queue to make sure that they've been removed

Task V: Implementing `ReactWater`

We will keep two global variables: H to count the number of hydrogen atoms present and O to count the number of oxygen atoms. The `hReady()` and `oReady()` methods will increment H and O respectively and the `makeWater()` method decrement the variables by the appropriate amounts and display a message.

The `ReactWater` Constructor

The constructor will simply initialize the H and O variables to 0.

See *Algorithm* 9.

Algorithm 9: the `ReactWater` constructor

```
set $H$ and $O$ to 0;
```

The `hReady()` Method

`hReady()` will increment *H* by 1 and call `makeWater()`.

See *Algorithm* 10.

Algorithm 10: the `hReady()` method

```
increment $H$ by 1;
call make water;
```

The `oReady()` Method

`oReady()` will increment O by 1 and call `makeWater()`.

See *Algorithm* 11.

Algorithm 11: the `oReady()` method

```
increment $O$ by 1;
call make water;
```

The `makeWater()` Method

`makeWater()` will check how many of *H* and *O* we have. If we have sufficient quantities, we will decrement *H* by 2 and *O* by 1 and print a message.

See *Algorithm* 12.

Algorithm 12: the `makeWater()` method

```
if $H \geq 2$ and $O \geq 1$ then
    decrement $H$ by 2;
    decrement $O$ by 1;
    print the ”I made water” message;
end
```