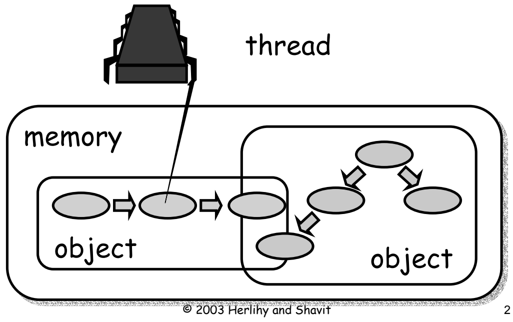
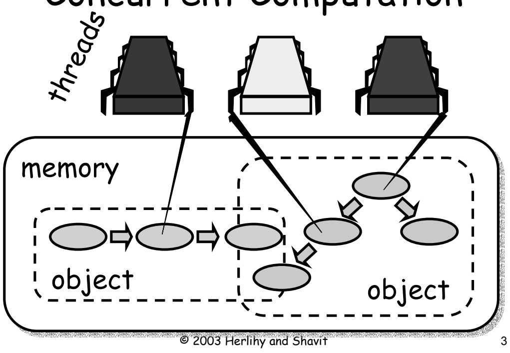
Mutual Exclusion

Nir Shavit
Sub-ing For Nancy Lynch
Distributed Computing
Fall Term

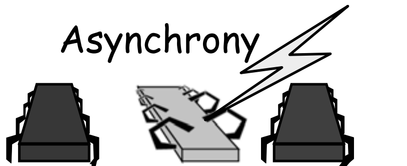
Sequential Computation



Concurrent Computation



Asynchrony



- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)

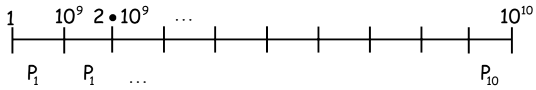
Model Summary

- Multiple *threads*
  - Sometimes called *processes*
- Multiple *CPU's*
  - Sometimes called *processors*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays

Parallel Primality Testing

- Challenge
  - Print primes from 1 to 10^{10}
- Given
  - Ten-processor multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)

Load Balancing



- Split the work evenly
- Each thread tests range of 10^9

Procedure for Thread *i*

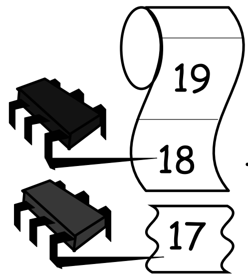
```
void run(int i) {
  for (j = i*10^9+1, j < (i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```


Issues

- Larger Num ranges have fewer primes
- Larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need *dynamic* load balancing

rejected

Shared Counter



each thread takes a number

Procedure for Thread *i*

```
int counter = new Counter(1);

void thread(int i) {
  int j = 0;
  while (j < 10^10) {
    j = counter.inc();
    if (isPrime(j))
      print(j);
  }
}
```


Procedure for Thread *i*

```
int counter = new Counter(1);

void thread(int i) {
  int j = 0;
  while (j < 10^10) {
    j = counter.inc();
    if (isPrime(j))
      print(j);
  }
}
```

Shared counter object

Stop when every value taken

Increment & return each new value

Counter Implementation

```
public class Counter {
  private long value;

  public long inc() {
    return value++;
  }
}
```

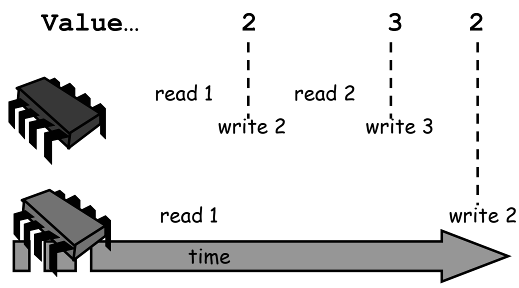
OK for uniprocessor, not for multiprocessor

What It Means

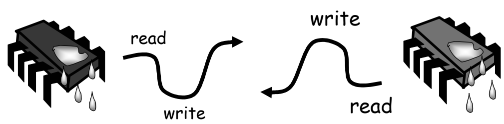
```
public class Counter {
  private long value;

  public long inc() {
    return value++;
  }
}
```

```
temp  = value;
value = value + 1;
return temp;
```


Uh-Oh



FLP: Facts of Life for Processors



If we could only glue reads and writes…

Challenge

