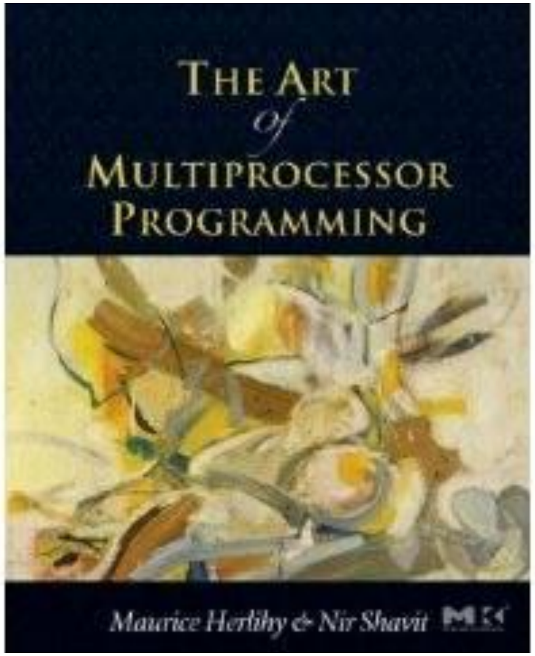
Department of Computer Science

COS 226

Chapter 2

Mutual Exclusion

Acknowledgement



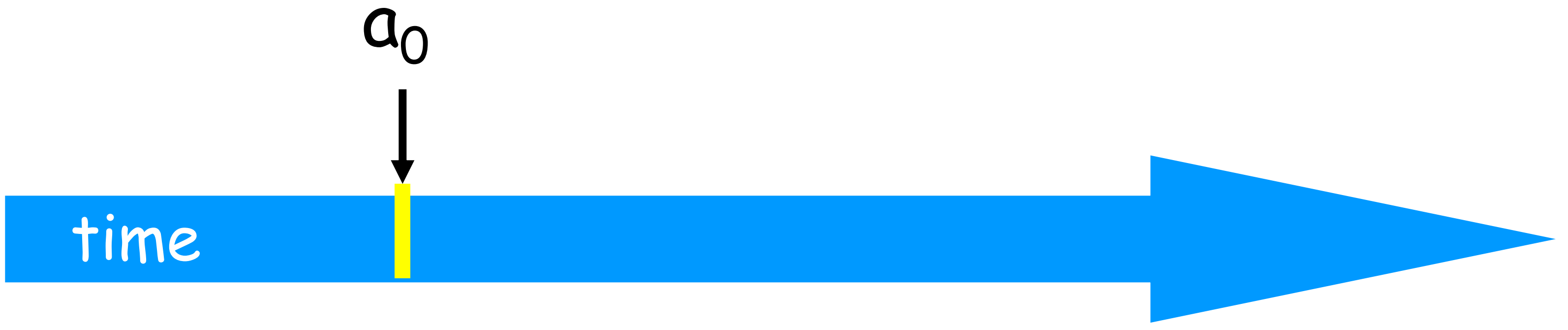
- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

Why is Concurrent Programming so Hard?

- Try preparing a seven-course banquet
  - By yourself
  - With one friend
  - With twenty-seven friends …
- Before we can talk about programs
  - Need a language
  - Describing time and concurrency

