Scalable Synchronization Techniques for Multicore Systems

Sarvesh Sathish[1], Neeraj Mittal[1]
University of Texas at Dallas [1]Advanced Networking And Dependable Systems Laboratory
Anson L. Clark Research Program

Introduction

With an alarming increase in computing resources, Moore's law states that the number of transistors on a single-core chip will double every two years which supersedes the rate at which CPU cooling techniques can keep up with. Recent advancements suggest an alternative to migrate to multi-core processing units to efficiently use computing services simultaneously and to harness the entire computing power of the machine. With every core having their unique shared memory model, concurrent programs are difficult to write as each thread needs to share data through a linked memory model. Failure to effectively distribute cache within CPU resistors lead to a phenomenon known as a "Race Condition". To eliminate race conditions, efficient Mutual Exclusion (ME) algorithms are implemented through locks. A Peterson Lock is used when two concurrent processes attempt to enter the critical section of a given process. Through this work, this poster proposes to apply the concept of a Peterson Lock to a 'n' node binary tree. The tree consists of 'n' Peterson locks on $\log_2(n)$ levels to allow threads to synchronize their actions to effectively distribute data through interlinked memory systems. With this tournament-based data structure, each thread must acquire a lock in each of their respective levels until they reach the root node of the given binary tree. While a Peterson lock is used amongst only two threads, this algorithm is a more advanced variation where it can access any 'n' threads and each thread must only access $\log_2(n)$ locks to enter the critical condition.

Multi-Core Programming

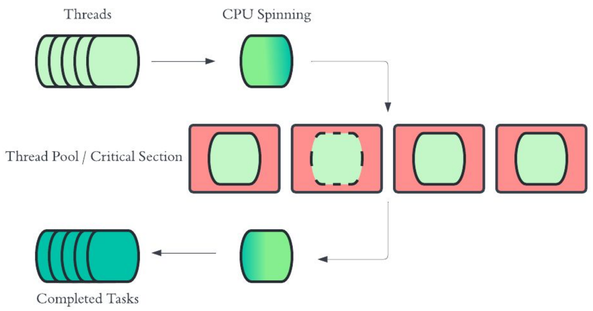

Figure 1: Multi-Core Programming Concept using Thread Pool

Lock Mutex

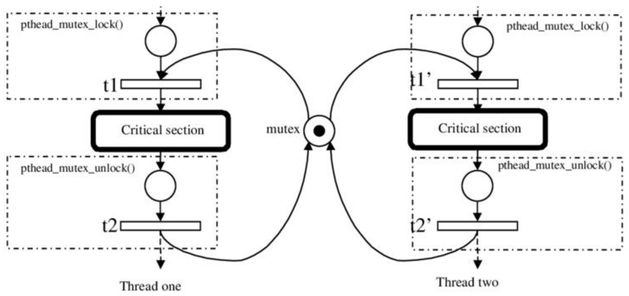

Figure 2: Two Concurrent Threads fighting for lock to enter critical section [3] (Kavi)

Binary Tree Architecture

Section I: Peterson Algorithm with Two Threads

Peterson Lock Algorithm is a deadlocked solution to race condition between two concurrent processes. It utilizes a 2-D Array *flag*[2] to describe a processes' need to enter the critical section and a variable "*turn*" that allows the other process to enter critical section if it's *flag* index is true/1.

Algorithm 1 Definition of Peterson Node Structure

```
1:  function PetersonNode(PetersonNode left, PetersonNode right, PetersonNode left)
2:      flag[0] = flag[1] = 0
3:      turn = 0
4:  end function
5:  function lock(i)
6:      flag[i] = 1 j = 1 - i                          ▷ Other thread
7:      turn = j
8:      while(turn == j  flag[j] == 1)                 ▷ Keeping Spinning
9:  end function
10: function unlock(i)
11:     flag[i] = 0
12: end function
```

The process performs something called a "CPU Spin" to wait and acquire a lock to enter the critical section. While the current ID is denoted with "*i*", the other thread ID can be computed through "*1- i*". The thread stops spinning when the lock is unlocked through its *flag* index being set to false/0.

Section II: Tree Instantiation

The binary tree is represented by a 2D array of Peterson Locks represented as nodes. Each level has double the amount of nodes than in the previous level when traversing downwards.

- Parent node's index is half of the child's node
- Child's node is left if the threadID is odd and vice-versa

Algorithm 2 Peterson Tree Initialization

```
1:  function PetersonTree(size)
2:      numLevels = log2(size), nodesPerLevel ← 1     tree[][]   ▷ numLevels is the number of rows in the array
3:
4:      tree[0][0] = new PetersonNode()
5:      for i ← 2 to numLevels-1 do
6:          nodesPerLevel *= 2
7:          tree[i] = new PetersonNode [nodesPerLevel]
8:          for j ← 0 to nodesPerLevel-1 do
9:              tree[i][j] = new PetersonNode()
10:             tree[i][j].parent = tree[i-1][j/2]                ▷ Set Parent Node
11:             tree[i-1][j/2].left or right = tree[i][j]         ▷ Set Child Node Respectively
12:         end for
13:     end for
14: end function
```

Section II : Leaf Node Retrieval

A leaf node represents the nodes on the base of the tree. The threads access their respective Leaf Node by dividing it's ID by 2.