```
public class Counter {
  private long value;

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

Make these steps *atomic* (indivisible)

An Aside: Java™

```
public class Counter {
  private long value;

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

Critical section

Synchronized block

Mutual Exclusion in Detail

- Formal problem definitions
- Solutions for 2 threads
- Solutions for *n* threads
- Fair solutions
- Inherent costs

Warning

- You will never use these protocols
  - Get over it
- You had better understand them
  - The same issues show up everywhere
  - If you can't reason about these, you won't get far with "real" protocols ….

Why is Concurrent Programming so Hard?

- Cooking an omelet is easy
- Cooking a five-course meal is hard
- Before we can talk about programs
  - Need a language
  - Describing time and concurrency

Time

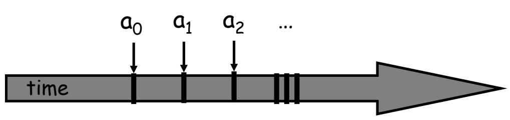
- "Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external." (I. Newton, 1689)
- "Time is Nature's way of making sure that everything doesn't happen all at once." (Anonymous, circa 1970)

Events

- An *event* a_0 of thread A is
  - Instantaneous
  - No simultaneous events

Threads

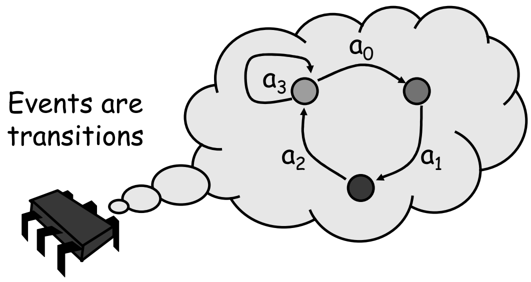
- A *thread* A is (formally) a sequence a_0, a_1, ... of events
  - "Trace" model
  - Notation: $a_0 \rightarrow a_1$ indicates order



Example Thread Events

- *Assign to shared variable*
- *Assign to local variable*
- Call method
- Return from called method
- Lots of other things …

Threads are State Machines



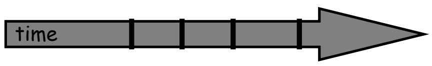
States

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

Concurrency

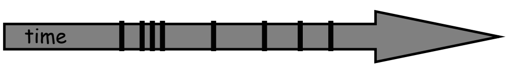
- Thread A

- Thread B



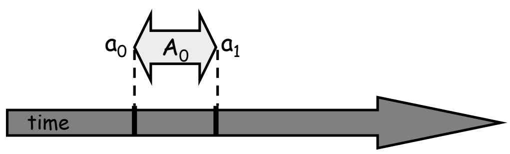
Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

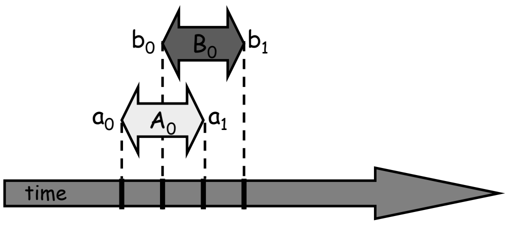


Intervals

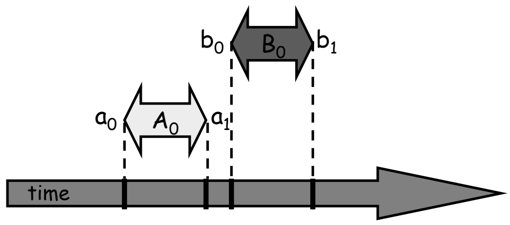
- An *interval* $A_0 = (a_0, a_1)$ is
  - Time between events a_0 and a_1



Intervals may Overlap

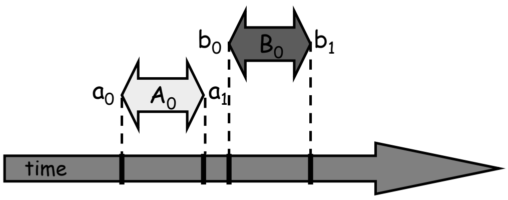


Intervals may be Disjoint

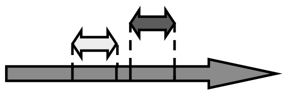


Precedence

Interval A_0 precedes interval B_0

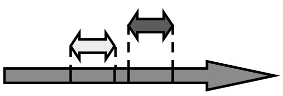


Precedence



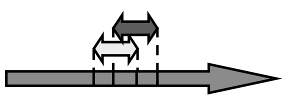
- Notation: $A_0 \rightarrow B_0$
- Formally,
  - End event of A_0 before start event of B_0
  - Also called "happens before"

Precedence Ordering



- Remark: $A_0 \rightarrow B_0$ is just like saying
  - 2002 $\rightarrow$ 2003,
  - Middle Ages $\rightarrow$ Renaissance,
- Oh wait,
  - what about this week vs this month?

Precedence Ordering



- Never true that $A \rightarrow A$
- If $A \rightarrow B$ then not true that $B \rightarrow A$
- If $A \rightarrow B$ & $B \rightarrow c$ then $A \rightarrow C$
- Funny thing: $A \rightarrow B$ & $B \rightarrow A$ might both be false!

Partial Orders

(you may know this already)

- Irreflexive:
  - Never true that $A \rightarrow A$
- Antisymmetric:
  - If $A \rightarrow B$ then not true that $B \rightarrow A$
- Transitive:
  - If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$

Total Orders

(you may know this already)

- Also
  - Irreflexive
  - Antisymmetric
  - Transitive
- Except that for every distinct a, b,
  - Either $a \rightarrow b$ or $b \rightarrow c$

Repeated Events

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

a_0^k — *k*-th occurrence of event a_0

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

Review: Atomic Increment