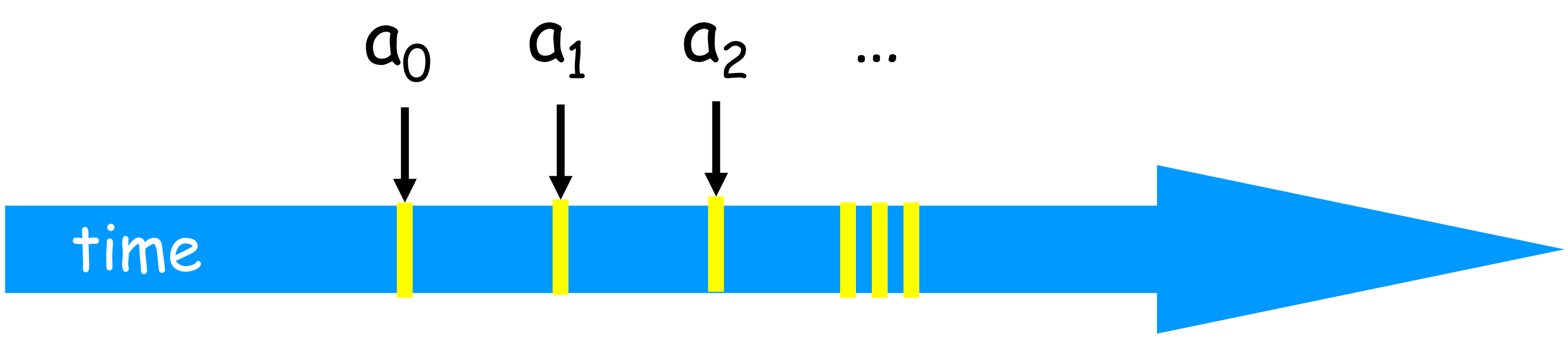
Events

- An *event* a_0 of thread A is
  - Instantaneous
  - No simultaneous events (break ties)



Threads

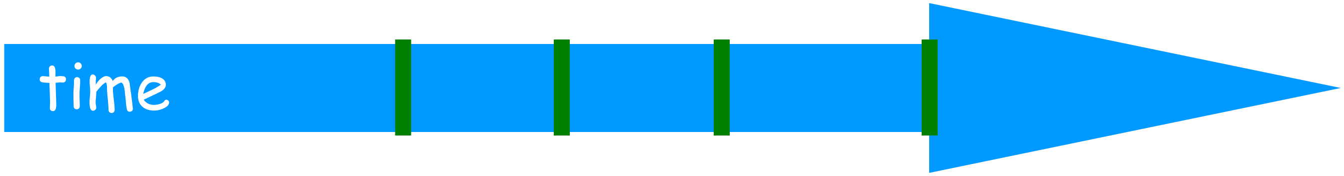
- A *thread* A is (formally) a sequence a_0, a_1, ... of events
  - "Trace" model
  - Notation: $a_0 \rightarrow a_1$ indicates order



Concurrency

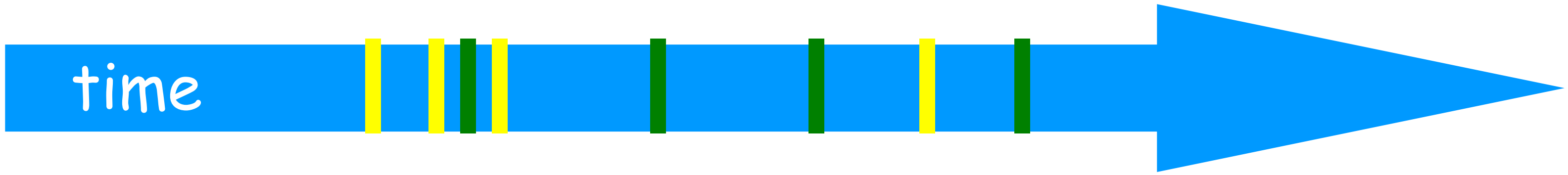
- Thread A

- Thread B



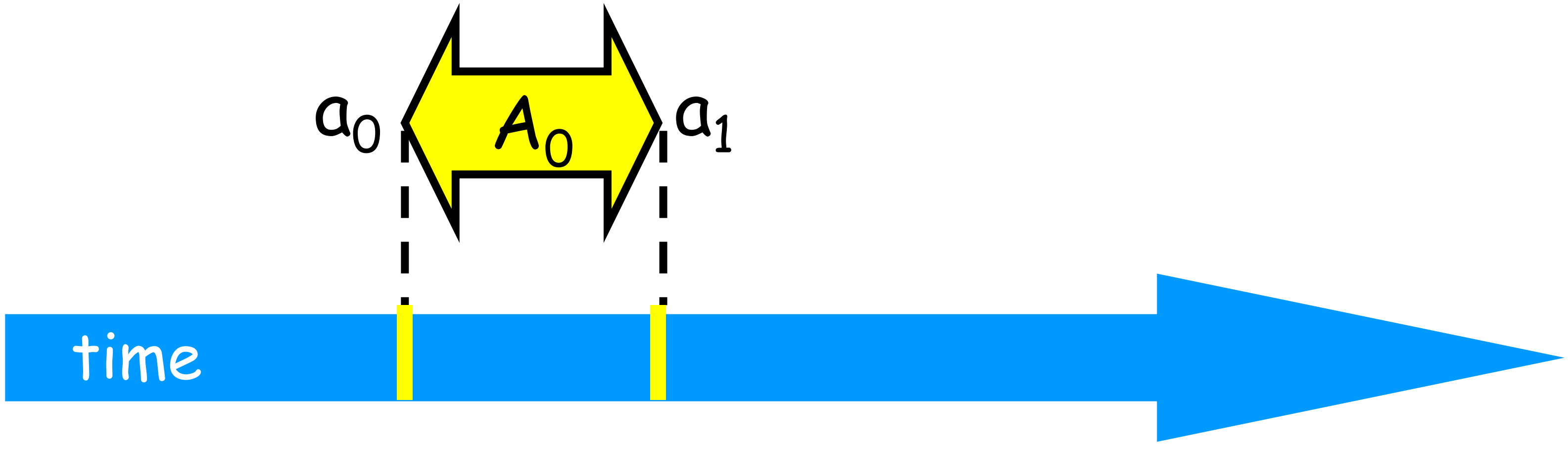
Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent

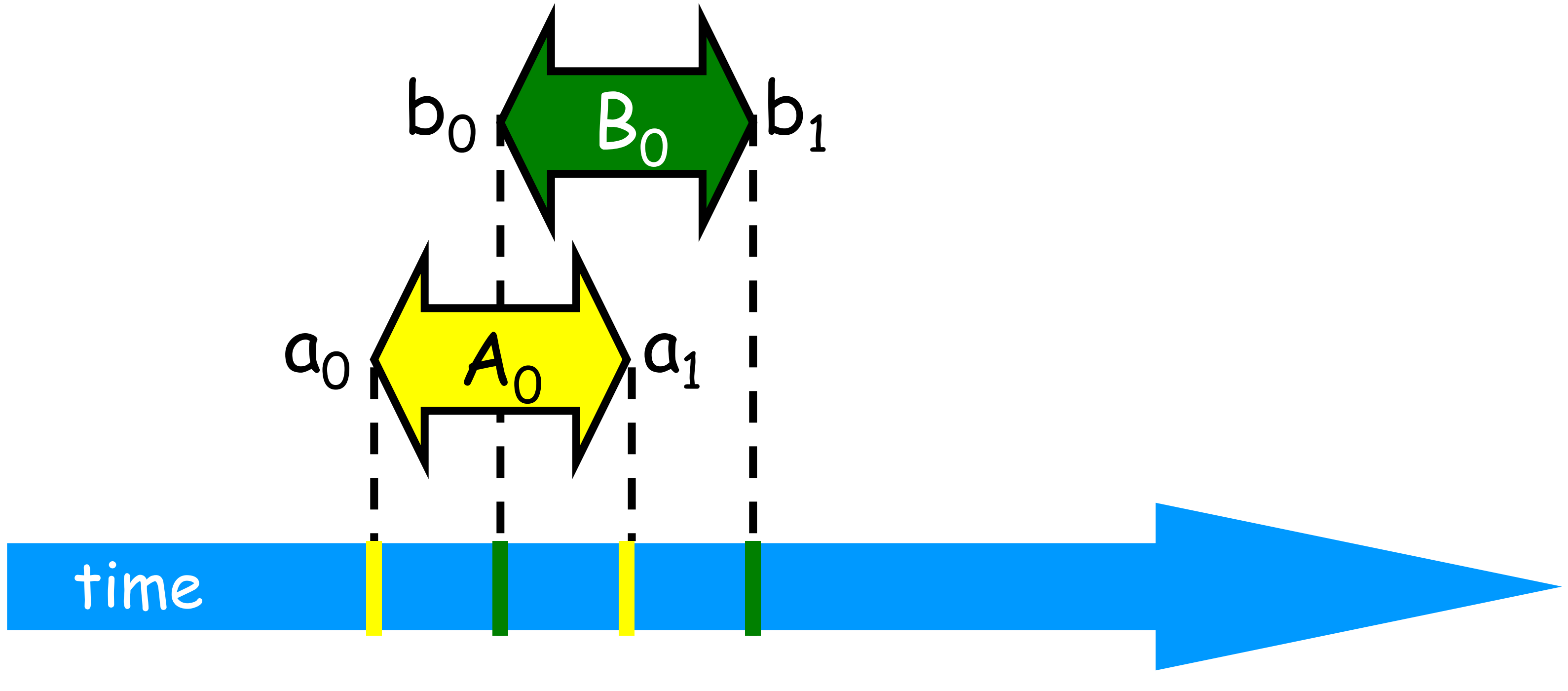


Intervals

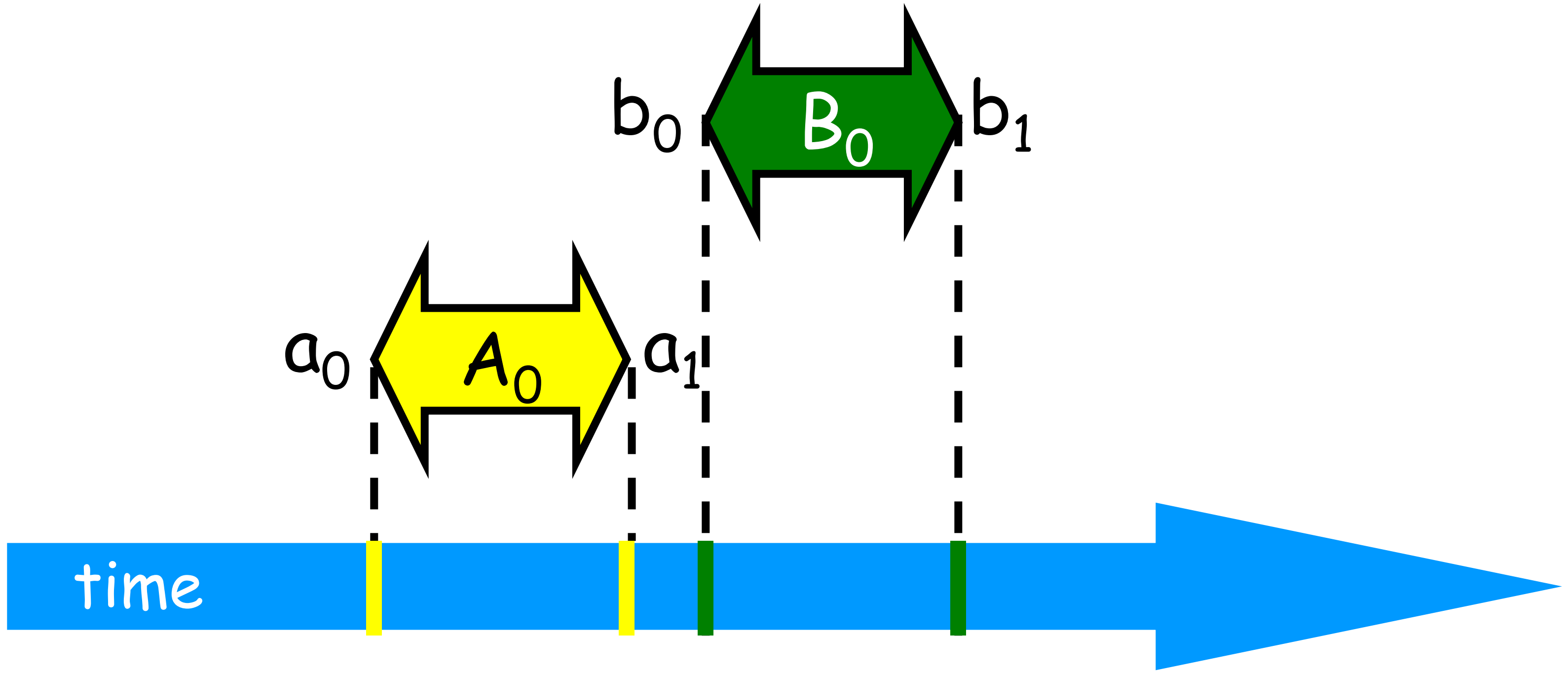
- An *interval* $A_0 = (a_0, a_1)$ is
  - Time between events a_0 and a_1



Intervals may Overlap

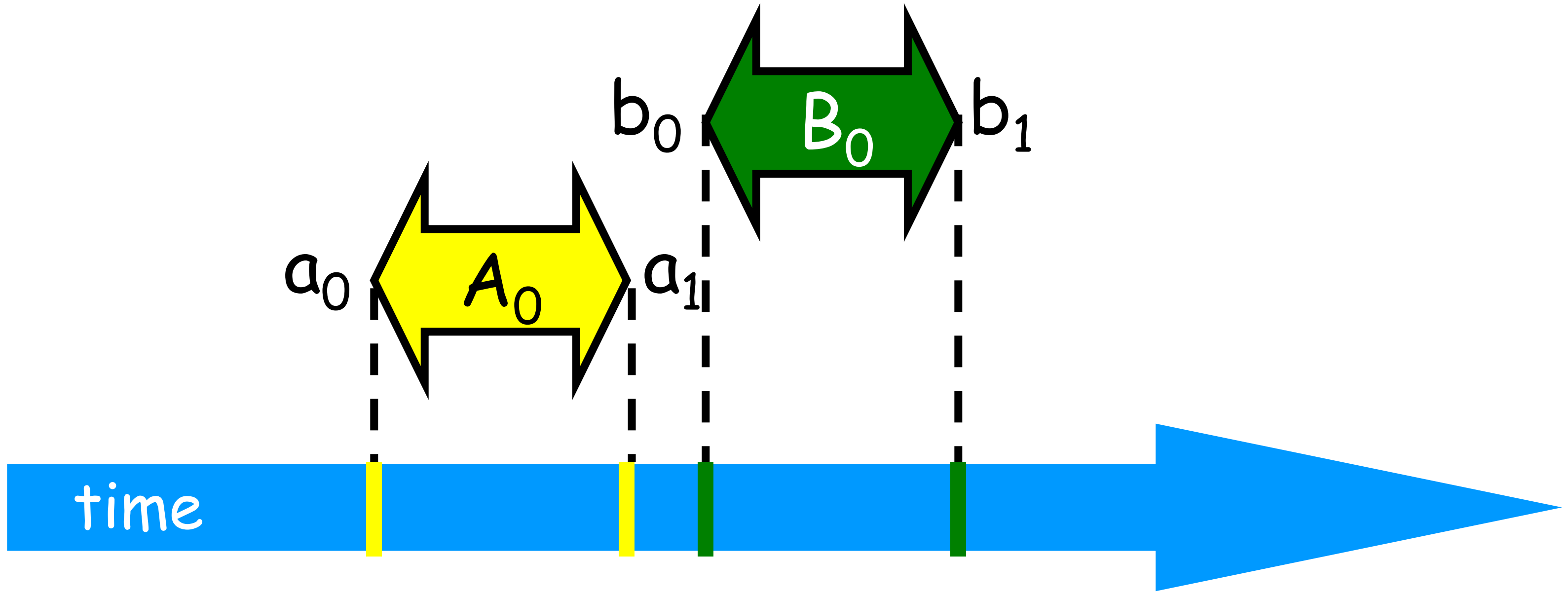


Intervals may be Disjoint

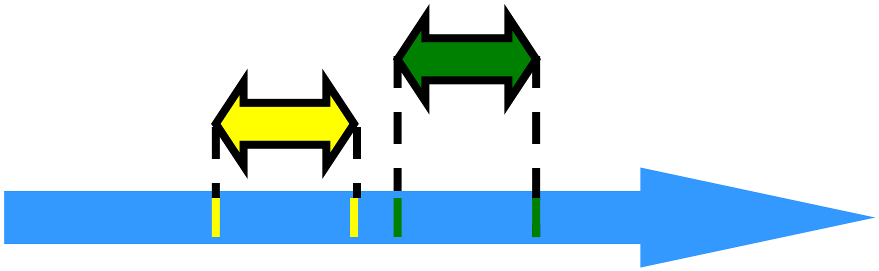


Precedence

Interval A_0 precedes interval B_0

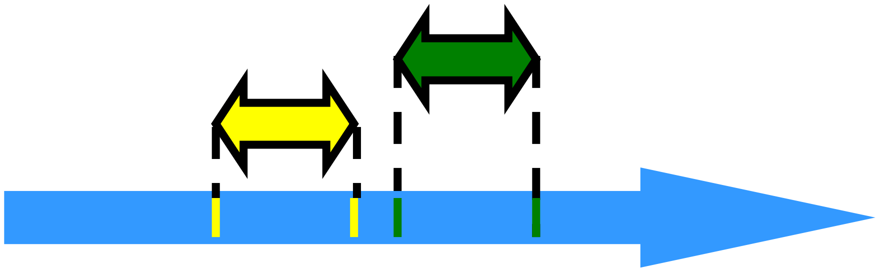


Precedence



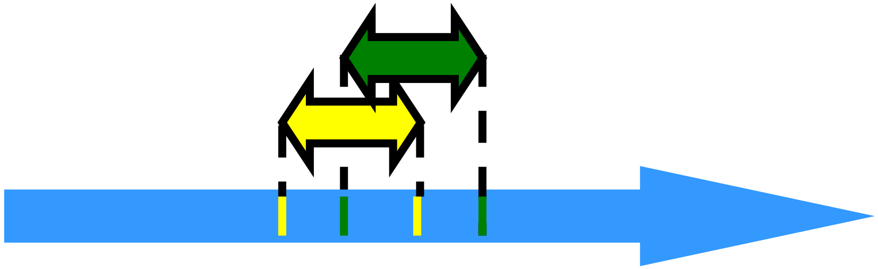
- Notation: $A_0 \rightarrow B_0$
- Formally,
  - End event of A_0 before start event of B_0
  - Also called “happens before” or “precedes”

Precedence Ordering



- Remark: $A_0 \rightarrow B_0$ is just like saying
  - 1066 AD → 1492 AD,
  - Middle Ages → Renaissance,

Precedence Ordering



- Never true that A → A
- If A →B then not true that B →A
- If A →B & B →C then A →C
- Funny thing: A →B & B →A might both be false!

Repeated Events

```
while (mumble) {
  a0; a1;
}
```

a_0^k

k-th occurrence of event a_0

A_0^k

k-th occurrence of interval $A_0 = (a_0, a_1)$

Implementing a Counter

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    value++;
  }
}
```

Implementing a Counter

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return value;
  }
}
```

Critical Section

- Block of code that can be executed by only one thread at a time
- Needs Mutual Exclusion
- Standard way to approach mutual exclusion is through locks

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

Locks (Mutual Exclusion)

```
public interface Lock {

  public void lock();

  public void unlock();
}
```

acquire lock

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock

release lock

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

acquire Lock

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Release lock
(no matter what)

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Critical section

Properties of a good Lock algorithm

- Mutual Exclusion
- Deadlock-free
- Starvation-free

Mutual Exclusion

- Threads do not access critical section at same time

Deadlock-free

- If some thread attempts to acquire the lock, some thread will succeed in acquiring the lock

Deadlock-free

- If <u>**some**</u> thread attempts to acquire the lock, <u>**some**</u> thread will succeed in acquiring the lock
