Design Document for Nachos

Phase 1: Building a Thread System

Justyn Bélanger, Tyler Jessup,

Clément Mindjimba, Alain Levasseur

COSC 3407 - Operating Systems

GROUP 04

February 9th, 2018

# 

# Task I: Modifying the KThread Class

The KThread class is a special instance of threads that allows the execution of kernel code. An important part of this class yet to be implemented is the ability to schedule threads that are dependant on others to finish before they can continue. This is done through the join() method.

## Implementing the join() method

This method joins the currently running thread to another KThread and forces it to wait until the called thread finishes. Provided that the called thread is not the current thread. Machine interrupts must be disabled in order to ensure atomicity of the method. In order to avoid any deadlocks caused by a thread waiting on a sleeping thread that will never awake a new method is implemented to check if the called thread is on the current thread's waitArray. Bellow is the pseudo code of this method:

join(){

assert this thread is not the current thread;

assert that this thread is not finished

disable machine interrupts

if(currentThread.isWaiting(this)){

error message: Cannot join to a thread in the waitArray

}

else{

add currentThread to the next spot in the waitArray.

call sleep();

}

re-enable machine interrupts

}

## Implementing the isWaiting() method

The simple isWaiting() method loops through the waitArray and returns whether or not the specified thread is found within.

isWaiting(KThread t){

for each thread in this.waitArray{

if(thread == t){ return True; }

}

return False;

}

## Modifying the finish() method

The finish() method must be modified to awake the sleeping threads inside the waitArray once the thread is finished. This modification must be completed after machine interrupts are disabled to ensure atomicity much like with the join() method. It must also be placed before the final sleep() is called.

finish()

disable machine interrupts

//Existing code

for all i in this.waitArray{

i.ready();

}

//Existing code

sleep();

}

## Test cases for task I

1. A thread may not join itself.

2. A joined thread is put to sleep.

3. Multiple threads may be joined to a single thread.

4. A thread may not join a thread if it is currently on its waitArray.

testCases(){

//Case 1:

Create a new KThread t1

Join t1 to itself

Expect error text

//Case 2:

Create a second Kthread t2

Join t2 to t1

//Case 3:

Create a third and fourth Kthread. t3 and t4

Join t3 and t4 to t1

//Case 4:

Join t1 to t2

Expect error text

}

# Task II: Implementing the Condition2 Class

The Condition2 class is a condition variable. It is a high-level synchronization primitive that enables a thread to go to sleep in the critical section and once that the condition is met the thread can be woken up. This can be achieved by using the locks to achieve mutual exclusion to unlock the thread and go to sleep and then reacquiring the lock. It also uses a queue to keep all the sleeping thread together to be woken later.

## Implementing the sleep() method

To implement this method, it is important to assert that the lock is being held by the current lock. After this, the interrupts must be disabled to ensure that the nothing can interrupt the process. Then add the thread to the waiting queue release the lock and put the thread to sleep atomically. After this the interrupts must be re enabled to keep the system running properly.

void sleep() {

Assert that the lock is held by the current thread

Disable interrupts

Add the thread to the wait queue

Release the lock

Put thread to sleep

Reacquire the lock

Re Enable the interrupts

}

## Implementing the wake() method

The wake method simply wakes up a thread on the wait queue. First it must be ascertained that the current thread holds the lock of the condition variable. After this the interrupts must be disabled before the thread is woken up. After removing the thread of the waiting queue, the thread is put into the ready state to wait for the processor. After this the interrupts must be re enabled.

void wake() {

Assert that the lock is held by the current thread

Disable the interrupts

if the queue is not empty{

Remove the thread from the queue

Ready the thread

}

Re Enable the interrupts

}

## Implementing the wakeAll() method

For this method, the implementation is simple, simply wake all the threads on the queue until the queue is empty.

void wakeAll() {

Assert that the lock is held by the current thread

while the queue is not empty {

wake the thread

}

}

## Test Cases for Task II

Test Case 1 - Putting a thread to sleep in the critical section: This is a simple test just puts the current thread to sleep.

void test1(){

Print out a message saying the thread is about to sleep

Acquire the lock

Put current thread to sleep

Release the lock

Print out message saying the thread has awoken

}

Test Case 2 - Putting a second thread to sleep: This test is to test the implementation of the queue and make sure that more than one thread can be put to sleep.

