Implementing Filter Lock and Peterson-based Tree Lock

Kushagra Indurkhya
(CS19B1017)

Introduction

For n thread locking solution I have tried to implement it in two ways:

- Filter Lock: A direct generalization Of Peterson's algorithm for n threads (The Art of Multiprocessor Programming Maurice Herlihy, Nir Shavit 2.4), is implemented exactly as per discussions in class and in the book.
- Peterson Tree Lock: (The Art of Multiprocessor Programming Maurice Herlihy, Nir Shavit 2.10 Exercise 13)
    - **n(number of threads) needs to be in the power of 2**
    - Arrange a number of 2-thread Peterson locks in a binary tree.
    - Each thread is assigned a leaf lock which it shares with one other thread.
    - Each lock treats one thread as thread 0 and the other as thread 1.
    - In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root.
    - The tree-lock's release method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired.

For testing these solutions as mentioned in the assignment instructions I create n threads which in turn have critical sections which they call k times.

Both the filter lock and the tree-based lock implements lock and unlock functions which are virtual functions from a lock-base class

```
/* This is the base class for all locks. */
class lock_base
{
public:
   virtual void lock(int thread_id) = 0;
   virtual void unlock(int thread_id) = 0;
};
```

Filter Lock

```
class filter : public lock_base
{
private:
   /* data */
   int *level;
   int *victim;
   int n;

public:
 /* This is the constructor of the filter class. It initializes the level and victim
arrays to 0. */
   filter(int n)
   {
       level = new int[n];
       victim = new int[n];
       this->n = n;
       for (int i = 0; i < n; i++)
           level[i] = 0;
   }
  /* This is the implementation of the Filter Lock algorithm. */
   void lock(int thread_id)
   {
       int me = thread_id;
       for (int i = 1; i < n; i++)
       { // attempt level 1
           level[me] = i;
           victim[i] = me;
           // spin while conflicts exist
           for (int k = 0; k < n; k++)
               while ((k != me) && (level[k] >= i && victim[i] == me)){}
       }
   }
   /* This function sets the level of the thread to 0. */
   void unlock(int thread_id)
   {
       level[thread_id] = 0;
   }
};
```

- In constructor 2 arrays level and victim of size n are initialized
- When a thread acquires the lock
- For every level i:
    - Level[thread_id] is set to i
    - Victim[i] is set to thread_id
- while ((∃k != me) (level[k] >= i && victim[i] == me)) {}; is the spinning condition which can be translated to code as:
    - for (int k = 0; k < n; k++)
    - while ((k != me) && (level[k] >= i && victim[i] == me)){}
- For Unlocking simply level[thread_id] is set to 0

Tree-based Peterson Lock

Peterson Lock Node

A binary tree is created with n/2 leaves where each node is a slightly **modified Peterson lock which takes in 2 threads for this lock those two threads are treated as 0,1**

- When two threads enter a Peterson lock they are treated as 0,1 for that particular node
- Outside their thread_ids maybe anything
- To facilitate this mapping i have used two addition variables i_id,j_id

Internal 0th Thread_id= i_idth External thread_id

Internal 1th Thread_id = j_idth External thread_id

Each node has:

node *leftChild,*rightChild,*parent; => for binary tree

atomic_bool *flag , atomic_int victim; => As a simple peterson lock requires

int i_id,iint j_id; => additionals to facilitate n thread locking

Constructor

```
node() {}
node(node *par)
{
    flag = new atomic_bool[2];
    parent = par;
    i_id = -1;
    j_id = -1;
}
```

- Takes in par (parent Node) sets parent to par
- Initializes the flag array of atomic bool size 2
- Initializes i_id and j_id to -1