  - System as a whole makes progress
  - Even if individuals starve
  - At least one thread is completing

Starvation-free

- Every thread that attempts to acquire the lock will eventually succeed

Starvation-free

- **Every** thread that attempts to acquire the lock will eventually succeed
  - If a thread calls lock() it will eventually acquire the lock
  - Individual threads make progress

Locks

- Let’s start with lock solutions for 2 concurrent threads…

Two-Thread Conventions

```
class … implements Lock {
  …
  // thread-local index, 0 or 1
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
  …
  }
}
```

Two-Thread Conventions

```
class … implements Lock {
  …
  // thread-local index, 0 or 1
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
  …
  }
}
```

Henceforth: i is current thread, j is other thread

LockOne

- Basic idea:
  - Thread indicates interest in acquiring lock
  - Checks to see if other thread is currently in critical section
    - If true, waits until other thread finishes
    - If not, enters critical section

LockOne

```
class LockOne implements Lock {
private boolean[] flag = new boolean[2];

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

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

LockOne

```
class LockOne implements Lock {
private boolean[] flag =
                        new boolean[2];
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }

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

Set my flag

LockOne

```
class LockOne implements Lock {
private boolean[] flag =
                        new boolean[2];
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }

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

Set my flag

Wait for other flag to go false

LockOne

```
class LockOne implements Lock {
private boolean[] flag =
                        new boolean[2];
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }

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

When release lock set my flag again

Deadlock Freedom?

