Mutual Exclusion

Thomas Schwarz, SJ

Locks in C++

```
#include <mutex>

class Y{
private:
    int some_detail;
    mutable std::mutex m;
    int get_detail() const {
        std::lock_guard<std::mutex> lock_a(m);
        return some_detail;
    }

public:
    Y(int sd):some_detail(sd){}
    friend bool operator==(Y const& lhs, Y const& rhs) {
       if(&lhs==&rhs)
          return true;
       int const lhs_value=lhs.get_detail();
       int const rhs_value=rhs.get_detail();
       return lhs_value==rhs_value;
    }
};
```

Protect with a lock
Resource Allocation is Initialization

C++: RAII

- Programming with locks mistakes:
    - Forgetting to lock
    - Forgetting to unlock
        - Especially if an error is thrown
- **R**esource **A**llocation **i**s **I**nitialization
    - Locks are unlocked when the code leaves the scope because the object no longer exists

Locks in Java

- Make sure to unlock by using unlock

```
mutex.lock();
try {
   // … code goes here …
}  finally  {
   // … restore invariants …
   mutex.unlock();
}
```

Threads

- A sequence of indivisible events
  - A: $a_1, a_2, \ldots, a_n, \ldots$
  - All events are ordered:
    - $a_1 < a_2 < a_3 < \ldots < a_n$
- Events of two or more threads are interleaved
  - Some depend on others
- Event Examples:
  - Assign to shared variable; Assign to local variable; Invoke method; Return from method …

Threads

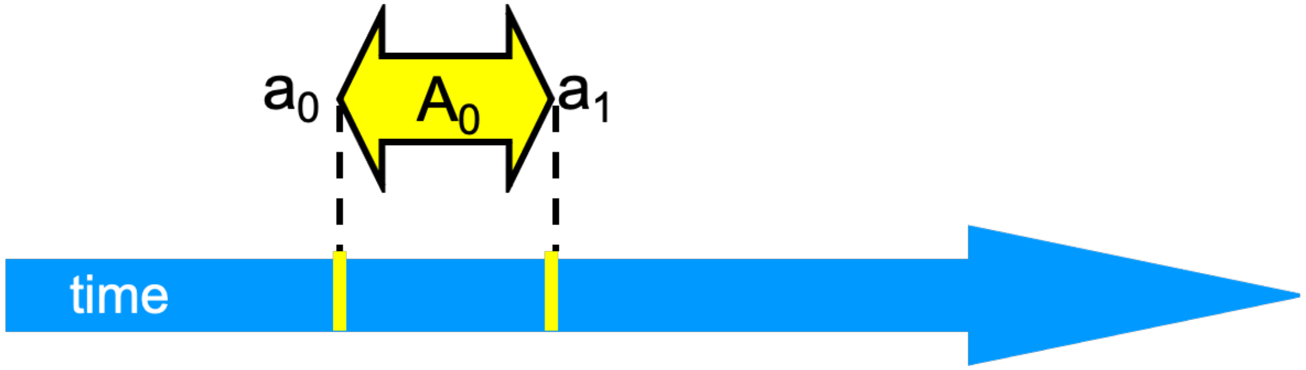
- Thread state:
  - Program counter
  - Local variables
- System state
  - Object fields (shared variables)
  - Union of thread states

