**Task 1 - Join**

**New Variables**

* Kthread class need a waitQueue to store the currrenThread, which will be paused because other threads call join() method. So, we add a private member waitQueue in Kthread class.

*Private ThreadQueue waitQueue*

**Implementation Details**

* Join() method

Firstly, we should use Lib.assertTrue(this != currentThread) to check the caller of join() whether is the current thread. Then, we should check the status of the current thread. If it already finished, then we just return and do nothing. If not, we use machine.interrupt.disable() to block the function interrupt and put the current thread in the waitQueue to sleep until the caller thread finish. Lastly, resume the interrupt function.

*public void join() {*

*check the caller of join() is not the Current thread*

*if status is not statusFinished {*

*block the function interrupt*

*add currentThread to a waitqueue;*

*cause the current thread sleeps;*

*}*

*resume the interrupt function;*

*}*

* Finish() method

Firstly, it blocks the interrupt function and then we set the status of current thread to statusFinished. Then, we use while loop to waken all thread in the waitQueue and put them in the readyQueue to wait for being executed. Lastly, we make the current thread sleep.

*public static void finish() {*

*block the interrupt function;*

*reset the status to statusFinished;*

*while( waitQueue is not empty ) {*

*send next waiting thread to ready queue*

*}*

*sleep current thread;*

*}*

**Test Cases**

Test 1:

Purpose (test scenario):

Test the currentThread call the join() method to join itself, the currentThread should only exit normally.

Steps:

1. Create a waitqueue; we write a debug message(Test case1: test about the currentThread try to join itself) in run method of KThread;

2. Then let currentThread to call join() method directly;

3. Write a debug message(TC1 should be exiting normally);

Output:

Test 2:

Purpose (test scenario):

Test the two normal thread can be successfully joined in another, the test 3 should exit any time after the joined thread finished.

Steps:

1. Create a waitqueue; we write a debug message(Test case1: try to join a thread) in run method of KThread;

2. We declare a new KThread called testCase2 in the currentThread’s run() method and set the loop of TestThread to 10

3. Then let testCase2 thread call the fork() and join()

4. Finally, write a debug message(TC2 should be exiting any time after thread 10sec delays)

Output:

Test 3:

Purpose (test scenario):

Test a finished thread to join the currentThread, the test should exiting normally and can not join a finished thread

Steps:

1. Create a waitqueue and write a debug message(Test case 3);

2. Then declare a new KThread called testCase3 in the currentThread’s run() method and set the loop of TestThread to 0;

3. Let TestCase3 call fork()

4. use if to judge whether the TestCase3 is finished.

5. Write a debug message(TC3 try to join a finished thread) to notice that a finished thread is joining

6. Let testCase3 call join method()

7. Write a debug message(TC3 should only exit normally, should not join a finished thread)

Output:

**Task 2 – Condition Variable**

**New Variables**

* Each condition variable needs lock variable, and the lock variable can also be acquired and released, and multiple condition variables can share the same lock variable.

*private Lock conditionLock;*

* In this task, we need a data structure to store the sleeping thread, and here use LinkedList to store them. When sleep() method is called, the current thread is put into LinkedList and let it sleep; when wake() method is called, the first thread on the LinkedList is wake up; when wakeAll() method is called, every thread on the LinkedList are wake up.

*private LinkedList<KThread> waitQueue;*

**Implementation Details**

* Sleep() method

When a thread calls the sleep method, first to releases the lock which is on a condition variable, at this time other threads can acquire this lock; then add this thread to the wait queue, and suspends itself until someone wakes it up, obtains the lock, and enters the critical section to execute. Switch interrupts are used to ensure atomicity of operations.

*public void sleep(){*

*block the interrupt function;*

*put the current thread into the wait queue;*

*let the current thread sleep();*

*resume the interrupt function;*

*}*

* wake() method

When the current thread in the critical region with Lock L performs a wake operation on the condition variable associated with Lock L, at most one of the suspended threads on the condition variable (caused by the call to sleep) is reawakened and puts it back on the ready queue.

*public void wake(){*

*block the interrupt function;*

*if (wait queue is not empty) {*

*send next wait queue thread to ready queue;*

*call ready() method;*

*}*

*resume the interrupt function;*

*}*

* wakeAll() method

When calls the wakeAll method, it will use a loop to wake up all suspended threads waiting on the condition variable (the wake() method is executed if the wait queue is not empty).

