Ethan Crawford

CSS 430

November 13, 2008

Program 3 - SyncQueue

# Part 1

## Kernel.old:

## SyncQueue.java

## Result when running Test2.class from Shell.class

[ethancr@uw1-320-18 ThreadOS]$ java Boot

threadOS ver 1.0:

Type ? for help

threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)

-->l Shell

l Shell

threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

shell[1]% Test2

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=2)

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=2)

threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=2)

threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=2)

threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=2)

Thread[b]:(1000) response time = 4007 turnaround time = 5028 execution time = 1021

Thread[e]:(500) response time = 7012 turnaround time = 7522 execution time = 510

Thread[c]:(3000) response time = 5009 turnaround time = 8069 execution time = 3060

Thread[a]:(5000) response time = 3005 turnaround time = 8106 execution time = 5101

Thread[d]:(6000) response time = 6010 turnaround time = 12130 execution time = 6120

shell[2]% exit

exit

-->

## Specification and description

SyncQueue uses Java ‘synchronized’, ‘wait’, and ‘notify’ functionality combined with a list of user-defined events to implement thread synchronization.

The enqueueAndSleep() method calls ThreadSyncNode.\_wait(). This uses the ThreadSyncNode object as a monitor for a specific event (or “condition”) and blocks the calling thread until dequeueAndWakeup() is called, which calls ThreadSyncNode\_notify() to wake up the blocked thread.

Within ThreadSyncNode, both methods are synchronized to prevent deadlocks. See figure 1.



Figure 1: SyncQueue/ThreadSyncNode class diagram

# Part 2

## Kernel.java (with SyncQueue in place of I/O spin-locks)

## Test3.java

## TestThread3.java

## Performance Results for Test3.java on Kernel.old (spinlock):

threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)

Thread[CPU]: response time = 4001 turnaround time = 17086 execution time = **13085**

Thread[FileSystem]: response time = 2999 turnaround time = 22145 execution time = **19146**

Thread[FileSystem]: response time = 1999 turnaround time = 29142 execution time = **27143**

Test3 finished

## Performance Results for Test3.java on Kernel.java (SyncQueue):

threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)

Thread[CPU]: response time = 4001 turnaround time = 16474 execution time = **12473**

Thread[FileSystem]: response time = 1998 turnaround time = 11054 execution time = **9056**

Thread[FileSystem]: response time = 2999 turnaround time = 19867 execution time = **16868**

Test3 finished

## Specification and description

Test3.java creates 3 TestThread3 threads, 2 that are I/O-heavy and one that is CPU-intensive. Each thread reports execution time as it exits.

## Performance Results Analysis:

Wait/notify-driven thread synchronization using SyncQueue shows itself to be significantly more efficient than spinlock over lengthy I/O cycles, due to the lack of CPU consumption during the wait() interval in a SyncQueue(). The benefit is somewhat lessened by the need for a context switch when rotating between threads, and is nearly non-existent when the number of I/O iterations is smaller (less than 60). This suggests that spinlock is likely the most efficient thread sync implementation when burst cycles are small, and wait/notify-driven techniques are more efficient when burst cycles are long.