CSE 150 Operating Systems

Group 2 03L

Project 1
Initial Design Document

Jason Feng, Arturo Ortiz, Cristian Ramirez, Mohammad Akbor Sharif, Nikko Solon

March 2nd, 2019

Introduction

Nachos thread system implements thread fork, thread completion, and semaphores for synchronization. The skeleton of the Nachos operating system is incomplete as we are required to implement certain unfinished functions into the code. Our groups goal for this project is to synchronize code for the scheduler to run without any synchronization problems.

The first phase of this design project is to build a thread system for the Nacho operating system.

Tasks:

- Implement KThread.join() in KThread Class
- Implement the condition variables directly in Condition2 Class
- Implement the Alarm Class by implementing the waitUntil(long x) method
- Implement synchronous send and receive messages using condition variables in Communicator Class
- Implement priority scheduling in PriorityScheduler Class

The second phase of this design project is to arrange a solution to transfer everyone from Oahu to Molokai using the thread system we built.

Implementing KThread.join()

Join method can only be called once. Join method's second call on the same thread is not defined even if the second call is on a different thread. If there is join called on a thread second time, it will still have a normal execution as the first call. The called thread can not be the current thread.

For example, If we have two threads named A and B. And we call join on A, A.join(), then B will be in the waiting queue and sleeping and will not be woken up until A completes its threads. We need to awake B when A is completed. In order to awake B, we need to modify the finish method in the Kthread.java.

KThread.join()

```
Join(){
     if(thread != finished && currentthread != thread_with_join && check that
join was not called previously){
     put the current thread to the waiting queue;
      currentthread.sleep();
     }
     else {
     finish() and return
     }
}
```

Testing Strategy

1. Create two thread, KThread t1 = new KThread(new PingTest(0)) and KThread t2 = new KThread(new PingTest(1)) in the selfTest method.

2. Then we call fork() on both threads and call join() on the first thread in between.

```
t1.fork()
t1.join()
t2.fork()
```

What we will check is that if t1 completely executes itself before any other threads is executed in the process.

3. We also will call join() twice on the first thread after calling the fork().

```
t1.fork()
t1.join()
t1.join()
t2.fork()
```

Here we check if calling join() twice does not change executing t1 normally as if it was called only once in the process.

4. We also will call join() on the second thread after calling it on the first thread.

```
t1.fork()
t1.join()
t2.join()
t2.fork()
```

Here we check if calling join() on the t2 after t1 does not affect the execution of t1 normally.

Implementing condition variables directly

Another way to implement condition variables directly besides using semaphores is to use mutex locks. We can acquire the same synchronization functionality that semaphores have, the functions P() and V(), but with mutex locks using the functions acquire() and release().

To implement this, we declare a private linked list of Locks called “waitQueue,” which will be used in different methods inside the Condition2 class. In the Condition2 constructor, we initialize waitQueue as a new linked list of Locks. In the sleep() method, we declare and initialize a new Lock called “waiter.” We then add waiter to waitQueue and call the function acquire() after conditionLock calls the function release(). In the wait() method, we first check that waitQueue is not empty. If it is empty, then we exit the method without doing anything. If it is not empty, then we remove the first element in waitQueue and have it call the function release(). In the wakeAll() method, we keep calling the method wake() on each element of waitQueue until waitQueue is empty.

Testing Strategy

For testing, we can have the java classes use the Condition2 class instead of the Condition class. We would then run these classes using the Condition2 class to make sure that it functions in the same way as the Condition class would.

Condition2 Class

```
private LinkedList<Lock> waitQueue
sleep(){
      Check that the current thread holds conditionLock
      waiter = new Lock
      Add waiter to waitQueue
      conditionLock.release()
      waiter.acquire()
      conditionLock.acquire()
}
wake(){
      Check that the current thread holds conditionLock
      if(waitQueue is not empty)
            waitQueue.removeFirst().release()
}
wakeAll(){
      Check that the current thread holds conditionLock
      while(waitQueue is not empty)
            wake()
}
```

Completing the implementation of the Alarm class.

When a thread calls the function waitUntil(long x), a variable called “wakeTime” is declared and initialized to the sum of the current time, by calling Machine.timer().getTime(), and x. We then create a private TreeSet of KThreads called “waitQueue” and insert this thread into waitQueue, which sorts the threads by increasing wakeTime. We have this thread call the function sleep() to make it sleep.

Now the thread is sleeping until it is awoken, and it will be awoken once the current time has passed wakeTime. This means that we will have to periodically check to see if the current time is greater than wakeTime and wake it up. This will be done within the method timerInterrupt(), since this method is called periodically by the machine’s timer. When this method is called, we first check to see if waitQueue is not empty. If it is empty, then we do nothing and return from the method. If it is not empty, then we get the current time and store it into a temporary variable called "currentTime." Then, starting from the head of waitQueue, we check if currentTime is greater than or equal to the thread's wakeTime, in which we have it call the function ready() to have it placed back into the ready queue and remove it from waitQueue. We keep doing this until currentTime is less than a thread's wakeTime. Since waitQueue is sorted by increasing wakeTime, we know that the rest of the threads' wake Times will also be greater than currentTime, so we don't have to check them and can safely return from the method.