Time

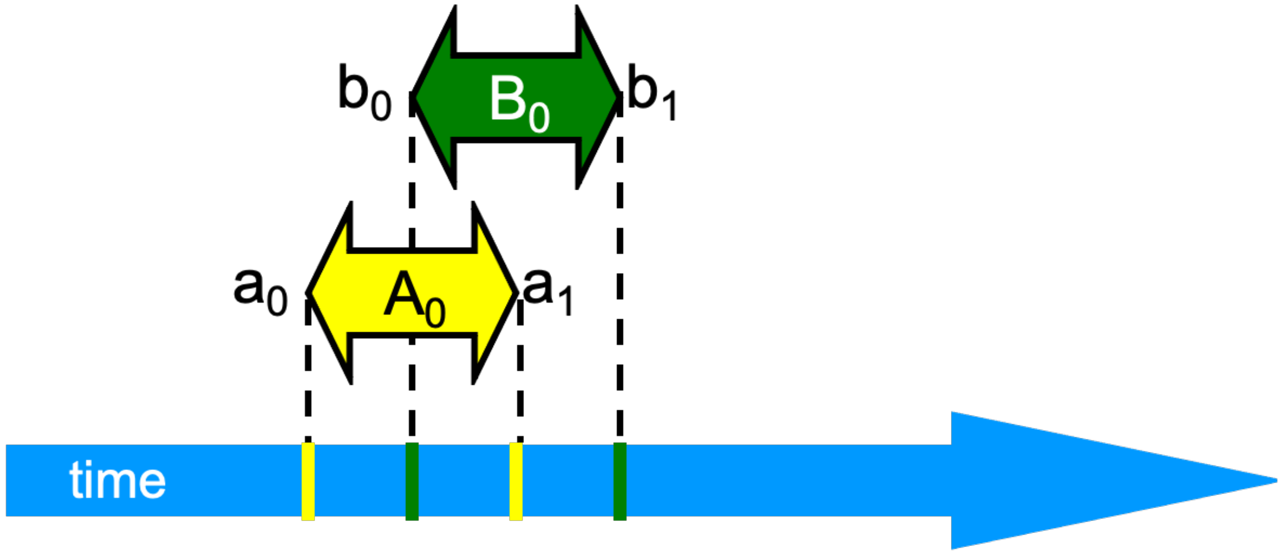
- Time: Shared by threads,
    - but threads do not have a common clock
- Events: Instantaneous
    - Two events never happen at the same time
        - Events in a thread are totally ordered

Intervals

- An interval $A_0 = (a_0, a_1)$ is the time between events a_0 and a_1

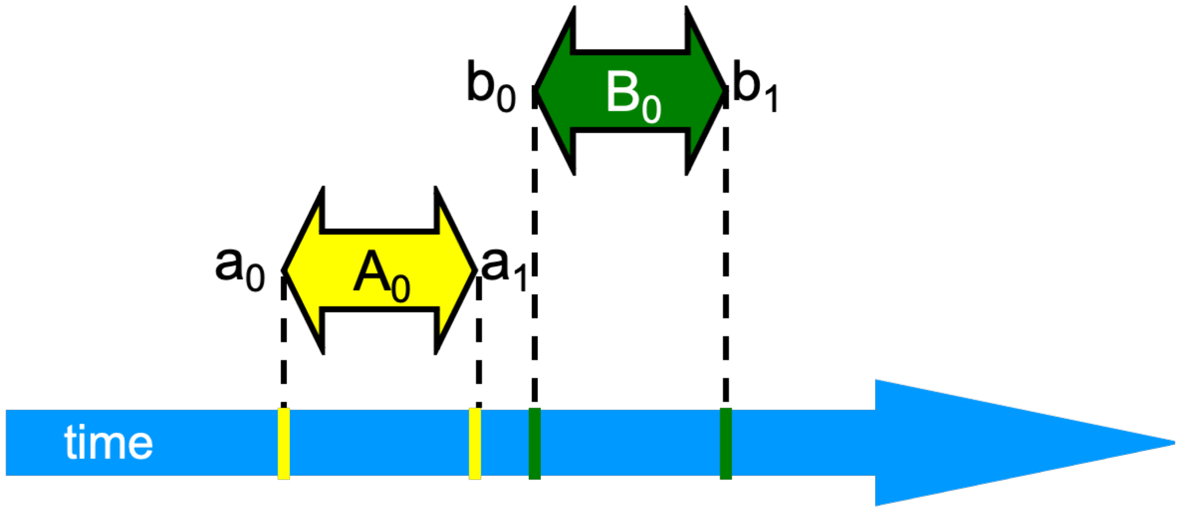


- Intervals might overlap



Intervals

- Intervals might be disjoint
  - In which case we can define a precedence



  - Interval A precedes interval B:
    - a.k.a: A happens before B
    - All events in A are before all events in B

Critical Section

- Block of code that can be executed only by one thread
- Mutual exclusion:
  - Let CS_i^k be thread i's k^{th} execution of critical section
  - CS_j^l be thread j's l^{th} execution of the critical section
  - **THEN** either $\text{CS}_i^k < \text{CS}_j^l$ or $\text{CS}_l^j < \text{CS}_i^k$

Goals

- Mutual exclusion
    - Either $CS_A^k < CS_B^l$ or $CS_B^l < CS_A^j$
- Freedom from deadlock
    - If some thread attempts to acquire the lock then some thread will acquire the lock
- Freedom from starvation (lockout freedom)
    - Every thread that attempts to acquire the lock eventually succeeds

Mutual Exclusion in Software only

Warning

- These algorithms are not used in practice
  - Efficient locks need hardware support, more than just registers
  - We are using only registers

- In the sixties, several wrong algorithms were published claiming to do mutual exclusion

Lock Algorithms for Two Threads

```
class LockOne implements Lock {
private boolean[] flag = new boolean[2];
public void lock() {
  int i = ThreadID.get();
  int j = i-1;
  flag[i] = true;
  while (flag[j]) {}
}
public void unlock() {
  int i = ThreadID.get();
  flag[i] = false;
}
```

Lock Algorithms for Two Threads

```
class LockOne implements Lock {
private boolean[] flag = new boolean[2];
public void lock() {
  int i = ThreadID.get();
  int j = i-1;
  flag[i] = true;
  while (flag[j]) {}
}
public void unlock() {
  int i = ThreadID.get();
  flag[i] = false;
}
```

i is current thread
j is the other thread

Lock Algorithms for Two Threads

- Assume that we have both threads 0 and 1 in the critical section

1. $w_0(\text{flag}[0] = 1) < r_0(\text{flag}[1] = 0) < \text{CS}_0$
2. $w_1(\text{flag}[1] = 1) < r_1(\text{flag}[0] = 0) < CS_1$

- Since both are still in their critical section, no flag is set to false.
  - Therefore:

3. $r_0(\text{flag}[1] = 0) < w_1(\text{flag}[1] = 1)$
4. $r_1(\text{flag}[0] = 0) < w_0(\text{flag}[0] = 1)$

```
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
}
public void unlock() {
  flag[i] = false;
}
```

Lock Algorithms for Two Threads

- This leads to a vicious cycle
  - $w_0(\text{flag}[0] = 1)$
  - $< r_0(\text{flag}[1] = 0)$ by (1)
  - $< w_1(\text{flag}[1] = 1)$ by (3)
  - $< r_1(\text{flag}[0] = 0)$ by (2)
  - $< w_0(\text{flag}[0] = 1)$ by (4)

Lock Algorithms for Two Threads

- Algorithm can easily dead-lock

```
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
}
public void unlock() {
  flag[i] = false;
}
```

```
flag[0]=1
flag[1]=1
;
;
;
;
;
;
;
```

Lock Algorithms for Two Threads

- A simpler lock algorithm that only works if both threads are active

```
class LockTwo implements Lock {
   private int victim;
   public void lock() {
      int i = ThreadID.get();
      victim = i;
      while (victim==i) {}
   }
   public void unlock()  {}
}
```

Lock Algorithms for Two Threads

- Prove mutual exclusion

```
public void lock() {
    victim = i;
    while (victim==i) {}
 }
 public void unlock()  {}
```

Lock Algorithms for Two Threads

- Thread *i* is in the critical region:
  - $w_0(v = 0) < r_0(v = 1)$ $w_1(v = 1) < r_1(v = 0)$
- Thread *i* is the only one that set *v* to *i* and that only once
  - Assume $w_1(v = 1) < w_0(v = 0)$:
    - Then $r_0(v = 1)$ never happens
  - Assume $w_0(v = 0) < w_1(v = 1)$:
    - Then $r_1(v = 0)$ never happens
- Contradiction!

```
public void lock() {
    victim = i;
    while (victim==i) {}
 }
 public void unlock()  {}
```

