ENGI 7894/9869

Spin Locks and Practical Lock

Presented by

Md Monjur Ul Hasan

Previous Class

Previous Class

- $Write_0(flag[0] = true)$
  - $Write_{\mathbf{0}}$ => Thread 0 is writing
  - Variable True[0] is updating

- $CS_{\mathbf{1}} \Longrightarrow$ Critical Section ***1*** *executed*
- $CS_1 \rightarrow CS_2 \Longrightarrow$ Critical Section CS_1 precedes event CS_2

- CS_1 and CS_2 occurring concurrently
  - $CS_1 \nrightarrow CS_2 \Longrightarrow$ Critical Section CS_1 did not precedes event CS_2
  - $CS_2 \nrightarrow CS_1 \Longrightarrow$ Critical Section CS_1 did not precedes event CS_2

Lock 1 : Mutual Exclusion…

Thread 0

```
class LockOne implements iLock
{
    public void lock()
    {
        flag[0] = true;
        while (flag[1]) {} // wait
    }

    public void unlock()
    {
        flag[0] = false;
    }
}
```

Thread 1

```
class LockOne implements iLock
{
    public void lock()
    {
        flag[1] = true;
        while (flag[0]) {} // wait
    }

    public void unlock()
    {
        flag[1] = false;
    }
}
```

$$Write_0(flag[0] = true) \rightarrow Read_0(flag[1] == false) \rightarrow CS_0$$
$$Write_1(flag[1] = true) \rightarrow Read_1(flag[0] == false) \rightarrow CS_1$$

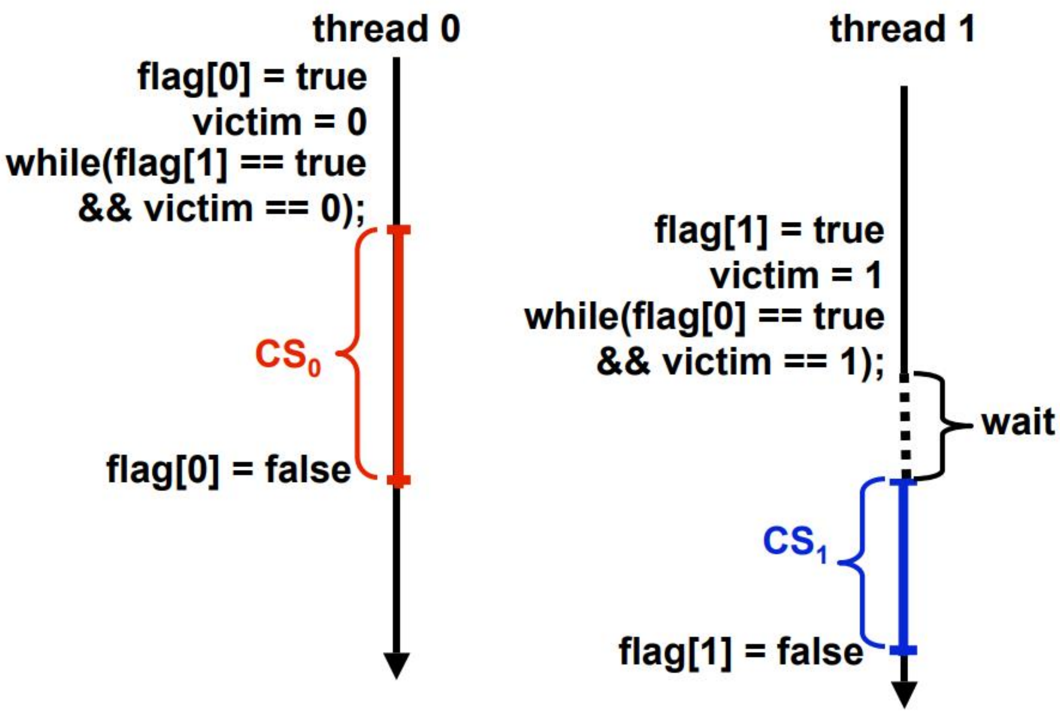
Contradiction: $\boldsymbol{CS_0 \nrightarrow CS_1 \text{and } CS_1 \nrightarrow CS_0}$

$$\boldsymbol{Write_0\ (flag[0] = true) \rightarrow Read_0(flag[1] == false) \rightarrow CS_0}$$
$$\boldsymbol{\rightarrow Write_1(flag[1] = true) \rightarrow Read_1(flag[0] == false) \rightarrow CS_1}$$

$$\boldsymbol{Write_1\ (flag[1] = true) \rightarrow Read_1(flag[0] == false) \rightarrow CS_1}$$
$$\boldsymbol{\rightarrow Write_0(flag[0] = true) \rightarrow Read_0(flag[1] == false) \rightarrow CS_0}$$