```
public class Counter {
  private long value;

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


Review: Atomic Increment

```
public class Counter {
  private long value;

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

Allow only one thread at a time

Synchronizaton

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

Synchronizaton

```
public interface Lock {
 public void lock();
 public void unlock();
}
```

acquire lock

Synchronizaton

```
public interface Lock {
 public void lock();
 public void unlock();
}
```

acquire lock

release lock

Synchronized Atomic Increment

```
public class Counter {
  private long value;
  private Lock lock;

  public long getAndIncrement() {
   lock.lock();
    int temp = value;
    value = value + 1;
   lock.unlock();
   return temp;
  }}
```


Synchronized Atomic Increment

```
public class Counter {
  private long value;
  private Lock lock;

  public long getAndIncrement() {
   lock.lock();
    int temp = value;
    value = value + 1;
   lock.unlock();;
   return temp;
  }}
```

Acquire Lock

Synchronized Atomic Increment

```
public class Counter {
  private long value;
  private Lock lock;

  public long getAndIncrement() {
   lock.lock();
    int temp = value;
    value = value + 1;
   lock.unlock();;
   return temp;
  }}
```

Acquire Lock

Release Lock

Synchronized Atomic Increment

```
public class Counter {
  private long value;
  private Lock lock;

  public long getAndIncrement() {
   lock.lock();
    int temp = value;
    value = value + 1;
   lock.unlock();;
   return temp;
  }}
```

Critical section

Critical Sections

- Let CS_i^k ⇔ be thread i's k-th critical section

Critical Sections

- Let CS_i^k ⇔ be thread i's k-th critical section
- And CS_j^m ⇔ be thread j's m-th critical section

Critical Sections

- Let CS_i^k ⇔ be thread i's k-th critical section
- And CS_j^m ⇔ be j's m-th execution
- Then either
  - ⇔ ⇔ or ⇔ ⇔

$CS_i^k \rightarrow CS_j^m$

$CS_j^m \rightarrow CS_i^k$

Deadlock-Free

- If thread A calls **lock()**
  - And never returns
  - Then other threads must complete **lock()** and **unlock()** calls infinitely often
- System as a whole makes progress
  - Even if individuals starve

Lockout-Free

- If thread A calls lock()
  - It will eventually return
- Individual threads make progress
- Exercise:
  - Map deadlock-Free vs lockout-free onto different models of Socialism

Two-Thread vs *n*-Thread Solutions

- Two-thread solutions first
  - Illustrate most basic ideas
  - Fits on one slide
- Notation watch: for 2-threads
  - Variable i is my thread
  - Variable j is other thread

Two-Thread Conventions

```
public class Thread {
  private int i;
  private int j = 1-i;

  public void run() {
  …
  }

}
```


Two-Thread Conventions

```
public class Thread {
  private int i;
  private int j = 1-i;

  public void run() {
  …
  }

}
```

ID for this thread

Two-Thread Conventions

```
public class Thread {
  private int i;
  private int j = 1-i;

  public void run() {
  …
  }

}
```

ID for this thread

ID for other thread

Two-Thread Conventions

```
public class Thread {
  private int i;
  private int j = 1-i;

  public void run() {
  …
  }

}
```

Henceforth: i is current thread, j is other thread.

Two-Thread Conventions