Lock and Unlock

- When a thread requests this Peterson lock first it checks if either i_id, j_id have never been set if that's the case then this incoming thread_id is mapped to 0,1 depending on the availability.
- If Both have been previously set it check if either the 0th or 1st thread in this Peterson lock is requesting the lock (flag[i] set to true),
    - if not then the incoming external id is mapped to that internal thread id is mapped
    - Else it waits for either thread to release the lock
- With this mapping in place, the Peterson lock works in the usual way.
- While Unlocking a thread_id simply the mapping is checked and corresponding 0th or 1st lock is release

```
   void lock(int thread_id)
   {
       //i_id is not set yet so set it to the thread_id requesting this lock
       if (i_id == -1)
           i_id = thread_id;
       // i_id is not set but j_id is not set so set j_id to the thread_id requesting
this lock
       else if (j_id == -1)
           j_id = thread_id;
       //both i_id and j_id are set
       else
       {
           //0th peterson thread is not requesting the lock
           if (flag[0].load() == false)
               i_id = thread_id;
           //1st peterson thread is not requesting the lock
           else if (flag[1].load() == false)
               j_id = thread_id;
           //both peterson threads are requesting the lock
           else
               //Wait till either frees up
               while (flag[0].load() == true && flag[1].load() == true){}
       }

       int i;
       if (thread_id == i_id)
           i = 0;
       else
           i = 1;

       int j = 1 - i;
       flag[i].store(true);
       victim.store(i);
       while (flag[j].load() && victim.load() == i){}
   }
   void unlock(int thread_id)
   {
       int me;
       if (thread_id == i_id)me = 0;
       else if (thread_id == j_id)me = 1;
       flag[me].store(false);
```

Binary Tree of locks

Overview

- A binary tree is created with n/2 leaves where each node is a slightly **modified Peterson lock which takes in 2 threads for this lock those two threads are treated as 0,1**
- Each thread is assigned a leaf lock which it shares with one other thread.
- Each lock treats one thread as thread 0 and the other as thread 1.
- In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root.
- The tree-lock's release method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired.

Ex

- 16 thread we will create a binary tree with 8 leaf nodes and total 1+2+4+8 = 15 nodes
- When thread 8 or 9 acquire the lock it acquires the leaf node-4=8/2=9/2 lock then makes its way to the root locking all the nodes in the path
- Similarly it release the lock it starts from the leaf node to the root node unlocking all the nodes in its path

<table>
<tr><td>0</td><td colspan="8">Root</td></tr>
<tr><td>1</td><td colspan="4"></td><td colspan="4"></td></tr>
<tr><td>2</td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td>3</td><td>0/1</td><td>2/3</td><td>4/5</td><td>6/7</td><td>8/9</td><td>10/11</td><td>12/13</td><td>14/15</td></tr>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
</table>

Constructor

- Leaf_nodes is a vector of leaf_nodes since other levels can be accessed using parents pointer i havent stored the other level vectors
- In constructor
  - For the 0th level i have created a vector and pushed the root to it
  - Then this vector is passed to the function recursiveBuild which builds the tree further

```
node *root; //root node of the tree

vector<node *> *leaf_nodes; //vector of leaf nodes

int height; //height of the tree

ptl(int n)

    this->n = n;//number of threads

    root = new node(NULL);//root node

    vector<node *> *tree = new vector<node *>; //initialize the tree

    tree->push_back(root);//push the root node to the tree

    leaf_nodes = recursiveBuild(tree);//recursively build the tree

    height = log2(n);//calculate the height of the tree
```

- This function takes in a vector of nodes
- Base case is the nodes vector size is n/2 ie that last level is reached.
- The level below the current nodes is built in new_nodes vector byiterating over nodes and creating and pushing their left and right Children in new_nodes

```
vector<node *> *recursiveBuild(vector<node *> *nodes)

{

    if (nodes->size() == ((this->n) / 2))

    {

        return nodes;

    }

    vector<node *> *new_nodes = new vector<node *>;
```