Peterson Lock

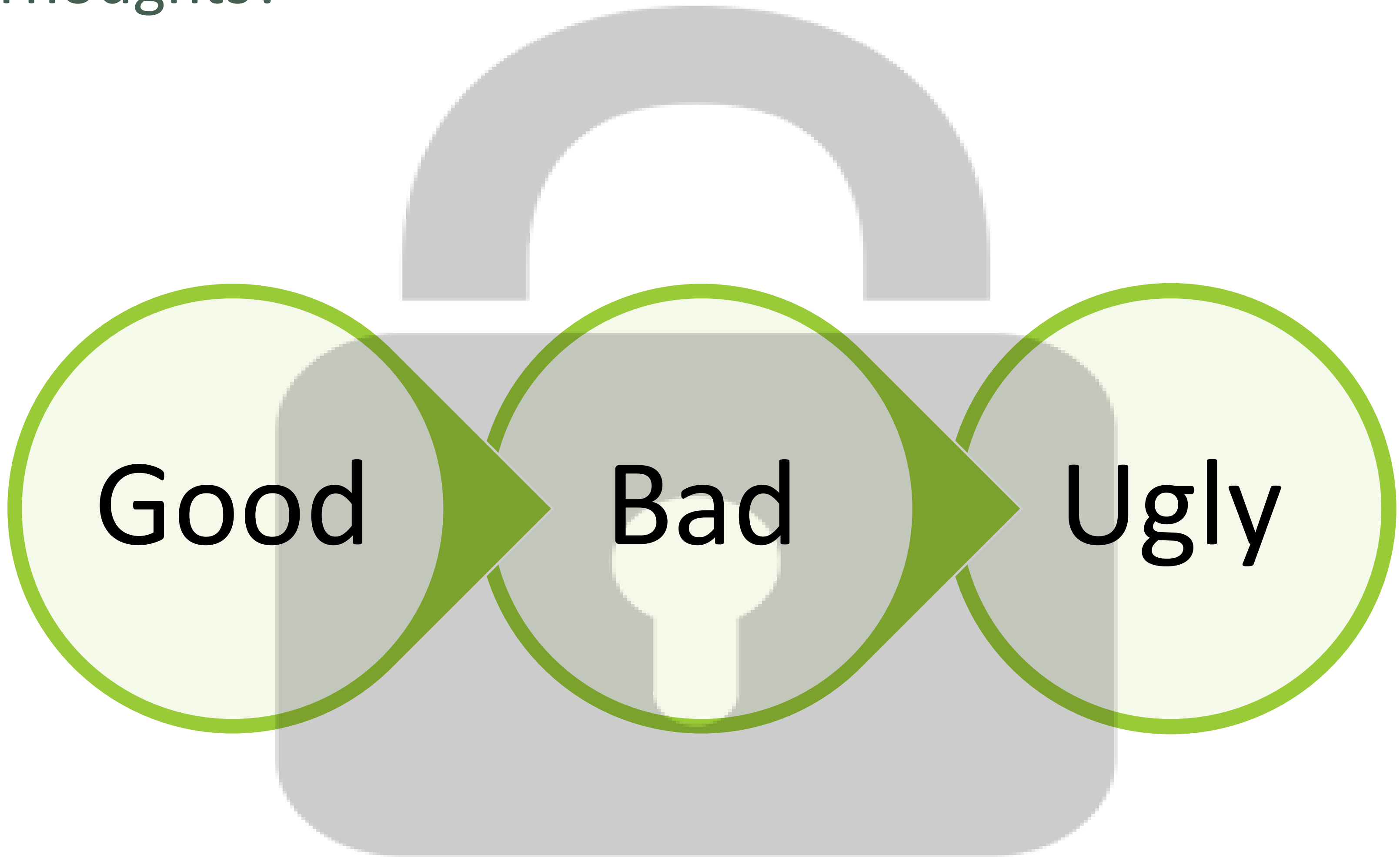
```
class Peterson implements iLock
{
      // thread-local index, 0 or 1
      private boolean[] flag = new boolean[2];
      private int victim;
      public void lock()
     {
               int i = ThreadID.get();
               int j = 1 - i;
               flag[i] = true;
               victim = i;
               while (flag[j] && victim == i) {};
     }
     public void unlock()
     {
               int i = ThreadID.get();
               flag[i] = false;
     }
 }
```