Lock Algorithms for Two Threads

- Peterson's algorithm
  - Combines both

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Lock Algorithms for Two Threads

- Peterson's algorithm
    - Combines both

express interest to enter critical section

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Lock Algorithms for Two Threads

- Peterson's algorithm
  - Combines both

Defer to other

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Lock Algorithms for Two Threads

- Peterson's algorithm
  - Combines both

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

give priority to other

Lock Algorithms for Two Threads

- Peterson's algorithm
  - Combines both

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

no longer interested

Lock Algorithms for Two Threads

- Mutual exclusion: Assume both 0 and 1 are in the critical section
- Observe
  - $w_1(f[1] = 1) < w_1(v = 1)$

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i)
{};
}
public void unlock() {
 flag[i] = false;
}
```

Lock Algorithms for Two Threads

- $w_0(f[0] = 1) < w_0(v = 0)$

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i)
{};
}
public void unlock() {
 flag[i] = false;
}
```

Lock Algorithms for Two Threads

- $w_0(v = 0) < r_0(f[1] = 1) < r_0(v = 1)$

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i)
{};
}
public void unlock() {
 flag[i] = false;
}
```

Lock Algorithms for Two Threads

- Assume 0 is the last thread to write to victim:
  - $w_1(v = 1) < w_0(v = 0)$

Lock Algorithms for Two Threads

- Combine observations:
  - $w_1(f[1] = 1)$
  - $< w_1(v = 1)$
  - $< w_0(v = 0)$
  - $< r_0(f[1] = 1)$
  - $< r_0(v = 1)$
  - So thread 0 is still spinning and NOT in the critical section

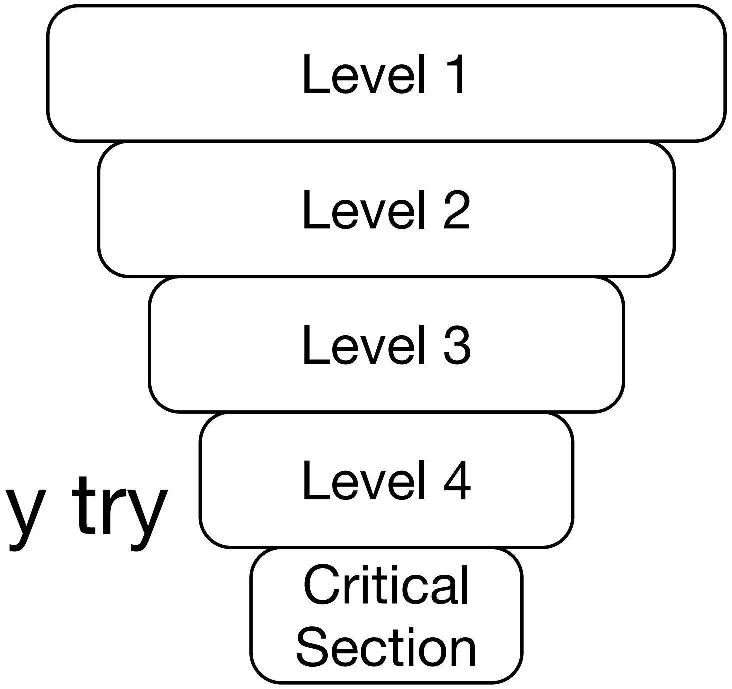
```
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i){};
}
public void unlock() {
 flag[i] = false;
}
```

Lock Algorithms for Two Threads

- Deadlock free:
  - Assume that the two threads are blocked
    - Thread 0 is blocked if
      - $r_0(f[1] = 1)\ \&\ \&\ r_0(v = 0)$
    - Thread 1 is blocked if
      - $r_1(f[0] = 1)\ \&\ \&\ r_1(v = 1)$
- If both flags are asserted then only one is blocked
- If only one flag is asserted, then the other can proceed

Lock Algorithms for *n* Threads

- Filter Algorithm
  - $n - 1$ "waiting rooms" called levels
  - At each level:
    - at least one thread enters level
    - at least one thread is blocked is many try
  - Only one thread makes it to the critical section



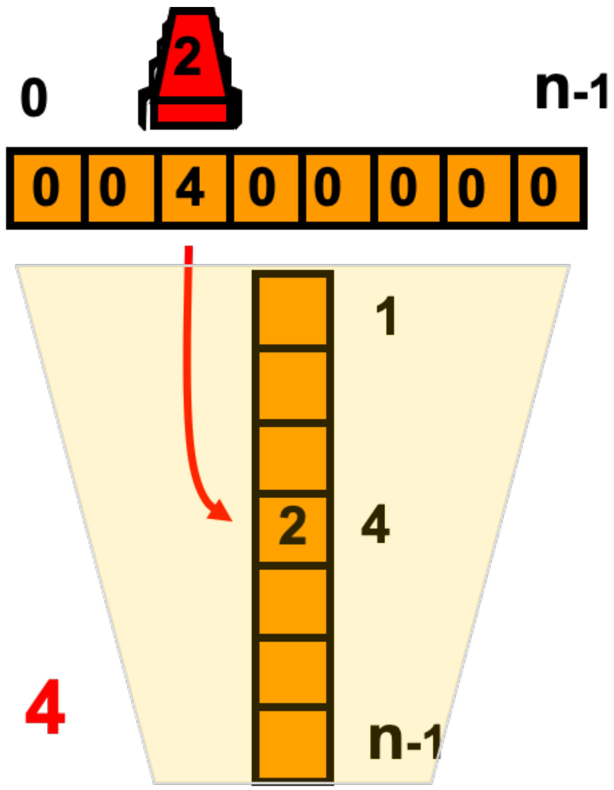
Filter Lock

- Peterson lock uses a two-element boolean flag array
  - indicates whether a thread is trying to enter the critical section
- Filter lock generalizes with an *n* element integer level[ ] array
- Each level has a distinct victim field to filter out one thread
- Initially, a thread is at level 0.
  - Thread A is at level *j* if it has completed the waiting loop with level[A]>=j

Filter Lock