- Concurrent execution:

```
flag[i] = true;         flag[j] = true;
while (flag[j]){}       while (flag[i]){}
```

- If each thread sets its flag to true and waits for the other, they will wait forever
- No deadlock freedom

LockOne Summary

- LockOne offers mutual exclusion
- When accessed sequentially, LockOne works fine
- However with concurrent threads, LockOne is not Deadlock-free

LockTwo

- Basic idea:
  - When attempting to acquire lock, offer to be the victim that has to defer to other thread
  - While current thread is the victim, wait until other thread becomes the victim
  - When current thread no longer the victim, enter the critical section

LockTwo

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

 public void unlock() {}
}
```

LockTwo

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

 public void unlock() {}
}
```

Let other go first

LockTwo

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

 public void unlock() {}
}
```

Wait for permission

LockTwo

```
public class Lock2 implements Lock {
 private int victim;
 public void lock() {
  victim = i;
  while (victim == i) {};
 }

 public void unlock() {}
}
```

Nothing to do

LockTwo Claims

- Satisfies mutual exclusion
  - If thread **`i`** in CS
  - Then **`victim == j`**
  - Cannot be both 0 and 1

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

LockTwo Summary

- LockTwo offers Mutual Exclusion
- Works fine with concurrent threads
- However results in Deadlock with sequential threads

- LockOne and LockTwo thus complement each other

Peterson Lock

- Combine LockOne and LockTwo
  - Enable successful sequential access provided by LockOne
  - Enables successful concurrent access provided by LockTwo

Peterson Lock

- Basic idea:
  - Current thread indicates interest in acquiring the lock
  - Current thread offers to be the victim
  - If no interest from other thread and no longer the victim, then continue to critical section

Peterson’s Algorithm

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

Peterson's Algorithm

Announce I'm interested

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

Peterson's Algorithm

Announce I'm interested

Defer to other

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

Peterson's Algorithm

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

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

Peterson's Algorithm

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

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

No longer interested

Mutual Exclusion

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

- If thread **0** in critical section,
  - `flag[0] = true`,
  - `victim = 1`

- If thread **1** in critical section,
  - `flag[1] = true`,
  - `victim = 0`

Cannot both be true

Deadlock Free

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

- Thread blocked
  - only at **`while`** loop
  - Only if other’s flag is true
  - only if it is the **victim**
- Solo: other’s flag is false
- Both: one or the other must not be the victim

Starvation Free

- Thread **`i`** would be blocked only if **`j`** repeatedly re-enters so that

  `flag[j] == true` and **`victim == i`**
- When **`j`** re-enters
  - it sets **`victim`** to **`j`**.
  - So **`i`** gets in
- Thus: Starvation free

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

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

Locks

- Moving on to solutions for *n* concurrent threads

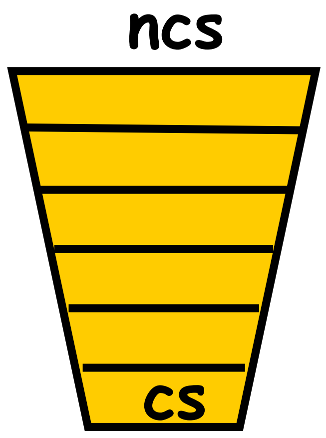
Filter Lock

- Peterson lock adapted to work with *n* threads instead of just 2
- Thread has to traverse *n*-1 waiting rooms in order to acquire the lock

The Filter Algorithm for *n* Threads

There are ***n-1*** "waiting rooms" called levels

- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through



Filter

```
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;
      }}
   …
}
```

Filter

```
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

```
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 release(int i) {
    level[i] = 0;
  }}
```

One level at a time

Filter

```
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); // busy wait
  }}
  public void release(int i) {
    level[i] = 0;
  }}
```

Announce intention to enter level L

Filter

```
class Filter implements Lock {
  int level[n];
  int victim[n];
  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 release(int i) {
    level[i] = 0;
  }}
```

Give priority to anyone but me

Filter

Wait as long as someone else is at same or higher level, and I'm designated victim

```
class Filter implements Lock {
  int level[n];
  int victim[n];
  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 release(int i) {
    level[i] = 0;
  }}
```