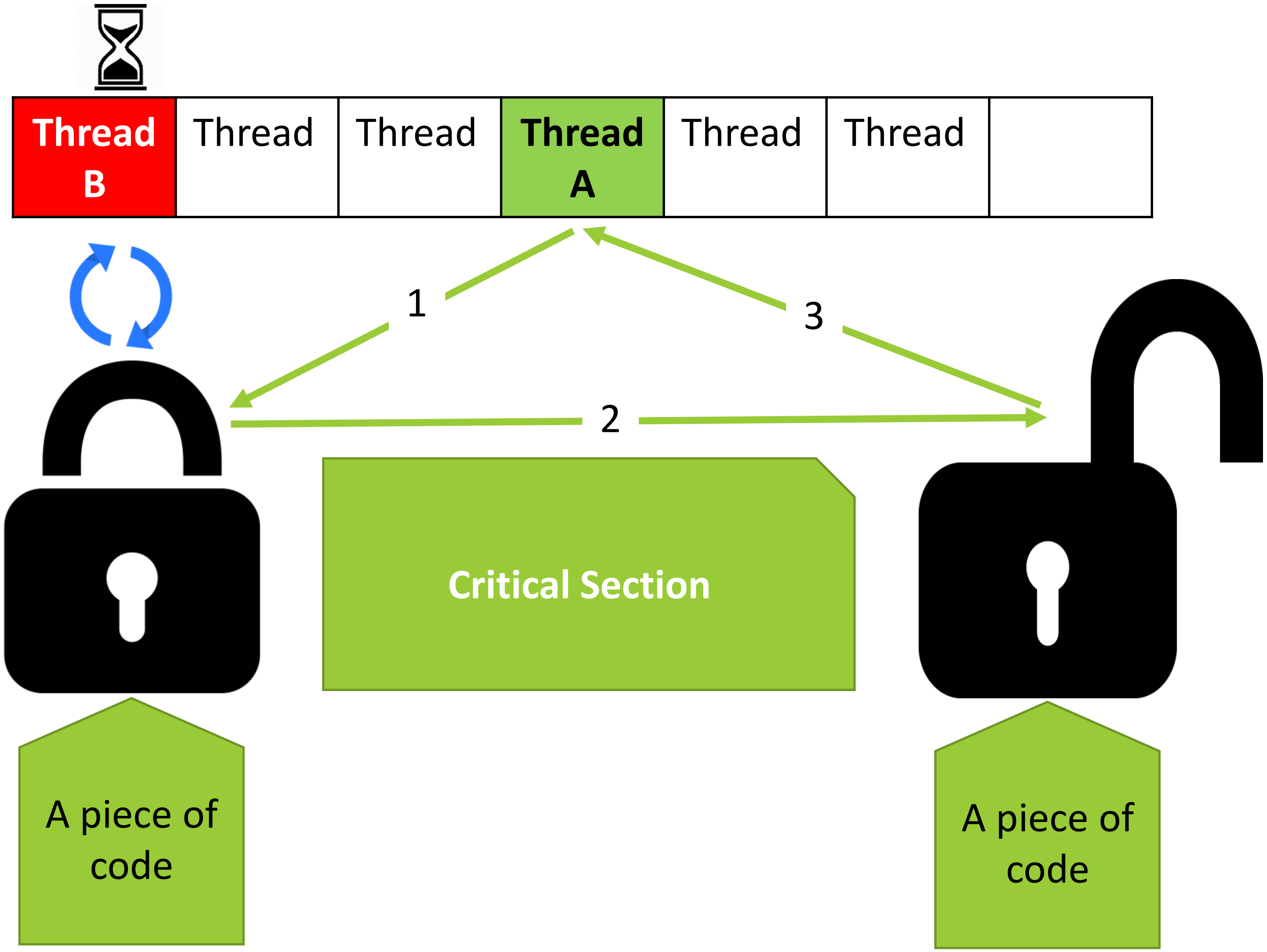
DIY: Proof the mutual exclusion?

