**Programming Assignment 3**

**Sharanya Sudhakar**

**CSS 430**

**11/10/19**

Part 1:

**SyncQueue.java**

*SyncQueue* allows a thread to sleep and wakeup on a specific condition. This monitor is implemented with a *QueueNode* as its inbuilt property. *SyncQueue* enables the implementation of *SysLib.join()* and *SysLib.exit().* Join and exit are implementations in the Kernel.java that utilizes the *SyncQueue* for implementing the *SysLib.join()* and *SysLib.exit()* calls of the thread based on a condition. The underlying Queue is *QueueNode*. This implementation is based on the Unix/Linux platform where the parent thread waits for the child thread to terminate. The child thread terminates by calling *SysLib.exit()* and it return the id of the child thread that wakes up the sleep parent thread in turn.

To implement *SysLib.join()* and *SysLib.exit(),* *Kernel.java* is modified to implement *SyncQueue* and *SyncQueue* is implemented to have an underlying *QueueNode*.

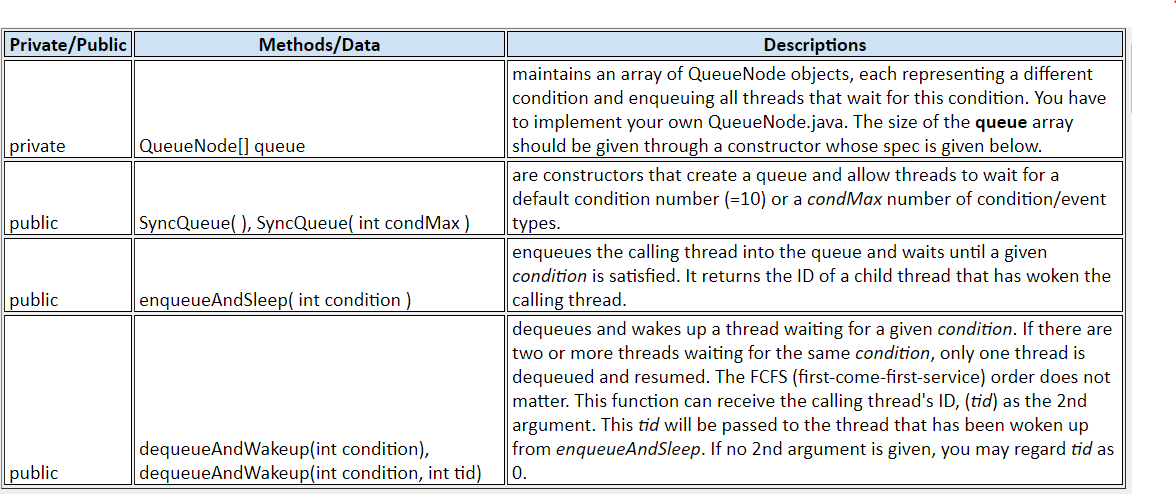
***Kernel.java***

In this file, under the WAIT and EXIT cases are modified to ensure the parent id is retrieved and put to sleep. And in the exit case, the parent is woken up by sending the id of the child that exited.

|  |  |  |
| --- | --- | --- |
| **Private / Public** | **Method / Date** | **Description** |
| CASE | WAIT | Get the *TCB* of the currently running thread and use it to get the *Tid* in order to *enqueueAndSleep()* that thread. |
| CASE | EXIT | Get the *TCB*, *dequeueAndWakeup()* thread using *pid* and *tid*, then delete the thread in scheduler. |

In ***SyncQueue*.*java*** file,

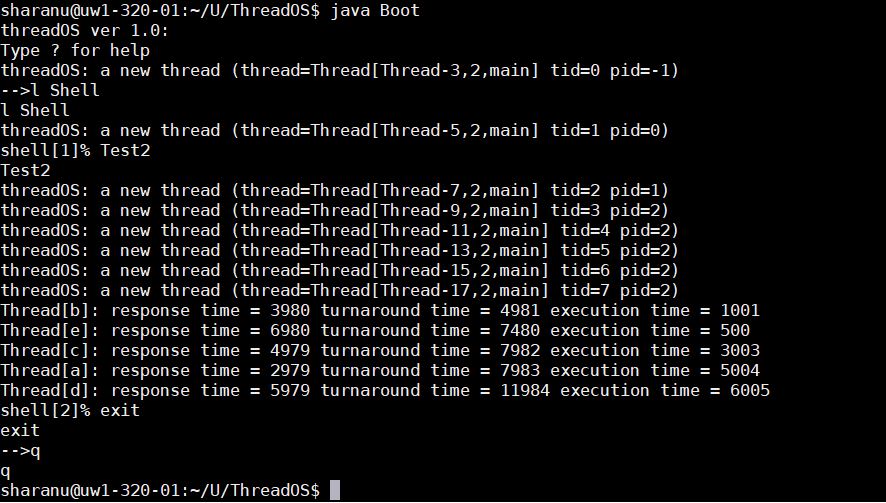
The class is initialized with a default constructor having 10 threads or a parameterized constructor with a custom number of threads. Then the two methods within the class are *enqueueAndSleep* and *dequeueAndWakeup*. In *enqueueAndSleep*, the thread with the condition is put to sleep and in *dequeueAndWakeup*, the thread is woken up with its child *tid* as a parameter or a default child of 0(zero). The design of Syncqueue.java follows the assignment specifications:



In ***QueueNode****.****java***

The class has an underlying queue which holds the child threads. This class has a vector that represents the threads with the same condition. There are two methods, *sleep()* and *wake()*. In the *sleep()*method, the thread is put to sleep until notification. It will return the id of the calling thread. In *wake()* method, the thread with *tid* is added to the queue and notifies the sleep method.

|  |  |  |
| --- | --- | --- |
| **Private / Public** | **Method / Date** | **Description** |
| Private | Vector<Integer> queue | Data structure to hold the threads that are enqueued by *wake()* |
| Public synchronized | Sleep() | The sleep method returns the id of the first thread that wakes up the thread in sleep() |
| Public synchronized | Wake() | This method enqueues the *tid* of the thread into the vector array and notifies *sleep*() |



Output for part1.

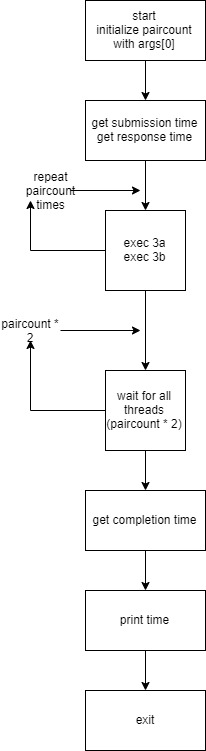
Part 2

In this section, three test classes where created to test the Kernel implementation of the RAWREAD, RAWWRITE and SYNC so that disk access no longer has spin loops.

Modifying the *Kernel.new /* ***Kernel.java***

Instead of the spin loop, the *ioqueue* is added to *enqueue* and *sleep* the threads based on certain condition.

***MyTest3.java***

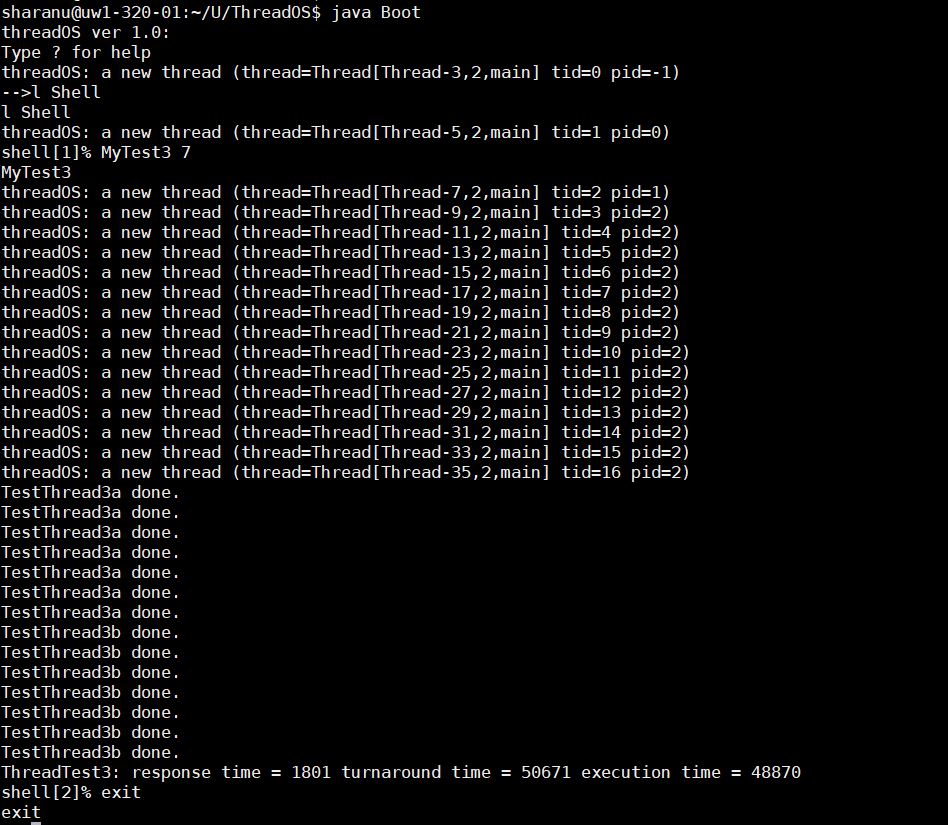
This class executes user threads as pairs. There are two threads to be executed. TestThread3a and TestThread3b. 3a, contains computational execution, while 3b contains disk read/write(I/O) operations. These threads are executed alternately, with an input argument number of times.   


***TestThread3a.java***

This thread performs computation. It produces the factorial and in-turn finds the *tan* and *atan* multiple times. The out put is never printed, this is a computationally intense thread. Its one and only purpose.

***TestThread3b.java***

This thread is a read operation only. It reads 512 byte blocks of data 250 times. Making it I/O intense.

Both threads print when done.