*public void wakeAll(){*

*while (wait queue is not empty)*

*execute wake() method;*

*}*

**Test Case**

* Test 1

Purpose (test scenario)

Test the condition variable by using 2 threads.

Steps:

1. create a lack variable and a condition variable;
2. in main thread, use new KThread() to create a new thread;
3. invoke sleep() method inside the lock, let current thread sleep and add it into wait queue;
4. use sleepThread.wake() method in main thread, wake up the first thread in the wait queue.

Expect result:

After use sleep() method, current thread will push into wait queue, after use wake() method, the first thread in the wait queue will wake up and push into ready queue; otherwise, if the current still being active this method has some problem.

* Test 2

Purpose (test scenario)

Test the condition variable by using many threads (assume 15 threads).

Steps:

1. create a lack variable and a condition variable;
2. in main thread, use new KThread() to create a new thread;
3. invoke sleep() method inside the lock, let current thread sleep and add it into wait queue;
4. duplicate the above step2 and 3 until it creates 15 threads
5. use sleepThread.wake() method in current thread, wake up the first thread in the wait queue and push it into ready queue.
6. use wakeAll() method in main thread, wake every thread up in the wait queue.

Expect result:

After use sleep() method, current thread will push into wait queue, after use wake() method, the first thread in the wait queue will wake up and push into ready queue, after use wakeAll() method, every thread in the wait queue will wake up; otherwise, if the current still being active this method has some problem.

**Part 3 – Alarm Class**

**New Variables**

* Alarm class newly create a private variable ---- waitingAlarmQueue for storing the sleep threads which will be awaken up by the alarm. It is implemented by Java’s PriorityQueue. The elements in this queue should be the new defined instances ---- waitingAlarmThread. This waitingAlarmThread has two arguments, threadCurrent represents the currentThread’s reference, waitUpTime represents the actual time that this waitingAlarmThread will wake up.

*private static PriorityQueue<waitingAlarmThread> waitingAlarmQueue;*

*class waitingAlarmThread {*

*define two local variables -- threadCurrent, wakeUpTime;*

*constructor( threadCurrent, wakeUpTime ) { ... };*

*}*

**Implement Details**

* waitUntil() method

waitUntil() method accepts an argument int x which represents the alarm time that the current thread has to wait then be awaken. For the code implementation, first of all, the interrupts need to disable. Then assign wakeUpTime by current machine time adding x. The following step is we create a new waiting thread contains the currentThread’s refference and the wakeUpTime. Next step the new waiting thread should be push to the waitingAlarmQueue for recording this waiting thread’s reference and wake up time. Then the current thread sleeps by sleep() method. At the end, resume the interrupts.

*public void waitUntil( int x ) {*

*disable interrupt function;*

*wakeUpTime = currentTime + x;*

*create a new waiting alarm thread instance – –*

*waitingAlarmThread( currentThread, wakeUpTime );*

*push this waitingAlarmThread to the waitingAlarmQueue queue*

*sleep( ) this current thread;*

*restore the interrupt function;*

*}*

* timerInterrupt() method

The timerInterrupt( ) method has to be called every 500 ticks. This timerInterrupt( ) method is responsible to wake the sleeping thread up in time like an alarm. Again, we need to disable the interrupt first. Then check if the waitingAlarmQueue is empty, if it is, run the code in the loop. Check the first thread which laying in the waitingAlarmQueue, if the machine’s current time exceeds the thread’s wake up time, should do two things: a) put this head thread to the ready queue; b) remove the first thread from waitingAlarmQueue. Again, restore interrupts when everything is done.

*public void timerInterrupt( ) {*

*disable interrupt function;*

*while( waitingAlarmQueue != empty ) {*

*if( the time of the head thread in waitingAlarmQueue <= current time ) {*

*put this head thread to readyQueue;*

*remove this head thread from waitingAlarmQueue;*

*} else {*

*// do nothing;*

*}*

*}*

*restore the interrupt function;*

*}*

**Test Case**

For this Alarm class, we have two goals regards to the test case:

* One is to check if timerInterrupt( ) method works, especially make sure the waitingAlarmQueue runs well and the threads which needs to be waken up successfully put into readyQueue.
* Another is to guarantee waitUntil( ) method properly set the alarm time and sleeps the current thread.

Thus, we suppose the test case includes the following steps:

* First, set the timerInterrupt( ) should be called every 500 ticks, output some words to make sure the call is on processing;
* Implement a loop that runs waitUntil( ) 10 times, and the alarm time x is to be set from long to short;
* Observe the output results, check if the threads awake in the right order.

**Task 4 – Communicator**

**New Variables**

* Lock is each communication is assigned a lock to guarantee the atomicity of the operation, and two condition variable speakers and listeners associated with the lock are used to ensure synchronization.

*lock=new Lock()*

* Listener and speaker have their own queue, we can check the queue to determine if there are any listeners and speakers available.

*queue=new LinkedList<Integer>()*

* Initialize the listener and speaker.

*speaker=new Condition2(lock);*

*listener=new Condition2(lock);*

*speakercount=0;*

*listenercount=0;*

* Set condition variables for speaker and listener.

*speaker = new Condition2(lock);*

*listener = new Condition2(lock)*

**Implementation Details**

* Speak() method

*Speak()*: first obtain the lock, then check if there is a listener waiting if there is a listener, put the words into the language queue, and wake up a listener. If there is no listener waiting, the number of speakers waiting is +1, put the words. After the language queue, sleep under the speaker condition variable () continues when the speaker is woken up: it should wake up a listener and will wait for the number of speakers -1.

*public void speak(int word) {*

*acquires the lock*

*if(listenerNum == 0){*

*number of speakers ++*

*speaker goes to sleep*

*wake up the listener by listener.wake()*

*number of speaker - -*

*}else{*

*output the word from the word queue*

*set flag to show that word is ready.*

*wake up all the listener*

*}*

*speaker releases the lock*

*}*

* Listen() method

*Listen():* Get the lock first, then make a judgment to try to wake up the speaker. If no one is waiting, put the listener in the queue and sleep. If a speaker waits, wake up a speaker, hang himself up to wait for the speaker to prepare the data and wake himself up, then pass the message and finally release the lock.

*public static void listen() {*

*listener acquires the lock*

*if(speakerNum!=0){*

*speaker wake up*

*listener sleep*

# *}else{*

*number of listeners++*

*listener sleep*

*number of listener- -*

*}*

*listener releases the lock*

*}*

**Test case**

* **Test propose**

Test A propose:

When we only have one listener and speaker, there will have a speaker and listener in the queue to wait another.

Test B propose:

When we have one speaker more listener, more speaker one listener and more speaker and more listener. Speaker may wait listener in the queue or listener wait speaker in the queue.

Test C propose:

Provide more message more speaker, and we have more speaker and more listener. Speaker or listener may have same number but have different order.

* steps

Test A steps:

1. Get a lock.

2.Generate the first speaker by c.speaker().

3.Generate the first listener by c.listener().

4. Release the lock.

Test B steps:

1. Base on the test A.
2. Use for loop to generate more speaker.
3. Use c.listener to create more speaker.
4. Release the lock.

Test C steps:

1. Base on the test B.
2. Provide more message for speaker to trans.
3. Release the lock.

* strategy

Test strategy:

A. The test method is to put only one listener or one speaker to test their performance.

B. The test method uses two threads in total. One thread calls the speak(int word) method five times, and the other thread calls the listen() method five times to simulate multiple listeners and speakers.

C. Add more messages based on the B method.

* result

Expected result:

A: Tests your communicator:

1. Test for one speaker, one listener, speaker waits for listener

2) Test for one speaker, one listener, listener waits for speaker

B: Tests your communicator, with more speakers/listeners

1. Test for one speaker, more listeners, listener waits for speaker
2. Test for one speaker, more listeners, speaker waits for listener
3. Test for one speaker, more listeners, listeners waits for speaker, and then create more listeners
4. Test for more speakers, one listener, listener waits for speaker

5) Test for more speakers, one listener, speaker waits for listener

6) Test for one listener, more speakers, speakers wait for listener, and then listener more speakers

C: Tests your communicator, with more speakers/listener, and transmits more messages

1. Test for more speakers, more listeners, listeners waits for speaker.
2. Test for more speakers, more listeners, speaker waits for listener.
3. Test for more speakers, more listeners, speakers and listeners have the same number but created with random order.

**Task 5 – Synchronization**

**New Variables**

* ReactWater need two queues to store hydrogens and oxygens. These hydrogens and oxygens will wait in queues until there are at least one oxygen and two hydrogens to make water. So, we add two private and static members to ReactWater.