void test2(){

Print out a message saying the thread2 is about to sleep

Acquire the lock

Put current thread to sleep

Release the lock

Print out message saying thread2 has awoken

}

Test Case 3 - Wake one thread that was put asleep earlier: This test is to ensure that a thread can be woken up from the wait queue.

void test3(){

Acquire the lock

Wake a thread in the queue

Release the lock

}

Test Case 4 - Wake all the threads in the queue: This test is to ensure that the wake all method works properly.

void test4(){

Acquire the lock

Call wake all thread

Release the lock

}

Test Case 5 - Waking and empty queue: This test is to make sure that trying to wake up an empty queue using the wake method does not result in an error.

void test5(){

Acquire the lock

Try to wake a thread

Release the lock

}

Test Case 6 - Using wakeAll to wake an empty queue: This test is to make sure that the wakeAll method does not result in an error when the queue is empty.

void test1(){

Acquire the lock

Tre to wake all thread

Release the lock

}

These test even tho they look similar must be performed in this order to maintain the validity of the test and the correctness of the output generated by the self test of the condition variable.

# Task III: Implementing the Alarm Class

## Section 1.0: General High-Level Implementation Methodology:

The implementation of Task III provides the functionality to meet the API requirements of the Alarm class to function as a pre-emptive timing interrupt handler for forceful pre-emptive external interrupts; the functionality of this mechanism within the system (or object within the object model) is made possible through code base modifications of the Alarm class’s constructor, the timerInterupt method, the waitUnitil() method as well any associated classes used to aid in the general implementation. This implementation will be described in full separated between 3 primary components: 1. A behaviour model which describes how the alarm class model acts between changes in state (a refined model to the previously implemented busy-waiting algorithm). 2. The behaviour of model 1. depicted as a flow chart between existing nachos objects and new objects developed used in the implementation of 1. 3. The object model of the code base modifications as class diagrams and relational dependencies for the high-level architecture used to implement the specified functionality required by the Alarm class’s API. The design documented here is able to change upon Course reviews and developmental difficulties or design shortcomings found in refactoring.

## *Section 2.0: Behaviour Model: State Vs. Busy-Waiting:*

The model for the functionality of the Alarm API requirements is based on a generic term of “state”. What is meant by “state” is that the timing interrupts are no longer being implemented by a tolling algorithm such as busy waiting, but the interrupts are managed by state changes between switching threads; the “state” here means that the threads are pre-empted or switched between by each thread having an active knowledge of the other threads state. In general, when a timing interrupt occurs, the current thread is requested to yield, and made second (emphasized in the paragraph bellows with via Semaphore) to the interrupting thread. When the interrupting thread has completed (in whatever manner) the old current thread is then again ready for execution. The state being realized in this model is the state that 1. The interrupting thread is allowed to make the interrupt. 2. The interrupting thread is acknowledged by the current thread to yield and 3. The yielded thread is then acknowledged by the interrupting thread to resume. This communication is governed by the “state” in the model.

This “state” model is implemented through the development of 4 primary components: 1,2. Two separate synchronization primitives (Lock and Semaphore), 3. A data structure (or abstract data type) used to implement a thread forcing an interrupt (to hold the state of the wake time, as well the state of communication between the interrupting thread and the yielding thread) and finally 4. A priority Queue to hold all pending interrupting threads (ordered and compared by wait times). The flow chart in Section 3 presents a high-level view of these interacting components and the object model described in Section 4.0 shows the pseudo-functionality of the code base modifications.

The constructor of Alarm will be used to redirect the Alarm class to a wrapper around Alarm that will be used to manage the complexity of the functionality of the class. Once redirected, the code base modifications will be architectured into an object model described in Section 4.0. This object model will handle setup and configuration, and also the complexities for the functionality of the timerInterupt() method as well the waitUntil() method, and these methods core implementation are described next.

The timerInterrupt() method functionality accomplishes 6 objectives: 1. Identifies all threads pending and waiting on interrupts. 2. Makes active all pending threads for interrupts with wait times greater than system time. 3. Gives state to the interrupting thread to make the interrupt (as acquiring a lock for its critical section/actions). 4. Releases the state of the lock to all other requesting threads for interrupts. 5. Establishes communication (or state) with the current thread via semaphore (Semaphore.V()) for the thread to sleep and 6. Request the current thread to yield to the interrupting thread.