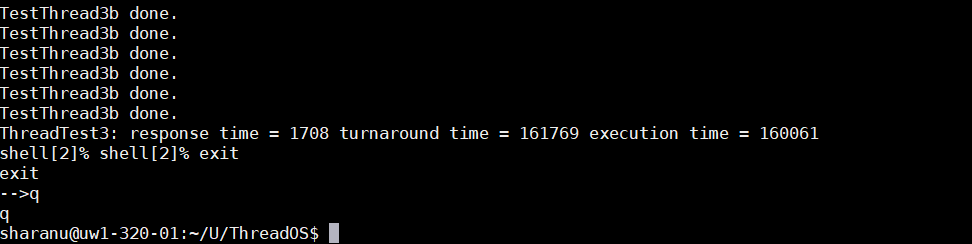
New Kernel Output.

**Discussion:**

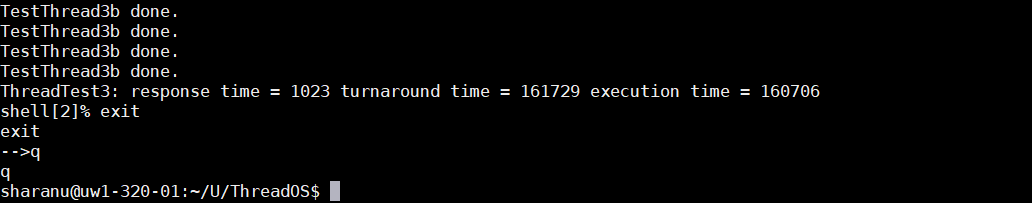
The output for 25 pairs of threads and 7 pairs of threads was implemented for both old and new Kernel. Looking at the output, it is obvious that the old kernel is faster than the new kernel. In the new kernel execution, the thread with computational intensity is completed first, while the I/O intense thread executes at the end. On the other hand, the old kernel has interleaved output. Meaning the waiting doesn’t happen due to context switch as much as in the new Kernel. Every I/O operation in the new kernel is interrupted and context switched to the computational threads, enabling this overhead that slows down the execution.

With the given thread tests in small pairs, it is clear that the old Kernel has better execution time. In other cases when number of threads increase, NEW kernel is faster by a margin. As shown in the case of 25 pairs. So every test case is different and old or new the kernel efficiency depends on the number of threads executed.

**NEW Kernel: 25 pairs**

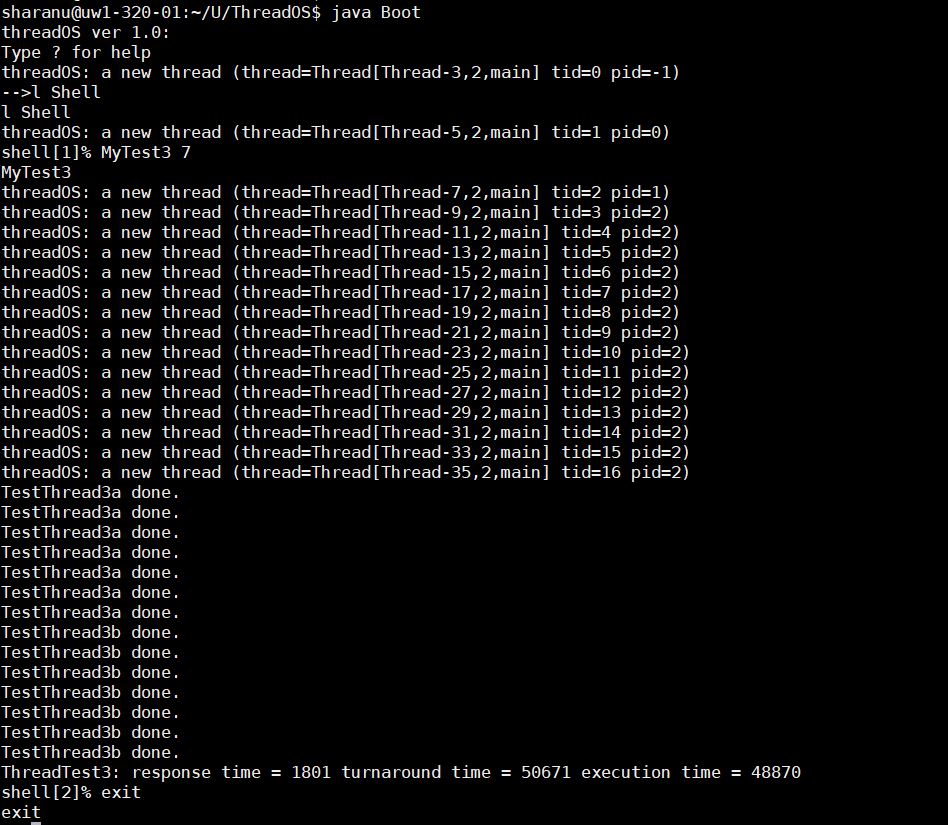


**OLD Kernel: 25 pairs**



Output for 7 pairs below:

NEW KERNEL 7 pairs:



OLD KERNEL 7 pairs:

