Advanced Multiprocessor Programming

Project 1

Enrico Coluccia, 12005483
Aron Wussler, 11717702

1 Introduction

Mutual exclusion is perhaps the most prevalent form of coordination in multiprocessor programming. In this report we analyze some mutual exclusion algorithms that work by reading and writing shared memory, called registers lock. In particular we present the results obtained on a x86_64 AMD Epyc 64-thread Cluster provided by the Technical University of Vienna (Nebula) related to the following register locks:

- Filter Lock
- Tournament tree of 2-thread Peterson locks
- Herlihy-Shavit Bakery Lock
- Lamport Bakery Lock
- Boulangerie Lock

The results are compared with three baseline provided by the test-and-set and, test-and-test-and-set and OpenMP locks: an analysis of performance and fairness is proposed.

...

TODO: Challenge: Memory behavior. Ensure that memory (register) updates become visible in required order! Explain what happens if not (Peterson).

2 Implementation

2.1 Setup

Description of main task, CS etc.. ... ...

2.2 Filter Lock

The Filter lock is a generalized version of the Peterson lock, it creates $n-1$ "waiting rooms", or levels, that a thread must traverse before acquiring the lock. Levels satisfy two important properties:

- At least one thread trying to enter level succeeds.
- If more than one thread is trying to enter level, then at least one is blocked

The Filter lock uses an n-element integer array (called `level[]`) to indicate the highest level that a thread is trying to enter. Each thread must pass $n - 1$ levels to enter in the critical section and each level has its own victim to filter out one thread and excluding it from the next level.

The Filter lock satisfies mutual exlusion, deadlock freedom and starvation freedom properties but is not guaranteed to be fair since the threads can overtake others an arbitrary number of times (see exercise 12 from the book).

```
#include "filterLock.hpp"

FilterLock::FilterLock (int n) {
        level = new int[n];
        victim = new int[n];
        this -> n = n;
}

FilterLock::~FilterLock () {
        delete[] level;
        delete[] victim;
}

void FilterLock::lock() {
        int me = omp_get_thread_num();
        for (int i = 1; i < n; i++) {
                level[me] = i;
                victim[i] = me;
                for( int j = 0; j < n; j++) {
                        while ((j != me) && (level[j] >= i && victim[i] ==
                        ↪  me)) {}
                }
        }
}

void FilterLock::unlock() {
        int me = omp_get_thread_num();
        level[me] = 0;
}
```

2.3 Tournament tree of 2-thread Peterson locks

Since the Peterson Lock works for a 2-thread implementation, a way to generalize it is to arrange a number of 2-thread locks in a binary tree. Suppose n is a power of two. 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, from the root back to its leaf.

The Tournament tree of 2-thread Peterson locks satisfies mutual exlusion, deadlock freedom and starvation freedom properties but it is not guaranteed

to be fair since a thread can be delayed and overtaken an arbitrary number of times (see exercise 12 from the book).

```
#include "petersonLock.hpp"

PetersonLock::PetersonLock(int n){
        numOfThreads = n;
        root = new PetersonNode(NULL, numOfThreads);

        std::vector<PetersonNode*> initList;
        initList.push_back(root);

        leaves = createTree(initList);
}

PetersonLock::~PetersonLock(){
}

void PetersonLock::lock() {
        int i = omp_get_thread_num();
        PetersonNode* currentNode = getLeaf(i);

        while (currentNode != NULL) {
                currentNode->lock();
                currentNode = currentNode->parent;
        }
}

void PetersonLock::unlock() {
        int i = omp_get_thread_num();
        PetersonNode* currentNode = getLeaf(i);

        while (currentNode != NULL) {
                currentNode->unlock();
                currentNode = currentNode->parent;
        }
}

// get leaf for a specific thread id, each lock shares two threads.
PetersonNode* PetersonLock::getLeaf(int num) {
        return leaves.at(num / 2);
}

std::vector<PetersonNode*>
↪  PetersonLock::createTree(std::vector<PetersonNode*> nodes) {
        if ((int)nodes.size() == numOfThreads / 2)
                return nodes;

        std::vector<PetersonNode*> currentLeaves;

        for (PetersonNode* node : nodes) {
                node->leftChild = new PetersonNode(node, numOfThreads);
                node->rightChild = new PetersonNode(node, numOfThreads);

                currentLeaves.push_back(node->leftChild);
                currentLeaves.push_back(node->rightChild);
        }
        return createTree(currentLeaves);
}
```

2.4 Herlihy-Shavit Bakery Lock

The Herlihy-Shavit Bakery Lock is a modified version of the original Lamport's Bakery Lock implementation. In this lock, `flag[i]` is a Boolean flag indicating whether i wants to enter the critical section, and `label[i]` is an integer that indicates the thread's order when entering the "bakery", for each thread i.

Each time a thread acquires a lock, it generates a new `label[]` in two steps. First, it reads all the other threads' labels in any order. Second, it reads all the other threads' labels one after the other (this can be done in some arbitrary order) and generates a label greater by one than the maximal label it read. We call the code from the raising of the flag (Line 19) to the writing of the new `label[]` (Line 30) the doorway. The lock establishes that thread's order with respect to the other threads trying to acquire the lock. If two threads execute their doorways concurrently, they may read the same maximal label and pick the same new label. To break this symmetry, the algorithm uses a lexicographical ordering on pairs of `label[]` and thread ids at line 36, each thread read the labels one after the other in some arbitrary order until it determines that no thread with a raised flag has a lexicographically smaller label/id pair, then it enters the critical section.

The Herlihy-Shavit Bakery Lock satisfies mutual exlusion, deadlock freedom and starvation freedom properties and it is also fair (first-come-first-serve, see Lemma 2.6.2 from the Book)

```
1  #include "bakeryLock.hpp"
2
3  BakeryLock::BakeryLock (int num) {
4          flag = new bool[num];
5          label = new long long[num];
6          for (int i = 0; i < num; i ++) {
7                  flag[i] = false;
8                  label[i] = 0;
9          }
10         n = num;
11 }
12 BakeryLock::~BakeryLock () {
13         delete[] flag;
14         delete[] label;
15 }
16
17 void BakeryLock::lock () {
18         int i = omp_get_thread_num();
19         flag[i] = true;
20         long long max = label[0];
21         for (int j = 1; j < n; j ++) {
22                 if (label[j] > max) {
23                         max = label[j];
24                 }
```

```
25          }
26          if (max == LLONG_MAX) {
27                  std::cout << "ERROR: Label Value Overflow" << std::endl;
28                  exit (-1);
29          }
30          label[i] = max + 1;
31
32          for (int j = 0; j < n; j++) {
33                  if (i == j) {
34                          continue;
35                  }
36                  while (flag[j] && ((label[j] < label[i]) ||( label[i] ==
                    ↪  label[j] && j < i))) {};
37          }
38  }
39
40  void BakeryLock::unlock () {
41          flag[omp_get_thread_num()] = false;
42  }
```

2.5 Lamport Bakery Lock

This is the original version of the Lamport's Bakery Lock[1].

Each time a thread acquires a lock, it generates a new `number[]` by reading all the other threads' labels one after the other (this can be done in some arbitrary order) and generates a number greater by one than the maximal number it read. We call the code from the raising of the choosing flag (Line 20) to the writing of the new `number[]` (Line 22) the doorway section.

It establishes that thread's order with respect to the other threads trying to acquire the lock. If two threads execute their doorways concurrently, they may read the same maximal label and pick the same new label. To break this symmetry, the algorithm uses a lexicographical ordering on pairs of `number[]` as already explained in the previous lock, then it enters the critical section.

The Lamport Bakery Lock satisfies mutual exlusion, deadlock freedom and starvation freedom properties and it is also fair, since if thread A executes the doorway before thread B then B is locked out while `number[A]` is greater than 0.

```
1   #include "lamportLock.hpp"
2
3   LamportLock::LamportLock (int num) {
4           choosing = new bool[num];
5           number = new int[num];
6           for (int i = 0; i < num; i ++) {
7                   choosing[i] = false;
8                   number[i] = 0;
9           }
10          n = num;
```

[1] http://lamport.azurewebsites.net/pubs/bakery.pdf

```
11 }
12
13 LamportLock::~LamportLock () {
14         delete[] choosing;
15         delete[] number;
16 }
17
18 void LamportLock::lock () {
19         int i = omp_get_thread_num();
20         choosing[i] = true;
21         number[i] = findMax() + 1;
22         choosing[i] = false;
23
24         for (int j = 0; j < n; j++) {
25                 if (j == i)
26                         continue;
27
28                 while (choosing[j]) {}
29
30                 while (number[j] != 0 && (number[i] > number[j] ||
                   ↪  (number[i] == number[j] && i > j))) {}
31         }
32 }
33
34 void LamportLock::unlock () {
35         number[omp_get_thread_num()] = false;
36 }
37
38 int LamportLock::findMax() {
39         int m = number[0];
40         for (int i=1; i <n; ++i) {
41                 if (number[i] > m)
42                 m = number[i];
43         }
44         return m;
45 }
```

