Mustafa Majeed

CSS 430

P3 Report

**Part 1**

**SyncQueue and QueueNode Algorithm Description**

Part 1 of this project was to implement a SyncQueue and QueueNode java classes that work with WAIT and EXIT SysLib calls from Kernal.java.

SyncQueue is a class that contains a private QueueNode array. This array is designed to hold QueueNode objects that will represent threads waiting for a spawned child to finish execution via the SysLib.join() call.

*SyncQueue.java*

SyncQueue has 2 public methods. enqueueAndSleep method puts the calling method in wait() state by calling QueueNode.sleep(). This method will return a thread id of any child that has notified the parent thread that it has finished executing.

The second method is dequeueAndWakeup() and it is called by any thread using SysLib.exit() ThreadOS system call. This method calls QueueNode.wake() and is meant to be used to notify a parent thread that a child has finished executing.

*QueueNode.java*

QueueNode has 2 methods that are both synchronized in java. This means that only 1 thread is allowed to execute these methods at a time. QueueNode also has a Vector of child thread ids that have finished executing.

The first synchronized method is sleep(). The calling object of this method will wait for a notify from a child thread that is has spawned. It will return a thread id of a child that has notified it to continue when the child has finished.

The second synchronized method is wakeup(). The calling object of this method will have its thread id added to the Vector of child tids and will notify the waiting parent that it has finished executing, so the parent can continue.

**Part2**

**Test3 Description**

Test3 is a class that was created and used to test elapsed time when running simultaneous CPU and DISK I/O threads using a busy wait IO kernel vs an asynchronous IO kernel.

Test3 extends Thread in java and overrides run(). Test3 accepts a string parameter that will represent how many pairs of computation and disk threads to launch. Run() will record the start time of execution, launch the requested pair of computation and disk threads using TestThread3a and wait for all created threads to finish execution before recording the end time of execution and displaying the total time elapsed for Test3.

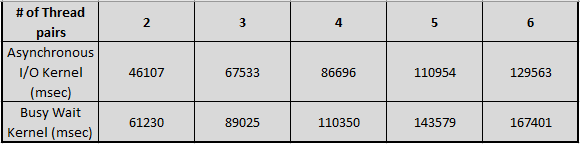
**TestThread3a**

TestThread3a accepts a string parameter that will guide which algorithm to execute. If the passed in string is “computation”, TestThread3a will execute a mathematical computation. If the passed in string is “disk”, TestThread3a will attempt to read from the DISK using SysLib.rawread.

TestThread3a will display a message when it has finished executing and is about to call SysLib.exit().

**Performance Results**

**Figure 1: Asynchronous IO vs Busy Wait IO Result Table**



*Busy Wait IO Kernel (Kernel.old)*

The busy wait kernel was able to perform the CPU bound threads very quickly and they were always the first to be completed and displayed on the screen. The IO bound threads took a long time to execute and as shown in Figure 1, increased by more than 20 seconds with each additional thread pair requested.

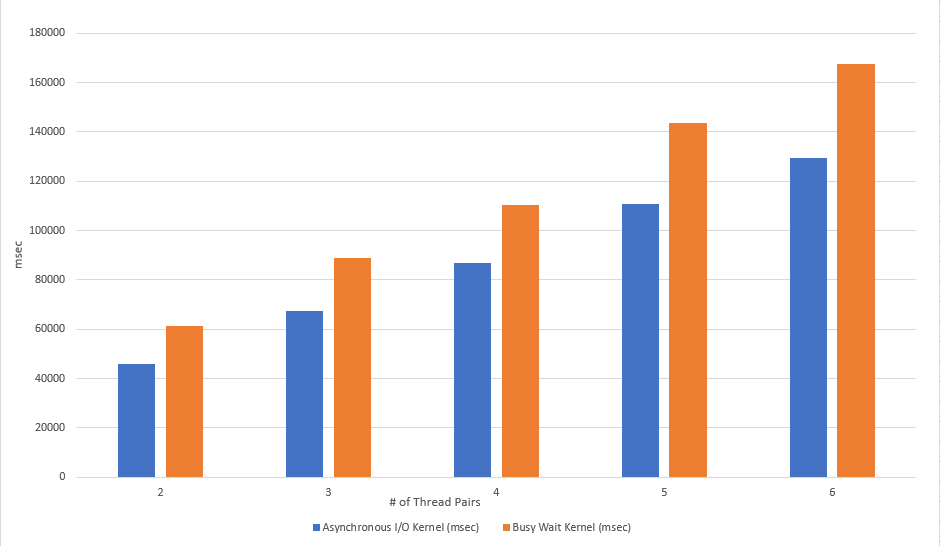
*Asynchronous IO Kernel*

The asynchronous kernel was able to perform CPU bound threads just as quickly as kernel.old and although IO bound thread still took quite a while to execute, it was not as long as kernel.old. As shown in Figure 1, with each additional thread pair requested, the elapsed time increased by about 20 seconds.

***Technical issues***

During testing of the Asynchronous IO Kernel, I kept running into an issue with ThreadOS hanging and not responding. This would continue to happen until I would delete the created DISK that ThreadOS creates on 1st boot up. This would force ThreadOS to create a fresh DISK which seemed to allow me to do testing a few more times before it would hang up again and I would have to repeat the process. I found out later that it is a race condition in ThreadOS Disk.class.

**Figure 2: Asynchronous IO vs Busy Wait IO Result Graph**



*Performance Comparison*

Figure 2 illustrates how much faster asynchronous IO kernel is compared to busy wait IO kernel. The elapsed time is about 20 seconds less in asynchronous IO kernel from 2 pairs to 4 pairs of spawned threads and nearly 30 seconds less when Test3 spawns 5 or 6 pairs of threads.

Technically this should be the case when you have many threads all fighting for CPU time. When an IO operation is in wait() mode, it will yield its CPU time to another thread that needs to use it rather than spinning in a busy wait loop for its allotted time quantum on the CPU. So that’s why the asynchronous IO kernel can finish executing faster than a busy wait IO kernel.

As the number of requested pairs of threads increase when running Test3, the gap will get wider between the 2 implemented kernels because the number of threads fighting for CPU time will also increase and figure 2 shows this type of growth in the gap of elapsed time between the 2 types of kernels tested in P3.