```
class Filter implements Lock {
   int[] level;  // level[i] for thread i
   int[] victim; // victim[L] for level L

  public Filter(int n) {
  level  = new int[n];
  victim = new int[n];
  for (int i = 1; i < n; i++) {
      level[i] = 0;
  }}
…
}
```



Thread 2 at level 4

Filter Lock

```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∄ k != i: level[k] >= L) &&
             victim[L] == i ) {};
    }}
  public void unlock() {
    level[i] = 0;
  }}
```

Filter Lock

```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∄ k != i: level[k] >= L) &&
             victim[L] == i ) {};
    }}
  public void unlock() {
    level[i] = 0;
  }}
```

One level at a time

Filter Lock

```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∄ k != i: level[k] >= L) &&
                 victim[L] == i ) {};
    }}
  public void unlock() {
    level[i] = 0;
  }}
```

Announce intention to enter L

Filter Lock

```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∄ k != i: level[k] >= L) &&
             victim[L] == i ) {};
    }}
  public void unlock() {
    level[i] = 0;
  }}
```

Give priority to anyone but me

Filter Lock

```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∄ k != i: level[k] >= L) &&
             victim[L] == i ) {};
    }}
  public void unlock() {
    level[i] = 0;
  }}
```

Wait as long as someone else is at same of higher level and I am the designated victim

Filter Lock

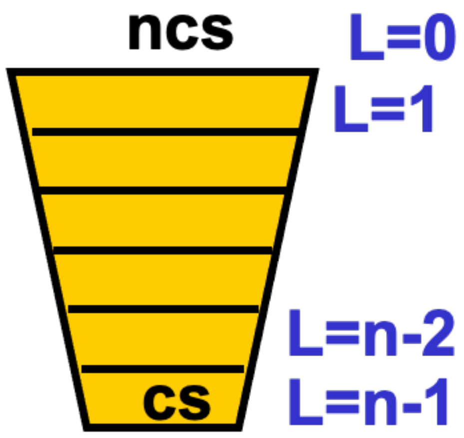
```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∄ k != i: level[k] >= L) &&
             victim[L] == i ) {};
    }}
  public void unlock() {
    level[i] = 0;
  }}
```

Thread enters level L when it completes the loop

Filter Lock

- Claim:
  - Start at level *L*=0
  - At most *n*-*L* threads enter level *L*
  - Mutual exclusion at level *L* = *n*-1



Filter Lock

- Induction hypothesis:
  - No more than $n - (L - 1)$ threads at level $L - 1$
- Assume all at level $L - 1$ enter level L
- Assume A is the last to write victim[L]
- Want to show: A must have seen B in level[L] and since victim[L] == A could not have entered

Filter Lock

- From code:

```
public void lock(){
  for (int L = 1; L < n; L++) {
    level[i]  = L;
    victim[L] = i;
    while ((∄ k != i: level[k] >= L) &&
           victim[L] == i ) {};
  }}
```

- $w_B(\text{level}[B] = L) < w_B(\text{victim}[L] = B)$

Filter Lock

- From code:

```
public void lock(){
  for (int L = 1; L < n; L++) {
    level[i]  = L;
    victim[L] = i;
    while ((∄ k != i: level[k] >= L) &&
           victim[L] == i ) {};
  }}
```

- $w_A(\text{victim}[L] = A) < r_A(\text{level}[B]) < r_A(\text{victim}[L])$

Filter Lock

- By assumption A is the last to write to victim[L]:
  - $w_B(\text{victim}[L] = B) < w_A(\text{victim}[L] = A)$

Filter Lock

- Combining observations:
  - $w_B(\text{level}[B] = L) < w_B(\text{victim}[L] = B)$
  - $w_A(\text{victim}[L] = A) < r_A(\text{level}[B]) < r_A(\text{victim}[L])$
  - $w_B(\text{victim}[L] = B) < w_A(\text{victim}[L] = A)$

Filter Lock

- Combining Observations
  - $w_B(\text{level}[B] = L) < w_B(\text{victim}[L] = B)$
  - $< w_A(\text{victim}[L] = A) < r_A(\text{level}[B])$
  - $< r_A(\text{victim}[L])$
- A read level[B]>=L and victim[L]=A, so A could not have entered level L

Filter Lock

- Filter is starvation-free
  - Reverse induction on number of levels
  - Base case: level $n - 1$: There is at most one thread, so there can be no starvation
  - Induction step:
    - Assume A is stuck at level *j*.
    - IH: All higher levels have eventually emptied out
    - Once A sets level[A] = j: Any thread reading level[A] is prevented from entering level *j*.

Filter Lock

- All threads stuck at level *j* are in the waiting loop