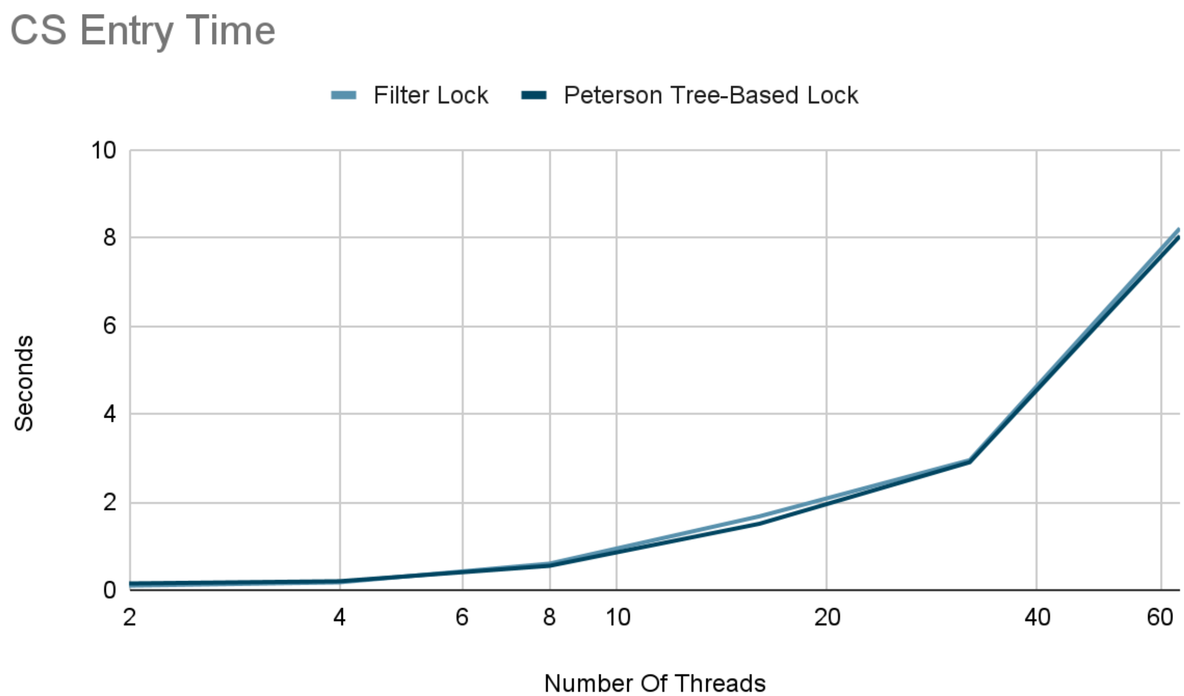
```
        for (int i = 0; i < nodes->size(); i++)

        {

            node *parent = nodes->at(i);

            node *left = new node(parent);

            node *right = new node(parent);

            parent->leftChild = left;

            parent->rightChild = right;

            new_nodes->push_back(left);

            new_nodes->push_back(right);

        }

        return recursiveBuild(new_nodes);

    }
```