2.6 Boulangerie Lock

The Boulangerie Lock [1] is a modified version of the Lamport Bakery Lock that applies two improvements:

- Optimizing for low contention: if the thread i has obtained `number[i] = 1` then the only processes j that can ever have a better ticket are ones whose tid is smaller than i. It follows that when `number[i] = 1`, there is no need to perform the spinning section for values $j > i$. To avoid this form of unnecessary blocking, we add the control lines 36-41.

- Taking advantage of inconsistent reads: let's consider two threads i and j, and that we perform read/write operation on safe registers. As long as j is in the bakery it performs no writes on `number[j]` thus, `number[j]` is stable and all reads to it must return the same value. If i reads two different values for `number[j]` while blocking during the spinning, it has

proof that j was on the outside at least during one of these reads. Since i is the bakery section at that point, it follows that $i < j$ is true, and i can stop blocking on j and move on to test the next process.

The Boulangerie Lock satisfies mutual exlusion, deadlock freedom and starvation freedom properties and it is also fair, since if thread A executes the doorway before thread B then B is locked out while `number[A]` is greater than 0.

```
#include "boulangerieLock.hpp"

BoulangerieLock::BoulangerieLock (int numThreads) {
        choosing = new bool[numThreads];
        number = new int[numThreads];
        num = new int[numThreads];
        for (int i = 0; i < numThreads; i ++) {
                choosing[i] = false;
                number[i] = 0;
                num[i] = 0;
        }
        n = numThreads;
}

BoulangerieLock::~BoulangerieLock () {
        delete[] choosing;
        delete[] number;
        delete[] num;
}

void BoulangerieLock::lock () {
        bool tmp_c = false;
        int *prev_n = nullptr;
        int *tmp_n = nullptr;
        int limit = n;
        int i = omp_get_thread_num();

        choosing[i] = true;
        for(int j=0; j<n; j++){
                num[i]=number[j];
        }
        num[i] = findMax() + 1;
        number[i] = num[i];
        choosing[i] = false;

        if(number[i]==1){
                limit = i;
        }
        else{
                limit = n;
        }

        for (int j = 0; j < limit; j++) {
                if (j == i)
                        continue;

                do{
                        tmp_c = choosing[j];
```

```
49                  } while(tmp_c);
50
51                  tmp_n = nullptr;
52
53                  do{
54                          prev_n = tmp_n;
55                          tmp_n = &number[j];
56                  } while (*tmp_n != 0 && (num[i] > *tmp_n || (num[i] ==
                    ↪  *tmp_n && i > j)) && (tmp_n == prev_n || prev_n ==
                    ↪  nullptr));
57          }
58  }
59
60  void BoulangerieLock::unlock () {
61          int i = omp_get_thread_num();
62          num[i] = false;
63          number[i] = false;
64  }
65
66  int BoulangerieLock::findMax() {
67          int m = num[0];
68          for (int k=1; k<n; ++k) {
69                  if (num[k] > m)
70                  m = num[k];
71          }
72          return m;
73  }
```

2.7 Base Locks

For a baseline performance we consider three additional locks:

- Test-and-Set Lock: has a single flag field per lock, the thread acquire lock by changing flag from false to true and it locks on success. To unlock it resets the flag. We know from the theory (see slides) that the performance of this lock is bad, due to the high memory contention. The lock is not fair and starvation free but it is fault tolerant.
- Test-and-Test-Set Lock: we test and set only if there is a chance of success. It has a better performance than TAS but memory contention and cache deletion problems are still present. The lock is not fair and starvation free but it is fault tolerant.
- Native OpenMP locks

```
1  #include "tas.hpp"
2
3  TestAndSetLock::TestAndSetLock(int n){
4          state=false;
5  };
6
7  TestAndSetLock::~TestAndSetLock(){}
```

```
8
9  void TestAndSetLock::lock(){
10         while(state.exchange(true)){}
11 };
12
13 void TestAndSetLock::unlock(){
14         state.exchange(false);
15 };
```

```
1  #include "ttas.hpp"
2
3  TestAndTestAndSetLock::TestAndTestAndSetLock(int n){
4          state=false;
5  };
6
7  TestAndTestAndSetLock::~TestAndTestAndSetLock(){}
8
9  void TestAndTestAndSetLock::lock(){
10         while(true){
11                 while(state){}
12                 if(!state.exchange(true))
13                 return;
14         }
15 };
16
17 void TestAndTestAndSetLock::unlock(){
18         state.exchange(false);
19 };
```

```
1  #include <omp.h>
2
3  omp_lock_t mylock ;
4  omp_init_lock(&mylock ) ;
5
6  omp_set_lock(&writelock);
7
8  //critical section
9
10 omp_unset_lock(&writelock);
```

3 Fairness

In order to compare the fairness across locks and different sizes we decided to create a scale from 0 to 1 where 0 is a perfectly fair lock and 1 is a lock where only one thread has repeatedly acquired the lock. To avoid creating a biased measure, we reuse the concept of standard deviation from statistics, that we

re-scale for our purposes. In particular, we call our measure U for *Unfairness*:

$$U = \frac{\sqrt{n}}{\sum_{i=1}^{n} k_i} \operatorname{sd}(k_i) = \frac{\sqrt{n}}{\sqrt{n-1}} \frac{\sqrt{\sum_{i=1}^{n} (k_i - \bar{k})^2}}{\sum_{i=1}^{n} k_i}$$

where n is the number of threads and k_i is the number of locks acquired by thread i.

If we set all k_i to be 0 but one (k_j) we obtain:

$$\begin{aligned}
U &= \frac{\sqrt{n}}{\sqrt{n-1}} \frac{\sqrt{(n-1)(\bar{k})^2 + (k_j - \bar{k})^2}}{k_j} \\
&= \frac{\sqrt{n}}{\sqrt{n-1}} \frac{\sqrt{(n-1)(k_j/n)^2 + (k_j - k_j/n)^2}}{k_j} \\
&= \frac{1}{k_j} \sqrt{\frac{n}{n-1} \left[(n-1)\frac{k_j^2}{n^2} + \frac{k_j^2}{n^2}(n-1)^2 \right]} \\
&= \frac{1}{k_j} \sqrt{n \left[\frac{k_j^2}{n^2} + \frac{k_j^2}{n^2}(n-1) \right]} \\
&= \frac{1}{k_j} \sqrt{n^2 \frac{k_j^2}{n^2}} = 1.
\end{aligned}$$

When all threads acquire the same amount of locks then $k_i = \bar{k} \forall i$ and by definition of standard deviation the result is zero.

Using this measure we can now meaningfully compare the (un)fairness across various implementations and with different sizes in figure 1.

We notice immediately that the TAS, TTAS, and OpenMP locks are highly unfair. In the collected data most of the executions are characterised by one or two threads mostly getting the lock. We can then exclude the from the plot to distinguish the custom locks in figure 2.

Between these, we have that the Bakery and derivates behave very similarly, with an extemely low unfairness, while the Peterson and Filter lock present some peaks. In particular we can observe a positive correlation between unfairness and size of the cluster for the Peterson lock.