The waitUntil () method functionality accomplishes 6 objectives: 1. Determines the new time that this interrupting thread will have for its interrupt (system time + x units). 2. Create a new data structure/abstract data type “Type X” of an interrupting thread (manages the state of waiting time and communication with the yielding thread via Semaphore). 3. Gain state for the new interrupting thread to be added to the wait queue (as acquiring a lock for its critical section/actions). 4. Releases the state of the lock to all other threads. 5. Establish communication (or state) with the current thread so it can be made ready again once the interrupting thread has completed via semaphore (Semaphore.P())

Finally, two core objects will be developed: 1. An abstract data type as an interrupting thread (stores the state of the wake time as well communication (or state) with the current thread via semaphore) and 2. A data structure as a priority queue to manage all waiting interrupting threads (the queue will be ordered upon comparing(implementing) wait times).

Any and all functionality complexities from the above implementation will be refactored through the object model introduced in Section 4.0.

## Section 3.0: Behaviour “State” Model Flow Chart

## Section 4.0: Object Model Classes/Methods/Dependencies/Pseudo-Implementation:

Class Alarm\_Manager:

//a wrapper class that redirects control from Alarm to manage the complexities of //Alarm’s functionality

//a generic high-level controller for the object model used for this implementation

//manages:

//Set\_Up\_Configuration

//Lock\_Manager

//Semaphore\_Manager

//Pre\_Emptive\_Thread

//PriortyQueue<Pre\_Emptive\_Thread>

//relational dependencies will be determined through architecture refactoring

Class Set\_Up\_Configuration:

//manages the object model set up and configuration

//initializes the state of the object model

Class Lock\_Manager:

//manages all behavioural complexities of the integration of locks within the behaviour //“state” model

//methods:

//Acquire\_Lock():

//complexities of acquiring a lock

//Release\_Lock():

//complexities of releasing a lock

//Critical\_Section():

//complexities of the critical section

Class Semaphore\_Manager:

//manages all behavioural complexities of the integration of semaphores within the //behavioural “state” model

//methods:

//Consume():

//complexities of .P()

//Produce():

//complexities of .V()

Class Pre\_Emptive\_Thread:

//manages the complexities of an interrupting thread

//stores the state of the wake time

//stores the state of communication with the current thread to be interrupted (via semaphore)

Class PriorityQueue<Pre\_Emptive\_Thread>:

//manages the complexities of all waiting interrupting threads

//ordered from highest to lowest of wait time priorities

//implements comparable upon highest priority wait time

## Section 5.0: Test Cases for Task III

*Test Case 1 - Highest Priority Wait Time:* The priority queue of pending interrupting threads always has a front with the highest priority wait time.

*Test Case 2 - Ordered Wait Time:* The priority queue of pending interrupting threads is compared and ordered from highest priority to lowest.

*Test Case 3 - Priority Wait Times are Scheduled:* The priority queue of pending interrupting threads makes active all interrupting threads with an exhausted wait time w.r.t system time.

*Test Case 4 - Lock Acquired (State Check):* Upon a timer interrupt, the critical section is given the lock to enter its critical section.

*Test Case 5 – Critical Section Completes:* Upon a timer interrupt, the critical section completes its specification.

*Test Case 6 – Lock Released (State Check):* Upon a timer interrupt, the lock is released from the current interrupting thread.

*Test Case 7 – Thread Communication (State Check):* Upon a timer interrupt, the interrupting thread establishes communication with the current thread (via Semaphore).

*Test Case 8 - Current Thread Yields:* The current thread yields to the interrupting thread.

*Test Case 9 – Lock Acquired (State Check):* Upon an interrupt request, the critical section is given the lock to enter its critical section.

*Test Case 10 – Critical Section Completes:* Upon an interrupt request, the critical section completes its specification.

*Test Case 11 – Lock Released (State Check):* Upon an interrupt request, the lock is released from the current interrupting thread.

*Test Case 12 – Thread Communication (State Check):* Upon an interrupt request, the interrupting thread establishes communication with the current thread (via Semaphore).

# 

# Task IV: Implementing the Communicator Class

The communicator allows threads to exchange messages between them. There can be multiple threads that wait to listen and speak, but there can never be a listener and a speaker listening at the same time. When one waits ("wakes"), the other one "sleeps".