```
public class Thread {
  private int i;
  private int j = 1-i;

  public void run() {
  …
  }

}
```

Method that does all the work

LockOne

```
public class LockOne implements Lock {

 private bool flag[2];
 public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }
```

LockOne

```
public class LockOne implements Lock {

 private bool flag[2];
 public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }
```

Set my flag

LockOne

```
public class LockOne implements Lock {

 private bool flag[2];
 public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }
```

Wait for other flag to go false

LockOne Satisfies Mutual Exclusion

- Suppose CS_A concurrent with CS_B
- Before entering critical section
  - **write$_A$(flag[A]=true) → read$_A$(flag[B]==false) → CS_A**
  - **write$_B$(flag[B]=true) → read$_B$(flag[A]==false) → CS_B**
- Implications:
  - **read$_A$(flag[B]==false) → write$_B$(flag[B]=true)**
  - **read$_B$(flag[A]==false) → write$_A$(flag[B]=true)**

LockOne Satisfies Mutual Exclusion

- Implications:
  - **read$_A$(flag[B]==false) → write$_B$(flag[B]=true)**
  - **read$_B$(flag[A]==false) → write$_A$(flag[B]=true)**
- From the code
  - **write$_A$(flag[A]=true) → read$_A$(flag[B]==false)**
  - **write$_B$(flag[B]=true) → read$_B$(flag[A]==false)**

LockOne Satisfies Mutual Exclusion

- Implications:
  - **read$_A$(flag[B]==false) → write$_B$(flag[B]=true)**
  - **read$_B$(flag[A]==false) → write$_A$(flag[B]=true)**
- From the code
  - **write$_A$(flag[A]=true) → read$_A$(flag[B]==false)**
  - **write$_B$(flag[B]=true) → read$_B$(flag[A]==false)**

LockOne Satisfies Mutual Exclusion

- Implications:
  - **read$_A$(flag[B]==false) → write$_B$(flag[B]=true)**
  - **read$_B$(flag[A]==false) → write$_A$(flag[B]=true)**
- From the code
  - **write$_A$(flag[A]=true) → read$_A$(flag[B]==false)**
  - **write$_B$(flag[B]=true) → read$_B$(flag[A]==false)**

LockOne Satisfies Mutual Exclusion

- Implications:
  - $read_A$(flag[B]==false) → $write_B$(flag[B]=true)
  - $read_B$(flag[A]==false) → $write_A$(flag[B]=true)
- From the code
  - $write_A$(flag[A]=true) → $read_A$(flag[B]==false)
  - $write_B$(flag[B]=true) → $read_B$(flag[A]==false)

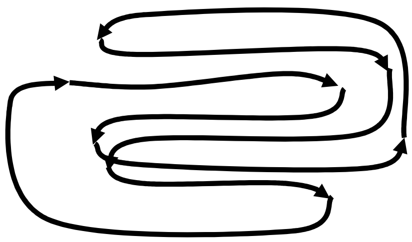
LockOne Satisfies Mutual Exclusion

- Implications:
  - $read_A$(flag[B]==false) → $write_B$(flag[B]=true)
  - $read_B$(flag[A]==false) → $write_A$(flag[B]=true)
- From the code
  - $write_A$(flag[A]=true) → $read_A$(flag[B]==false)
  - $write_B$(flag[B]=true) → $read_B$(flag[A]==false)

LockOne Satisfies Mutual Exclusion

- Implications:
  - $read_A$(flag[B]==false) → $write_B$(flag[B]=true)
  - $read_B$(flag[A]==false) → $write_A$(flag[B]=true)
- From the code
  - $write_A$(flag[A]=true) → $read_A$(flag[B]==false)
  - $write_B$(flag[B]=true) → $read_B$(flag[A]==false)

Cycle!



Deadlock Freedom

- LockOne Fails deadlock-freedom
  - Concurrent execution can deadlock

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

  - Sequential executions OK

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
  - Never both 0 and 1
- Not deadlock free
  - Sequential deadlocks
  - Concurrent does not

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


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