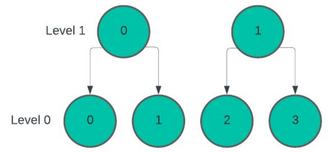

Figure 3: Leaf Node Diagram with Binary Tree Representation

Section III : Lock Inorder Traversal

The respective locks are traversed based on the binary representation of the thread ID. If the binary representation of the thread ID is represented on an array and each index represents a level: 0 signifies the left child and 1 means the right.

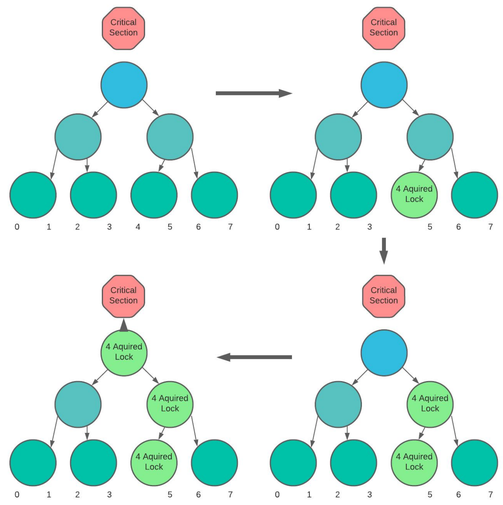

Figure 4: CPU Resistor Linked Lock Acquiring Diagram

Performance Metrics Evaluation

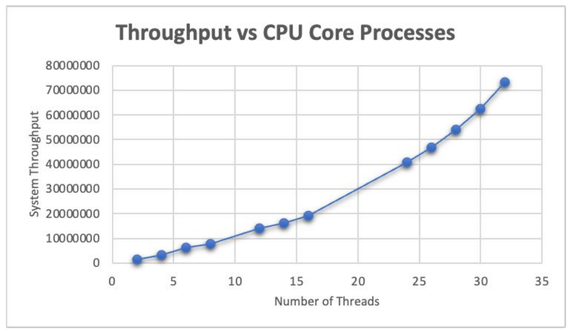

Figure 5: Graphical Comparison between CPU Processes vs System Operations

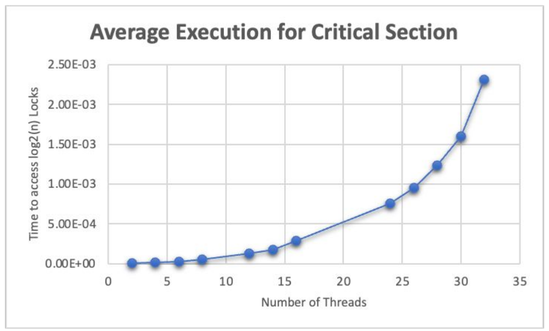

Figure 6: Graphical Analysis between Processes and Time to Enter Critical Section

Discussion

The algorithm was tested 10 iterations for each 'k' threads. For every n nodes, there are $\log_2 n$ levels where each thread iterates through $2 \times \log_2 n$ nodes to enter and exit the critical section. Time to critical section grows slower as n increases.

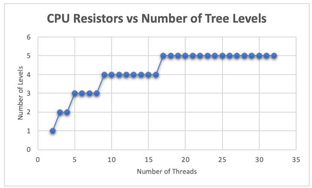

Figure 7: logarithmic Expansion of Tree Levels

Future Research

Future research could improve on the following factors:

Complexity

The number of locks to be acquired to access critical condition with a different ADT.

Fairness

This algorithm isn't fair in with only two threads being able access to access a Node. Stack could replace the Node.

Memory

The size of the binary tree increases in depth and could later be implemented to grow in width concurrently.

Conclusion

1) The mutex proved to be effective in solving race condition with a consistent end result but time taken to acquire and unlock locks increased logarithmically with the threads.
2) Future research could incorporate different ADT such as a queue to avoid a logarithmic expansion of the binary tree and to reduce a waste of memory resources with nodes that are not used..
3) Each Node could utilize a stack to store 'k' concurrent processes.

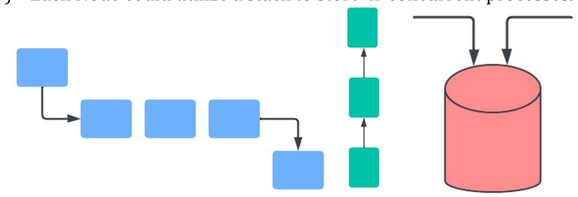

Figure 8: Future research in Persistent ADT entails Queues, Stacks, Linked List, Hash

References

Sirhan, Najem N., Multi-Core Processors: Concepts and Implementations [Accessed June 16, 2020].[1]

M. Rouse, "Definition: multi-core processor." TechTarget. [Retrieved 20 Jul 2022). [2]

Modeling Multithreaded Applications Using Petri Nets - Scientific Figure on ResearchGate. [Accessed 20 Jul, 2022] [3]

Shankar, A.. (2013). Lock Using Peterson's Algorithm. 10.1007/978-1-4614-4881-5_9. [Accessed 20 Jul, 2022] [4]