## Implementing the speak() method

The speak() method will wait for a thread to listen through this communicator using a while loop: as long as no thread is listening and that there are no messages, the speaker sleeps. So while the speakers sleeps the method can create a message and wake the listener to listen to a new message.

public void speak(int word){

communicator.acquire()

while(no thread is listening || no messages){

speaker.sleep()

}

message = word //Create a new message.

listener.wake()

communicator.release()

}

## Implementing the listen() method

The *listen()* method first creates a listener that will listen to a new message, and using a while loop, as long as there is no messages the speaker is awake and the listener sleeps. Once the message is heard the listener has accomplished its "mission" so it can be removed using *listener--*. After the message has been heard (passed to *speak()*) the method returns it.

public int listen(){

communicator.acquire()

listener++

while(no messages){

speaker.wake()

listener.sleep()

}

int word = this.word

listener--

communicator.release()

return word

}

## Test Cases for Task IV

1) Both listener and speaker are waiting:

If there are multiple messages to "speak" and "listen":

public void run(){

for(some boundaries){

communicator.speak();

communicator.listen();

}

}

This can be fixed by simply removing either communicator.speak() or communicator.listen().

2) In speak(), more than one listener receive word.

public void run(){

for(some boundaries){

communicator.speak();

communicator.speak();

}

}

This can be fixed by using one communicator that receives word instead of two.

3) listen() doesn't return the integer transferred.

This could happen didn't "hear" the message

public void run(){

for(some boundaries){

int messageHeard = 0;

}

}

This can be fixed by changing the value of messageHeard of 0 to communicator.listen();

# Task V: Implementing the ReactWater Class

## Implementing the ReactWater() constructor

The constructor simply initializes the lock and variables used within the ReactWater class.

Public ReactWater(){

Create a lock

hElements -> 0;

oElements -> 0;

Create hWait and oWait condition2

}

## Implementing the hready() method

The method adds elements coming in to a queue. So the count is increased by 1 with *hElements++* each time a new element comes in, it is then added to the queue and the method *Makewater()* is called.

public void hReady(){

Disable machine interrupts

hElements++

hWait.sleep()(add to the queue)

if (hElements >= 2){

Makewater() // Call the method MakeWater.

}

Re-enable machine interrupts

}

## Implementing the oReady() method

The implementation of oReady() is heavily dependent on the architecture and implementation of the overall class. When oReady() has been called it is made obvious that the state of ReactWater has an oxygen atom available/created or that oxygen(at least 1) exists. When the state of ReactWater enters this state it is possible for water to be created; water can be created at this point because water = (H x 2) + (O x 1). Because react water has reached the state where it is possible to create water (because of the existence of at least 1 oxygen), it must check if H is ready (hReady()). In checking hReady(), hReady() is only ready when the H count is >= 2. If hReady() is ready, then make water is possible and a timer interrupt should be made for ReactWater to execute the making of water, and then once this is complete returns recursively to the state of ReactWater for the condition of making water, and the actual making of water.

oReady()

{

//from the implementation of ReactWater, the state of ReactWater has realized that there exists at least 1 oxygen in the system :: oxygen count >= 1

//since oReady() has met its condition, it must check if hReady() has made its identical condition

//if both conditions are met, make water should be called a timer interrupt for ReactWater should interrupt the system so water can be made

//water should be made

//oReady() should interact with the entire state of the ReactWater class from its implementation specifics

//the state of react water should be recursively reinitialized to testing the condition for the make water condition, and if water should be made, and forced back to the defining state of ReactWater

}

## Implementing the makeWater() method

For this method, it must as always be asserted that the lock is held by the current thread. After, remove two hydrogens and one hydrogen from the waiting queue and decrement the two counters in the code. After this is done it can be said that the water was been made.

public void makeWater {

while there are more than 2 hydrogens and 1 oxygen {

Remove first oxygen

Remove first and second hydrogen

Decrement hydrogen count by 2

Decrement oxygen count by 1

Print out Water has been made

}

}

## Test Cases for Task V

Test Case 1: If there is an abundance of hydrogen water must be formed when there is a new oxygen added.

Test Case 2: If there is an abundance of oxygen, water must be formed when there are two hydrogens added.

Test Case 3: makeWater() must not create water if there is a lack of oxygen or hydrogen in the system.

Test Case 4: Upon the implementation with the use of locks, semaphores and condition variables, test boundary cases on the synchronization of these primitives.