Question or Comments From Previous Class?

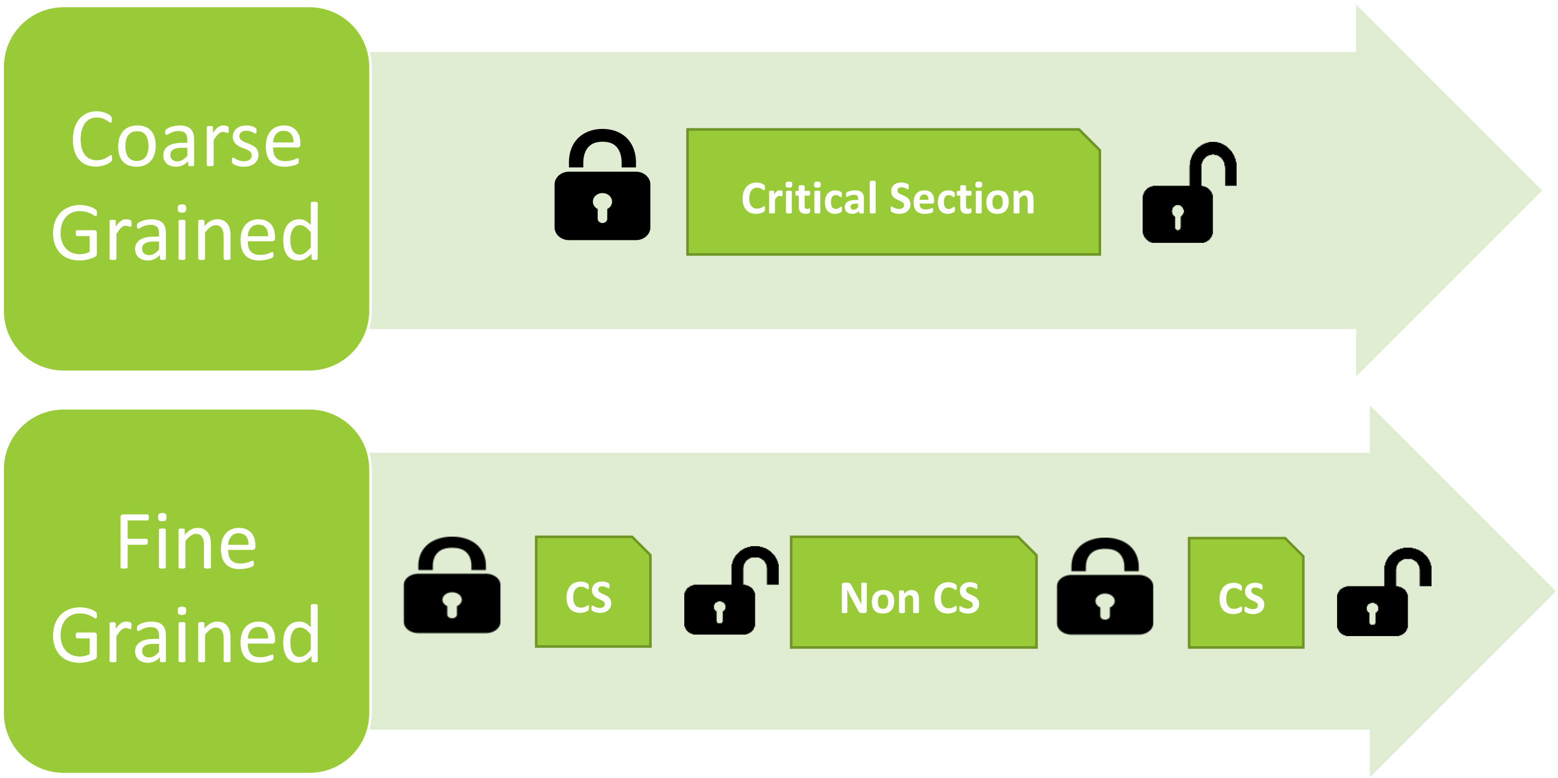
Thoughts?