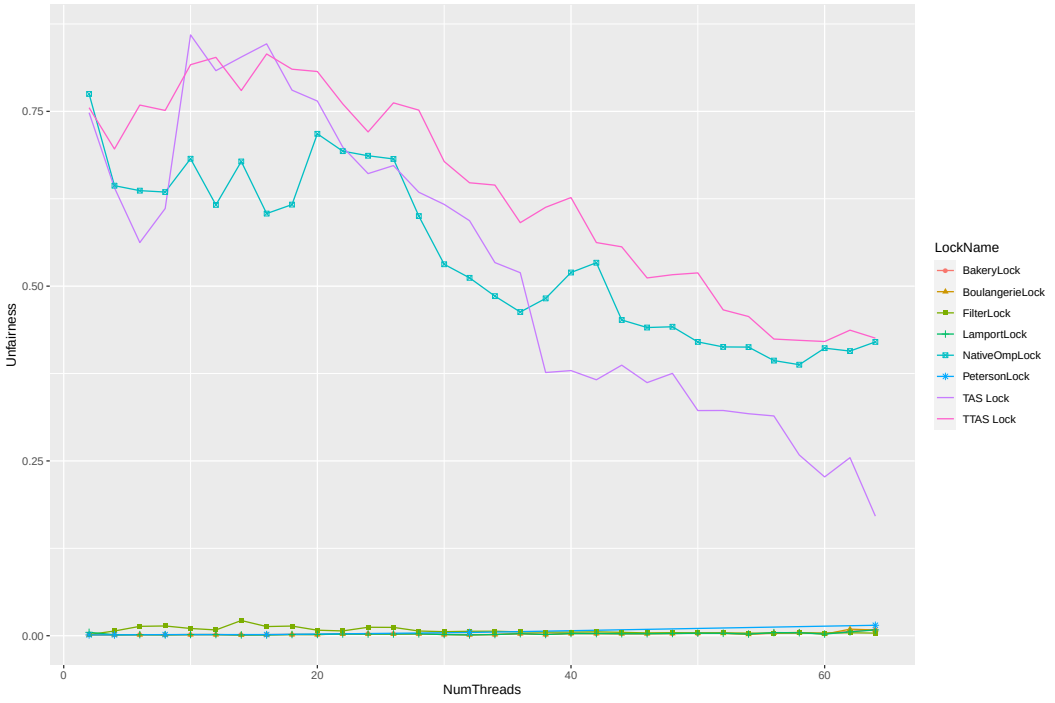

Figure 1: Fairness for all implemented locks

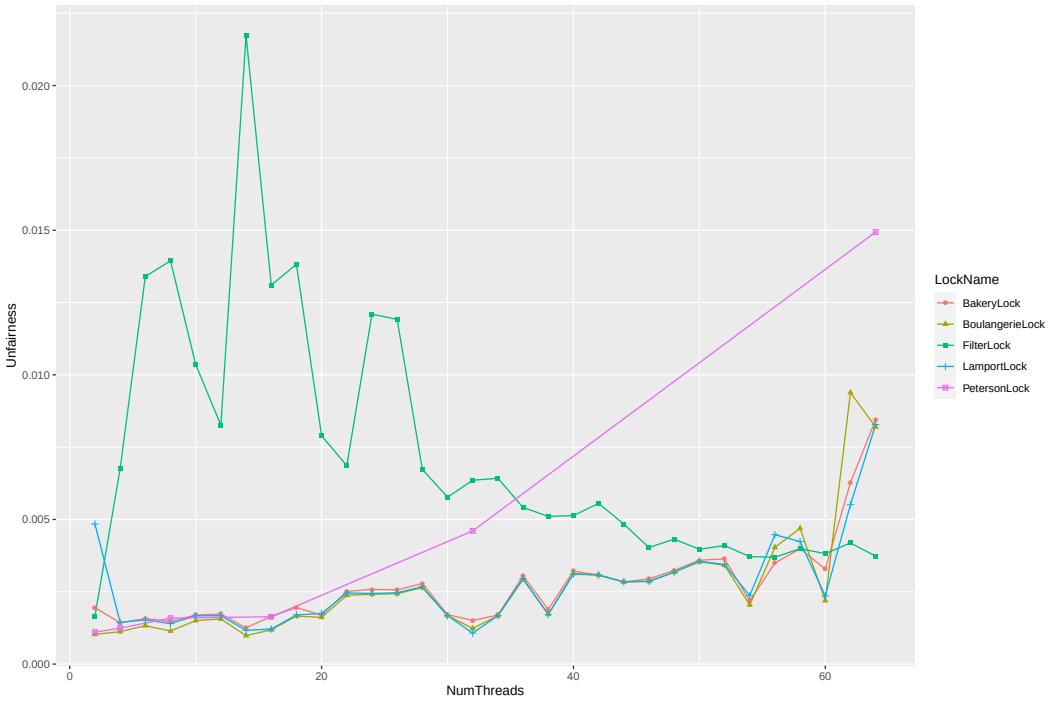

Figure 2: Fairness for custom locks

4 Performance

References

[1] Yoram Moses and Katia Patkin. “Mutual exclusion as a matter of priority”. In: *Theoretical Computer Science* 751 (2018). Structural Information and Communication Complexity, pp. 46–60. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2016.12.015. URL: https://www.sciencedirect.com/science/article/pii/S0304397516307435.