*ObjectQueue hydroWait;*

*ObjectQueue oxyWait;*

* ObjectQueue class is designed to store the thread object in the queue. Inside the class, we declare two local variables(linked and queueLock). Linked is a list to store the objects and queueLock is used to make the thread wait. Then we also create some basic method to store the thread atom correctly. We use the removeFirst() method to remove the first atom in the queue.

Public class ObjectQueue{

Protected LinkedList <object>linked

Lock queueLock;

Public void add(KThread thread){….}

Public object removeFirst(){…}

}

* In order to track the number of hydrogens and oxygens waiting in the queue, we add two private members to record the count of them.

*Int hydroCount;*

*Int oxyCount;*

* The lock we need to make the thread wait until the condition is satisfied. We add a private member.

*Lock modLock;*

* The ReactWater constructor initializes hydroCount and oxyCount to zero and the ReactWater destructor deletes it.

**Implementation Details**

* hReady() method

hReady() simply stores a hydrogen atom in the hydroWait queue and add 1 to the hydroCount. Then call the MakeWater() method.

*Public void hReady(){*

*Increase the hydroCount by one*

*add the current thread in the hydroWait queue*

*call the makeWater() method*

*}*

* oReady() method

oReady() simply stores a oxygen atom in the oxyWait queue and add 1 to the oxyCount. Then call the MakeWater() method.

*Public void oReady(){*

*Increase the oxyCount by one*

*oxyWait.add(KThread.currentThread())*

*add the current thread in the oxyWait queue*

*call the makeWater() method*

*}*

* Makewater() method

Makewater() method judge whether there are at least two hydrogen and at least one oxygen in queues respectively. If the condition is satisfied, the method will make a water molecule, in other words, showing the debugging message. Then remove one oxygen thread and two hydrogen threads from queues. Lastly, hydroCount minus 2 and oxyCount minus 1.

*Public void MakeWater(){*

*Lib.assertTrue(modLock.isHeldByCurrentThread())*

*While hydroCount is greater than 1 and oxyCount greater than 1*

*Display debugging message*

*remove one oxygen from oxydroWait*

*remove two hydrogens from hydroWait*

*decrease hydroCount by two*

*increase oxyCount by one*

*}*

**Test Cases**

* Test 1

Purpose: Check if Makewater() method will be successful when there is only one hydrogen in the hydroWait queue.

Steps:

1. declare a KThread called water1

2. then get a Lock

3. only call hReady() method once

4. display the hydroCount in debugging message

5. release the lock

Expected Result: Show the hydroCount is 1 in the debugging message and the Makewater() method will not be triggered.

* Test2

Purpose: Check if Makewater() method will be successful when there are one hydrogen and one oxygen in the hydroWait and oxyWait queues.

Steps:

1. based on test1, we declare a Kthread called water2

2. then get a Lock

3. only call oReady() method once

4. display the hydroCount and oxyCount in debugging message

5. release the lock

6. lastly, we call water2.fork() method

Expected Result: Show the hydroCount is 1 and the oxyCount is 1 in the debugging message and the Makewater() method will not be triggered.

1 Hydrogen Atom and 2 Oxygen test3: on the basis of test2, we create a Kthread called water3 and call the oReady() once in the run() method. Then, we call the water3.fork() and water2.join(). It will generate three Hydrogens in the hydroWait queue.The Makewater() method will also not be triggered because there are only three Hydrogens and zero Oxygen in queues.

* Test3

Purpose: Check if Makewater() method will be successful when there are one hydrogen and two oxygen in the hydroWait and oxyWait queues.

Steps:

1. based on test2, we declare a Kthread called water3

2. then get a Lock

3. only call oReady() method once

4. display the hydroCount and oxyCount in debugging message

5. release the lock

6. lastly, call water3.fork() method

Expected Result: Show the hydroCount is 1 and the oxyCount is 2 in the debugging message and the Makewater() method will not be triggered.

* Test4

Purpose: Check if Makewater() method will be successful when there are two hydrogen and two oxygen in the hydroWait and oxyWait queues.

Steps:

1. based on test3, we declare a Kthread called water4

2. then get a Lock

3. only call hReady() method once

4. display the hydroCount and oxyCount in debugging message

5. release the lock

6. lastly, call water4.fork() method

Expected Result: Show the hydroCount is 0 and the oxyCount is 1 in the debugging message and the Makewater() method will be triggered.