Lock and Unlock

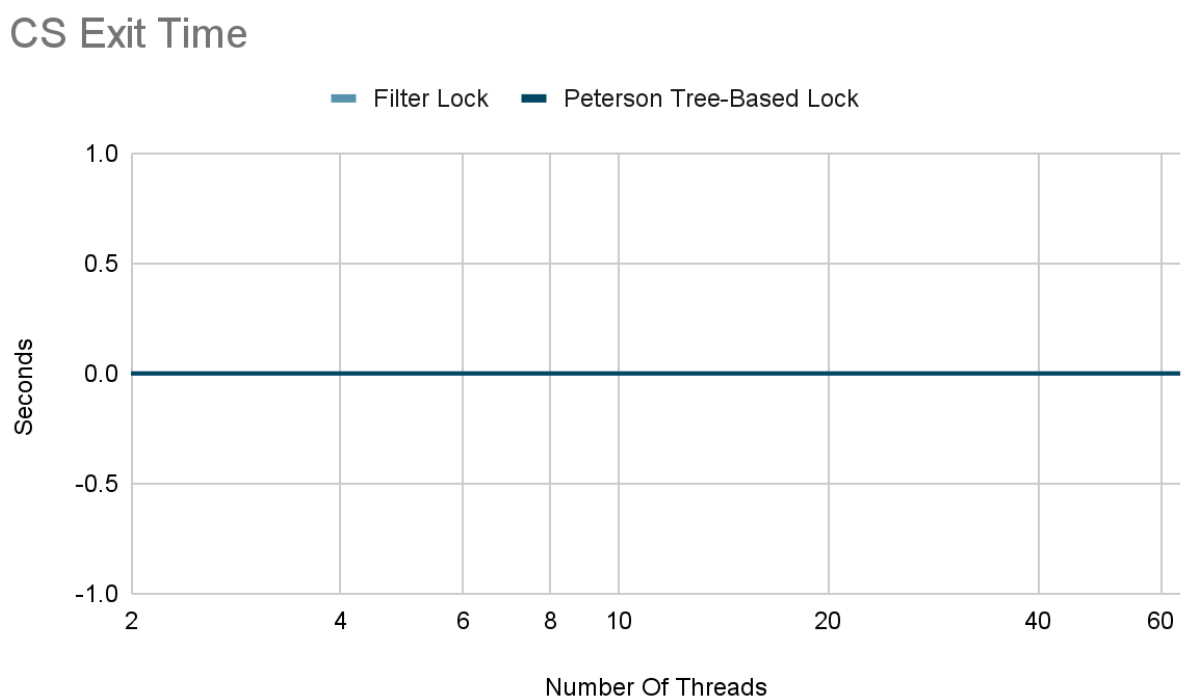
- Locking and unlocking is easy we just need to traverse from leaf to root locking/unlocking the nodes in path

```
        node *curr = (*leaf_nodes)[thread_id / 2];

        while (curr)

        {

            curr->(lock/unlock)(thread_id);

            curr = curr->parent;

        }
```

Analysis



CS Entry Time
Filter Lock
Peterson Tree-Based Lock
Seconds
10
8
6
4
2
0
2
4
6
8
10
20
40
60
Number Of Threads



CS Exit Time
Filter Lock
Peterson Tree-Based Lock
Seconds
1.0
0.5
0.0
-0.5
-1.0
2
4
6
8
10
20
40
60
Number Of Threads

CSEntry and CSExit Time= avg(avg(k CS by a thread))over n threads

When we vary the number of threads in 2^n we observe

- CSEntry Time: Increases exponentially with an exponential increase in the number of threads and both Filter Lock and Peterson Tree-based lock algorithms provide nearly similar performance with sometimes filter lock having an edge while other time tree lock gains an edge.
- CSExitTime: Is almost zero in every case as CSExit time is the difference if we dive in further the difference comes out to be in a few microseconds as its the time taken in unlocking the lock which is a very less time-consuming task in Tree-based lock if the table grows very large then it may take some time as it will have to unlock log2(n)=height nodes

```
reqExitTime = getSysTime();
cout << i << "th CS Exit Requ
" << id << " (mesg 3)";
Test.unlock();
actExitTime = getSysTime();
```

- On further trying out with 128 threads some Exit time we can see in seconds.

```
9th CS Entry At          2022-02-12 13:27:45:740 by thread 79 (mesg2)
9th CS Exit Request At   2022-02-12 13:27:45:799 by thread 79 (mesg3)
9th CS Entry At          2022-02-12 13:27:45:799 by thread 95 (mesg2)
9th CS Exit Request At   2022-02-12 13:27:45:850 by thread 95 (mesg3)
------------------------------------------------------
Average CS Entry Time for Filter Lock: 18.7852
Average CS Exit Time for Filter Lock: 0.003125
Average CS Entry Time for PTL: 20.7328
Average CS Exit Time for PTL: 0.0015625
------------------------------------------------------
./a.out  2641.35s user 23.69s system 645% cpu 6:52.94 total
(base) *[master][~/Data/Semester/Assignments/ProgAssn2-CS19B1017]$
```

References:

- The Art of Multiprocessor Programming Maurice Herlihy, Nir Shavit