```
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

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 has the turn
- One or the other must have the turn

Lockout Free

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

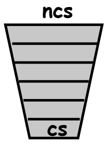
  `flag[j] == true` **and** `victim == i`
- When `j` re-enters
  - it sets victim to `j`.
  - So `i` gets in

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

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

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[n];  // level I want to enter
  int victim[n]; // stop me before I advance again
  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 unlock() {
    level[i] = 0;
  }}
```


Filter

```
class Filter implements Lock {
  int level[n];  // level I want to enter
  int victim[n]; // stop me before I advance again
  public void acquire(int i) {
    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;
  }}
```

One level at a time

Filter

```
class Filter implements Lock {
  int level[n];  // level I want to enter
  int victim[n]; // stop me before I advance again
  public void acquire(int i) {
    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;
  }}
```

Announce intention to enter level L

Filter

```
class Filter implements Lock {
  int level[n];  // level I want to enter
  int victim[n]; // stop me before I advance again
  public void acquire(int i) {
    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;
  }}
```

Give priority to anyone but me

Filter

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

```
  public void acquire(int i) {
    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;
  }}
```

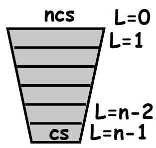
Filter

```
class Filter implements Lock {
  int level[n];  // level I want to enter
  int victim[n]; // stop me before I advance again
  public void acquire(int i) {
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∃ k != i) level[k] >= L) &&
             victim[L] == i); // busy wait
    }}
```

Thread enters level L when it completes the loop

Claim

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



Induction Hypothesis

- No more than n-L+1 at level L-1
- Induction step: by contradiction
- Assume all at level L-1 enter level L
- A last to write victim[L]
- B is any other thread at level L

```
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) {};
 }}
```


First Observation

(1) $\text{write}_B(\text{level[B]=L}) \rightarrow \text{write}_B(\text{victim[L]=B})$

```
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) {};
 }}
```

Use the code, Luke!

Second Verse, Same as the First

(2) $\text{write}_A(\text{victim[L]=A}) \rightarrow \text{read}_A(\text{level[B]})$