```
public void lock(){
  for (int L = 1; L < n; L++) {
    level[i]  = L;
    victim[L] = i;
    while ((∄ k != i: level[k] >= L) &&
           victim[L] == i ) {};
  }}
```

- Values of victim and level fields no longer change

Filter Lock

- Argue by induction on the number of threads at level *j*.
    - If A is the only one, then it will enter level *j*+1
    - Induction step: Assume A and B are stuck at level *j*.
        - A is stuck as long as it reads: victim[j]=A
        - B is stuck as long as it reads: victim[j]=B
    - Since the victim field is no longer written, one of them will enter level *j*+1.
    - This reduces the number of threads stuck by one
    - IH: All these make progress

Filter Lock

Filter Lock

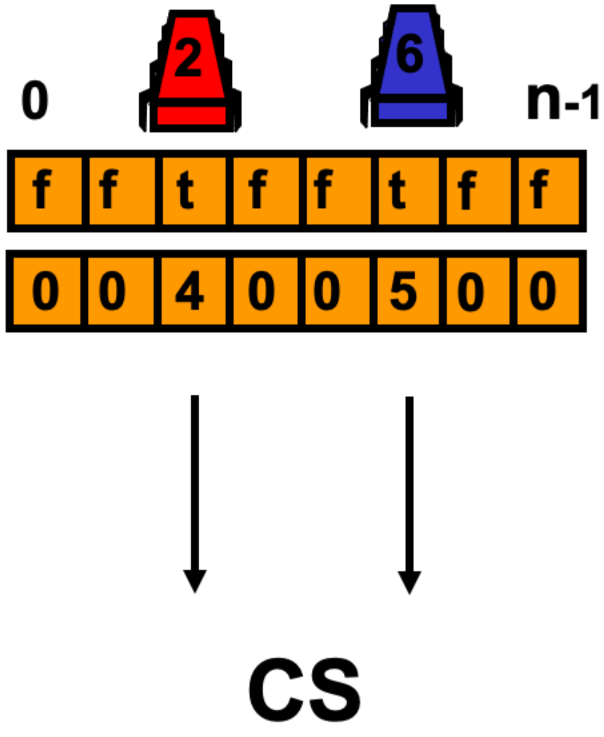
- Properties:
  - No starvation
    - A fortiori: no deadlocks
  - But threads can be overtaken by others

Lock Algorithms: Bakery

- Idea:
    - You enter the waiting queue and take a number
    - Wait until all lower numbers have been served
- Use lexicographic ordering on pairs of threads x numbers

Lock Algorithms: Bakery

- Use a flag and a label array
  - Labels



```
class Bakery implements Lock {
   boolean[] flag;
   Label[] label;
  public Bakery (int n) {
    flag  = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
```

Lock Algorithms: Bakery

```
class Bakery implements Lock {
   …
  public void lock() {
   flag[i]  = true;
   label[i] = max(label[0],…,label[n-1])+1;
   while (∃k flag[k]
             && (label[i],i) > (label[k],k));
  }
```

Lock Algorithms: Bakery

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],…,label[n-1])+1;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Doorway

Lock Algorithms: Bakery

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],…,label[n-1])+1;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

I am interested

Lock Algorithms: Bakery

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],…,label[n-1])+1;
  while (∃k flag[k]
          && (label[i],i) > (label[k],k));
 }
```

Take increasing labels
read in arbitrary order

Lock Algorithms: Bakery

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],…,label[n-1])+1;
  while (∃k flag[k]
          && (label[i],i) > (label[k],k));
 }
```

Someone is interested

Lock Algorithms: Bakery

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],…,label[n-1])+1;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

And they have a lower label

Lock Algorithms: Bakery

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],…,label[n-1])+1;
  while (∃k flag[k]
            && (label[i],i) > (label[k],k));
 }
```

So I am waiting

Lock Algorithms: Bakery

```
class Bakery implements Lock {
   …
  public void lock() {
   flag[i]  = true;
   label[i] = max(label[0],…,label[n-1])+1;
   while (∃k flag[k]
            && (label[i],i) > (label[k],k));
  }
```

Lock Algorithms: Bakery

- To unlock, just express that you are no longer interested

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

Lock Algorithms: Bakery

- No deadlock
    - There is always one thread with earliest label
    - Ties are impossible

Lock Algorithms: Bakery

- •If $D_A < D_B$ then
  - A's label is smaller
- And:
  - writeA(label[A]) < readB(label[A]) < writeB(label[B]) < readB(flag[A])
- So B sees
  - smaller label for A
- locked out while flag[A] is true

Lock Algorithms: Bakery

- Mutual Exclusion
  - Suppose A and B are in CS together
  - Suppose A has earlier label
  - When B entered, it must have seen:
    - flag[A]==0   or   label(A) > label(B)
  - But labels are strictly increasing so:
    - B must have seen flag[A] == 0

Lock Algorithms: Bakery

- Mutual Exclusion:
  - Therefore: $\text{Labeling}_B < r_B(\text{flag}[A])$
    - $< w_A(\text{flag}[A]$
    - $< \text{Labeling}_A$
  - This contradicts the assumption that A has an earlier label

Lock Algorithms: Bakery

- Why is this not practicable:
  - Labels cannot be guaranteed to be always increasing
  - Threads need to read as many registers as there are potential threads