Filter

```
class Filter implements Lock {
  int level[n];
  int victim[n];
  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 release(int i) {
    level[i] = 0;
  }}
```

Thread enters level L when it completes the loop

No Starvation

- Filter Lock satisfies properties:
  - Just like Peterson Algorithm at any level
  - So no one starves
- But what about fairness?
  - Threads can be overtaken by others

Waiting

- Starvation freedom guarantees that every thread that calls lock() eventually enters the critical section
- It however makes no guarantee about how long that can take

Waiting

- Ideally if A calls lock() before B, then A should enter critical section before B
- However this does not currently work since we cannot determine which thread called lock() first
- Locks should thus be further defined

Fairness

- Locks should be first-some-first served

Filter Lock again

- Filter Lock satisfies properties:
  - No one starves
  - But very weak fairness
    - Can be overtaken **arbitrary** # of times
  - That's pretty lame…

Bakery Algorithm

- Provides First-Come-First-Served
- How?
  - Take a “number”
  - Wait until lower numbers have been served

- Each thread takes a number when attempting to acquire the lock and waits until no thread with an earlier number is trying to acquire it

Bakery Algorithm

```
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;
    }
  }
 …
```

Bakery Algorithm

- flag[*A*] is a boolean flag indicating whether *A* wants to enter the critical section
- label[*A*] is an integer that contains thread A's "number" when entering the bakery

Bakery Algorithm

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

Bakery Algorithm

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

I'm interested

Bakery Algorithm

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

Take increasing label (read labels in some arbitrary order)

Label is created as one greater than the maximum of the other thread's labels

Bakery Algorithm

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

Someone is interested

Bakery Algorithm

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

Someone is interested

And someone has a smaller number than me

THEORETICALLY

Bakery Algorithm

- If two threads try to acquire the lock concurrently, they may read the same maximum number
- Threads thus have unique pairs consisting of number as well as thread ID

Bakery Algorithm

(label[i],i) << (label[j],j)

If and only if

label[i] < label[j] OR label[i] = label[j] and i < j

Bakery Algorithm

```
class Bakery implements Lock {
   boolean flag[n];
   int label[n];

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

Someone is interested

With lower (label,i) in lexicographic order

Bakery Algorithm

- In other words:
- Thread A must wait if:
  - Another thread is interested AND the other thread’s number is lower than thread A
    - OR
  - Another thread is interested AND the two threads have the same number but the other’s threads ID is smaller than A

Bakery Algorithm

```
class Bakery implements Lock {

    …

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

Bakery Algorithm

```
class Bakery implements Lock {

    …

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

No longer interested

To analyse:

- Does the lock provide:
  - Mutual exclusion?
    - YES – two threads cannot be in the critical section at the same time since one of them will have an earlier label pair

To analyse:

- Starvation freedom?
  - YES – if a thread exists the critical section and immediately wants to reacquire the lock, he will first have to take a new, later number allowing the other waiting threads to gain access first

To analyse:

- Deadlock freedom?
  - YES – there is always one thread with the earliest label, ties are not possible because of labels consist of number and order in array

To analyse:

- The Bakery algorithm also provides First-come-first-served
  - If A calls lock() before B, then A's number is smaller than B's number
  - So B is locked out while flag[A] is true

Potential issue:

- With the current Bakery algorithm we are assuming that we have an infinite amount of numbers to use
- In practice this is not the case

Bounded timestamps

- Labels in the Bakery lock grow without bounds
- In a long-lived system we may have to worry about overflow
- If a thread’s label silently rolled over from a large number to zero, the first-come-first-served property no longer holds

Bounded timestamps

- In the Bakery algorithm, the idea of labels can be replaced by timestamps
- Timestamps can ensure order among the contending threads
- We will thus need to ensure that if one thread takes a label after another, then the latter has the higher timestamp

Bounded timestamps

- Timestamps need the ability to:
  - Scan – read the other thread's timestamps
  - Label – assign itself a larger timestamp

Possible solution

- To construct a Sequential timestamping system
- Each thread perform scan-and-label completely one after the other
- Uses mutual exclusion

Bakery algorithm

- The Bakery algorithm is elegant and fair
- However it is not considered practical
  - Why?
  - Principal drawback is the need to read *n* distinct location where *n* can be very large