Previous Class



Locking Strategy

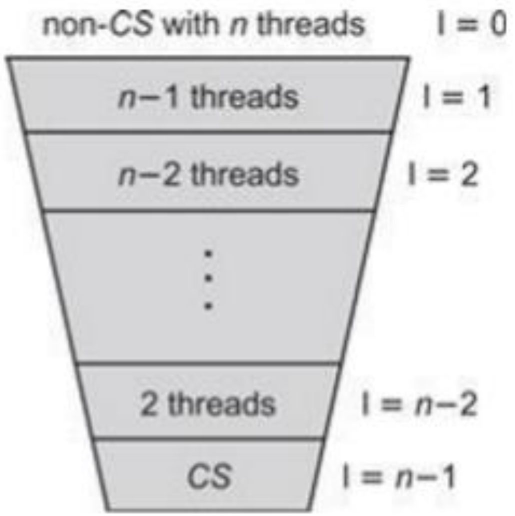


$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

Lock for ‘n’ threads

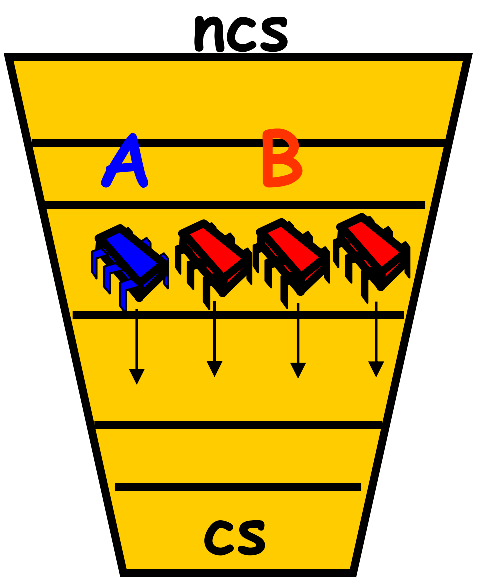
Filter Lock

- Generalization of Paterson Lock
- For N threads, N level of waiting queue (Spinning)
- The higher the level, the fewer threads are spinning on that level



Filter Lock (Code)

```
class Filter implements Lock
{
    int[] level;
    int[] victim;
    public Filter(int n)
    {
        level = new int[n];
        victim = new int[n]; // use 1..n
        for (int i = 0; i < n; i++)
        {
            level[i] = 0;
        }
    }
    public void lock()
    {
        int me = ThreadID.get();
        for (int i = 1; i < n; i++)
        { //attempt level i
            level[me] = i;
            victim[i] = me; // spin while conflicts exist
            while ((there exists k != me) (level[k] >= i && victim[i] == me)) {};
        }
    }
    public void unlock()
    {
        int me = ThreadID.get();
        level[me] = 0;
    }
}
```



- **while** (
  - (there exists k != me)
  -
    - (level[k] >= i
      - &&
    - victim[i] == me)

)

- Anyone other than me
  - Is waiting in higher level than my level
    - And
  - I am the victim for waiting

If no one is in higher level, then I will proceed
Or
If I am not victim, then I will proceed

Filter Lock: Mutual Exclusion

- ***Proof by Induction (Method)***
  - *Base case (Usually n = 0, 1, or n)*
  - *Inductive case:*
    - *Hypothesis: True for n = k*
    - *Need to proof*
      - *True for n = k+1 (or k-1) when the above are true.*

```
class Filter implements Lock
{
    int[] level;
    int[] victim;
    public Filter(int n)
    {
        level = new int[n];
        victim = new int[n]; // use 1..n
        for (int i = 0; i < n; i++) {
            level[i] = 0;
        }
    }
    public void lock() {
        int me = ThreadID.get();
        for (int i = 1; i < n; i++)  { //attempt level i
            level[me] = i;
            victim[i] = me; // spin while conflicts exist
            while ((there exists k != me) (level[k] >= i && victim[i] == me)) {};
        }
    }
    public void unlock()  {
        int me = ThreadID.get();
        level[me] = 0;
    }
}
```