Alarm Class

```
private TreeSet<KThreads> waitQueue
timerInterrupt(){
      if(waitQueue is not empty){
            currentTime = get the current time
            while(currentTime >= waitQueue.first().wakeTime)
                  waitQueue.pollFirst.ready()
      }
      KThread.currentThread().yield()
}
waitUntil(long x){
      wakeTime = x + current time
      Insert KThread into waitQueue
      sleep()
}
```

Implementing synchronous send and receive of one word messages.

To start sending and receiving one word messages, we first allocate memory space for a new communicator. Under the communicator() method, we will make a new Lock called the mutex then create two new condition variables with the mutex called speakRead and listenReady. Under the speak() method, we want to start the method by acquiring the mutex and ending it by releasing the mutex. Between acquiring and releasing we want the listenReady variable to sleep while the listen variable is equal to zero. After the while loop, the message variable is set to one word then wake up the speakReady variable . Under the listen() method, the mutex is acquired and released and returns the messenger. After the mutex is acquired listenReady is woken up and while the speaker variable is equal to zero, the speakerReady variable is put to sleep.

Communicator Class

```
communicator(){
      mutex = new Lock()
      speakReady = new condition(mutex)
      listenReady = new condition(mutex)
}
speak(int word){
      acquire mutex
      while(listen == 0){
            listenReady.sleep()
}
messenger = word
wake Up speakReady
      release mutex
}
listen(){
      acquire mutex
      listenReady.wake()
while(speak == 0){
            speakReady.sleep()
}
release mutex
return messenger
}
```

Implementing priority scheduling in Nachos.

One solution to this is to implement a queue to handle different levels of priority. Where threads with higher priority will get CPU time first and afterwards, the thread will yield the cpu and donate their own priority to the lower priority threads. This will subsequently raise the given's threads effective priority and lower the original thread's own effective priority. The priority scheduler works when an interrupt or yield happens and the os will need to calculate the next thread to run. nextthread() is called which will in itself call picknextthread() if the thread received is present then we would remove it from the waitqueue and return the thread, else if we did not get a thread from picknextthread() we would return null. Picknextthread() would call geteffectivepriority() which will take in account of priority donation and ensure that priority inversion would not take place. The design for geteffectivepriority would be that in a waitqueue/donationqueue, if a lower priority thread is holding the lock and there is a higher priority thread waiting.

Priority Scheduler Class

```
priorityScheduler(){

}
KThread nextThread() {
      Threadstate thread = this.picknextthread();
      If (thread != null){
this.acquire(thread)    // we get the resource wanted from the queue
this.priorityqueue.remove(thread);      //we implement priority queue to
//choose thread with highest effective priority and remove it from queue
```

```
            Return thread;
      }
      else
{return null}
}
picknextthread(){
If(waitqueue == empty)
      Return null;

Else {
      Threadstate best thread;
      Sift through priority queue
      waitqueue.thread.Geteffectivepriority();
      Best thread = highest effective priority


      Return best thread to get CPU time
}

acquire(){
      If (lock is free){
Current thread becomes the lock holder
}
Else {
thread.waitforaccess(waitingqueue);
thread.sleep;
}
}
waitForAccess(){
      If (another thread has lock)
 then add current thread to waitqueue
}
geteffectivepriority(){

Effective priority = this.priority
```

```
Return effective priority;
}
```

Arranging a solution to transfer everyone from Oahu to Molokai.

One solution to transfer all adults and children from Oahu to Molokai is to first transfer all the children from Oahu to Molokai. We will assign one child to act as the pilot and load one additional child on the boat to Molokai, reserving the condition that only two children can be on the boat at the same time. With the same pilot bring back the boat to Oahu, we will repeat this step until there are no more children on Oahu. Then when all the children are on Molokai, we can make one child pilot bring back the boat to Oahu to start loading an adult. When the adult is on Molokai, a child can bring back the boat to Oahu and the process repeats until all the adults are on Molokai. When there is one child on Molokai and more adults on Oahu, we will repeat the first process of bringing all the children to Oahu. Then once all adults are on Molokai, we can bring back the children back to Molokai.

Possible cases: (Assuming at least two children)

Odd Children & Adults	**Even Children & Adults**	**Odd Children & Even Adults**	**Even Children & Odd Adults**
Children > Adults	Children > Adults	Children > Adults	Children > Adults
Children < Adults	Children < Adults	Children < Adults	Children < Adults
Children = Adults	Children = Adults		

Task 6 Solution

```
Solution(){
      Bring 2Child to Molokai
      while(Everyone is not on Molokai){
           while(Child on Oahu){
                Bring 1Child to Oahu
                Bring 2Child to Molokai
           }
           while(Child on Molokai > 1){
                Bring 1Child to Oahu
                Bring 1Adult to Molokai
           }
      }
}
```

Conclusion: