Sharan Babu Paramasivam Murugesan (sxp141731)

Report – Project 1

Experimental setup:

- Used Java for development
- External library for exponential random number generation which is used for interlock request delay among the threads
- Critical section used is an integer counter – each thread executes 1000 times (so final value expected is num_threads * 1000).
- Through put is measured as total time for the program to complete
- Results are plotted with two parameters – interlock request delay, number of threads against system throughput.
- System details:
  Processor : i7 – quadcore machine with 6MB L3 cache

Algorithm for lock :

Shared variables:

- peteTree[] – array of instances of Peterson's algorithm
- Each instance of Peterson's algo contains Boolean[] flag and int victim as shared variables

Idea:

- An array of Peterson's algorithm instances of size equal to number of nodes in the binary tree(number of nodes is calculated manually)
- The array of instances is a representation of binary tree with leaves in the beginning of the array and root at the end of the array.
- So a thread has to reach the end of the array competing all the way through from the leaf, to reach the critical section.
- If it loses the lock at any level of the tree, it will be waiting at that level and it can only move further if the thread to whom it had lost unlocks it after it finishes the CS

PseudoCode:

Lock()

```
me = current thread id
for(h :0 to height - 1)
    i = me
    me = floor(me/2) + numTheads
    peteTree[me].lock(i)
```

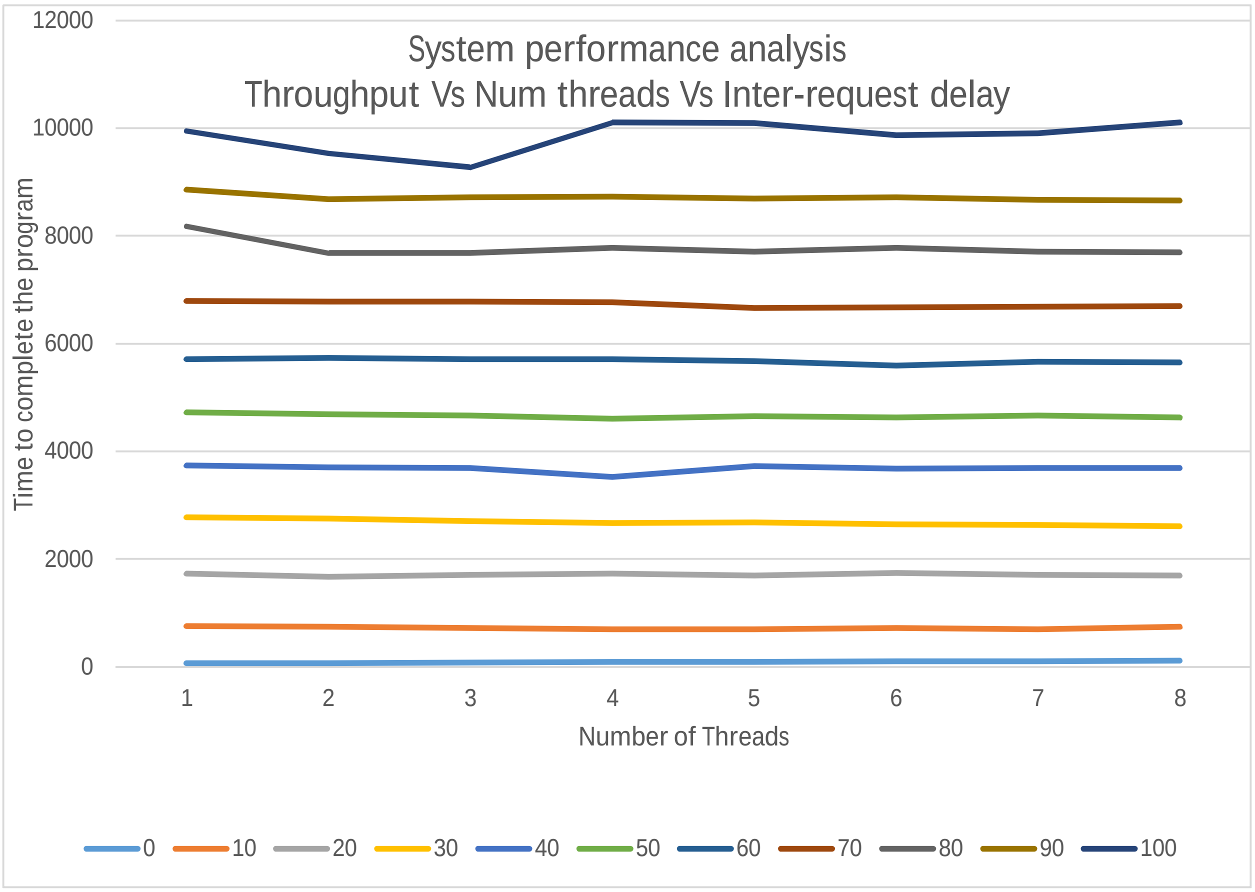
```
UnLock()
    list pathToRoot
    me = current thread id
    for(h :0 to height - 1)
        me = floor(me/2) + numTheads
        pathToRoot.add(me)
    for(j : pathToRoot.size() – 1 to 1)
        peteTree[pathToRoot[j]].unlock(pathToRoot[j-1])
```

Results:

Below shown results are averaged over 10 runs.

Graph parameters:

X axis – number of threads varied from 1 to 8 (2X number of cores)
Y axis – time taken by the program to finish in milliseconds.
colored lines – system throughput measured against number of threads for different values of inter-request delay from 0 time unit till 100 time units (1 time unit = 0.1 ms)



Observation:

It is observed that when number of threads increases, system throughput increases (time taken to complete the program becomes lesser in terms of milliseconds). Though the lines seems to be flattening out in the graph, the time taken to finish the program actually gets reduced as the number of thread increases for various inter-request delay. Also one other observation made is when the number of threads increases at high system load (low inter-request delay), the through put decreases. Though this behavior is not consistent, it is observed at times. e.g., when the number of thread is increased from 4 to 5 or 6 when the inter-request delay is 10 Time units, through put decreases. This explains the fact that when the number of threads increases during high system load, number of contention increases, making every thread compete for the lock leading them to spin on the lock. When the inter-request delay Vs throughput behavior is observed, it seems like throughput doesn't increases significantly when the system load is decreased. The delay caused to reduce the system load actually becomes an overhead for the system throughput. The time taken to compute the delay time interval (from a java package for exponential distribution) is taking most of the program run time and it varies for every run since the sampling of those values from the package is done randomly. This inconsistent time for delay interval computation also might explains the reduced system throughput even though the system load is reduced.