- **MutEx Proof by Induction**
  - **Base Case:** Level 0 trivial
  - **Inductive Case:**
    - Hypothesis: At level j there is n-j threads.
    - Need to proof, at Level j-1, there are at most n-j-1 threads

Filter Lock: Mutual Exclusion

Proof of Inductive Case

*show that **at least** one thread **will not make it** to next level.*

- **Proof by contradiction**: (All threads make it to the next level)
- At Level J:
  - Let A be the last thread entered at level j
    - A write to victim[j] = A
  - Let B is another thread at Level j. Then:
  - $write_B(level[B] = j) \rightarrow write_B(victim[j] = B) \rightarrow write_A(victim[j] = A) \rightarrow read_A(level[B])$
  - **Contradiction**

```
class Filter implements Lock
{
      .....
      public void lock() {
            int me = ThreadID.get();
            for (int i = 1; i < n; i++)  { //attempt level i
                  level[me] = i;
                  victim[i] = me; // spin while conflicts exist
                  while ((there exists k != me) (level[k] >= i && victim[i] == me)) {};
            }
      }
       ........
}
```

- Because there are at most n-j threads at level j, and the CS is at level n-1, there is at most 1 thread in the CS. So we have mutual exclusion!

Filter Lock: Freedom of Starvation

- Proof by Induction
  - Level 0 trivial
  - Level j-1, there are at most n-j-1 thread (Inductive hypothesis)
  - Level J:
    - Need to show that **at least** one thread **will make it** to next level.
    - Proof by contradiction: (No thread will able to make it to the next level)
      - Let A be the last thread entered at level j
      - Let B is another thread at Level j. Then:

$$write_B(level[B] = j) \rightarrow write_B(victim[j] = B) \rightarrow write_A(victim[j] = A) \rightarrow read_A(level[B])$$

- **Contradiction**

```
class Filter implements Lock
{
      int[] level;
      int[] victim;
      public Filter(int n)
      {
            level = new int[n];
            victim = new int[n]; // use 1..n
            for (int i = 0; i < n; i++) {
                  level[i] = 0;
            }
      }
      public void lock() {
            int me = ThreadID.get();
            for (int i = 1; i < n; i++)  { //attempt level i
                  level[me] = i;
                  victim[i] = me; // spin while conflicts exist
                  while ((there exists k != me) (level[k] >= i && victim[i] == me)) {};
            }
      }
      public void unlock()  {
            int me = ThreadID.get();
            level[me] = 0;
      }
}
```

What was the difference between the two proofs?

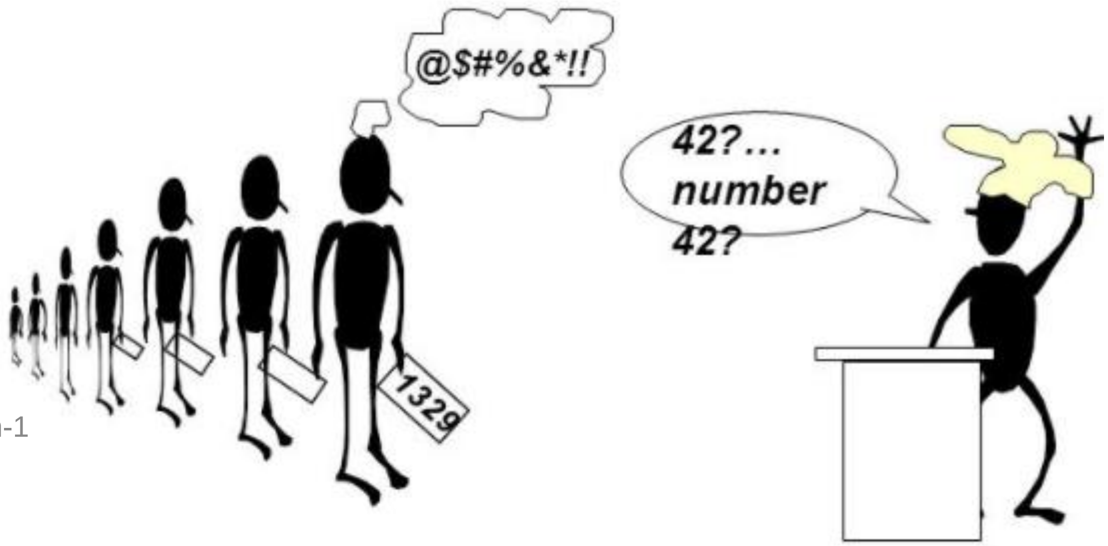
Mutex: ***at least*** *one thread* ***will not make it*** *to next level*

Freedom of Starvation: ***at least*** *one thread* ***will make it*** *to next level.*

Question or Comments?

Lamport's Bakery Algorithm

(*MUN Wellness Center*)




```
class Bakery implements iLock
 {
        boolean[] flag;
        Label[] label;
        public Bakery(int n)
        {
                flag = new int[n];
                label = new Label[n]; // use 1..n-1
                for (int i = 0; i < n; i++)
                {
                        flag[i] = false;
                        label[i] = 0;
                }
        }
        public void lock()
        {
                int i = ThreadID.get();
                flag[i] = true;
                label[i] = max(label[0], ..., label[n-1]) + 1;
                while ((there exists k != i) (flag[k] && (label[k],k) << (label[i],i))) {};
        }
        public void unlock()
        {
                flag[ThreadID.get()] = false;
        }
 }
```

Bakery algorithm (Explanation)

- flag[A] indicates whether A wants to enter CS, while
- label[A] indicates thread's relative order when entering bakery.

- Each thread generates unique label greater **(or may be equal?)** than all other labels.
- Lexicographical ordering is used to break ties in labels
  - Tie is broken by thread ID.

- Clearly deadlock free, because some thread will have the lowest (label, thread ID) pair.
- Can also show that first-come first-served property holds => Fair lock

- Therefore, also starvation free. **(Why?)**

Lamport's Bakery Algorithm: Property

Freedom of Deadlock

Freedom of Starvation

Mutual Exclusion

- **Proof by contradiction**
- *flag[k] && (label[A],A) << (label[B],B)*
- Lets assume: (label[A],A) << (label[B],B) is **true**
  - i.e., thread A comes before thread B (Wolog)
- For both thread A and thread B to be in CS:
  - flag[A] is read false in Thread B
    - *It means thread A is not in CS*
  - flag[B] is read false for Thread A
    - *It means thread B is not in CS*
- **Contradiction**

```
flag[i] = true;
label[i] = max(label[0], ..., label[n-1]) + 1;
while ((there exists k != i) (flag[k] && (label[k],k) << (label[i],i))) {};
```

Question or Comments?

Spin Locks Issue

Algorithm

- Waste Clock Cycle
- Memory Operation are time consuming
- Memory Congestion
- Slower

System

- Buffer (Delayed) Write Process
- Reorder Instruction (compiler)
- Memory Barrier
- Volatile Memory (No Cache)

Test-and-Set locks

```
public class TASLock implements iLock
{
        AtomicBoolean state = new AtomicBoolean(false);

        public void lock()
        {
                while (state.getAndSet(true)) {}
        }

        public void unlock()
        {
                state.set(false);
        }
}
```

- Test-and-set
  - A Boolean value
  - Set true and return old value
- In Java, testAndSet() is equivalent to atomic getAndSet(true) function call.
- Less bus traffic with cache structure
- Fairness not guaranteed.

Test-and-test-and-Set locks

```
public class TTASLock implements iLock
{
		AtomicBoolean state = new AtomicBoolean(false);

		public void lock()
		{
		     while(true) {
		         while (state.get()) {};
		          if (!state.getAndSet(true))
		                   return;
		      }
		}

		public void unlock()
		{
				state.set(false);
		}
}
```

- No write operation early stage of the locking
  - More efficient on real multiprocessors, since we can rely on the cache value
- Faster unlocking

Topics

- Spin Locks (N Threads)
  - Filter locks
  - Lamport's Bakery Algorithm
- Practical Locks
  - Test and Set Lock
  - Test and Test and Set Lock

Thank you for your attention

Any Questions?

https://www.youtube.com/watch?v=gqw2VS_vKIA