```
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) {};
 }}
```


Third Observation

(3) $\text{write}_B(\text{victim[L]=B}) \rightarrow \text{write}_A(\text{victim[L]=A})$

By Hypothesis, A is the last thread to write `victim[L]`

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\rightarrow \text{read}_A(\text{level}[B])$

So A read level[B]>=L and could not have entered level **L** – a contradiction

r-Bounded Waiting

- Want stronger fairness guarantees
- Thread not "overtaken" too much
- Need to adjust definitions ….

r-Bounded Waiting

- Divide `lock()` method into 2 parts:
  - Doorway interval:
    - Written $\mathbf{D_A}$
    - always finishes in finite steps
  - Waiting interval:
    - Written $\mathbf{W_A}$
    - may take unbounded steps

r-Bounded Waiting

- For threads A and B:
  - If $D_A^k \rightarrow D_B^j$
    - A's k-th doorway precedes B's j-th doorway
  - Then $CS_A^k \rightarrow CS_B^{j+r}$
    - A's k-th critical section precedes B's (j+r)-th critical section
    - B cannot overtake A by more than r times
- First-come-first-served means r = 0.

Fairness Again

- Filter Lock satisfies properties:
  - No one starves (no lockout)
  - But very weak fairness
    - Not **r**-bounded for any r!
    - That's pretty lame…

Bakery Algorithm

- Basic Idea
  - Take a "number"
  - Wait until lower numbers have been served
- Lexicographic order
  - (a,b) > (c,d)
    - If a > c, or a = c and b > d

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;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```


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;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Doorway

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;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Waiting

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;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

I'm interested

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;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Take increasing label

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;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Someone is interested

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;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Someone is interested

With higher label

Bakery Algorithm

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

    …

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


Bakery Algorithm

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

    …

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

No longer interested

No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)

First-Come-First-Served

- If $D_A \rightarrow D_B$ then A's label is earlier
  - $write_A$(label[A]) → $read_B$(label[A]) → $write_B$(label[B]) → $read_B$(flag[A])
- So B is locked out while flag[A] is true

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

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


Mutual Exclusion

- Suppose A and B in CS together
- Suppose A has earlier label
- When B entered, it must have seen
  - flag[A] is *false*, or
  - label[A] > label[B]

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

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

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen flag[A] == false
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$
- Which contradicts the assumption that A has an earlier label

Bakery Y2^{32}K Bug

```
class Lock5 implements Lock {
  boolean flag[n];
  int label[n];

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


Bakery Y2^{32}K Bug

```
class Lock5 implements Lock {
  boolean flag[n];
  int label[n];

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

FCFS breaks if `label[i]` overflows

Does Overflow Actually Matter?

- Yes
  - Y2K
  - 18 January 2038 (Unix `time_t` rollover)
  - 16-bit counters
- No
  - 64-bit counters
- Maybe
  - 32-bit counters

Timestamps

- Label variable is really a timestamp
- Need ability to
  - Read others' timestamps
  - Compare them
  - Generate a later timestamp
- Can we do this without overflow?

The Bad News

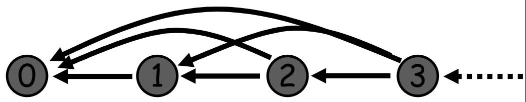
- One can construct a
  - Wait-free (no mutual exclusion)
  - Concurrent
  - Timestamping system
  - That never overflows

This part is hard

Instead …

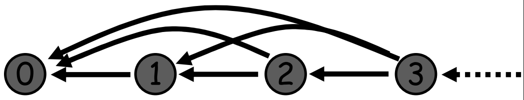
- We construct a Sequential timestamping system
  - Same basic idea
  - But simpler
- Uses mutex to read & write atomically
- No good for building locks
  - But useful anyway

Precedence Graphs



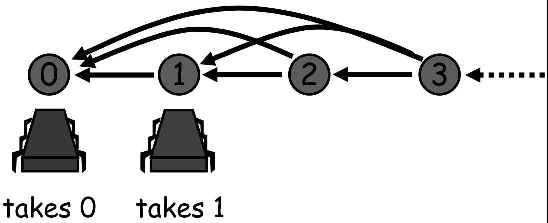
- Timestamps form directed graph
- Edge x to y
  - Means x is later timestamp
  - We say x dominates y

Unbounded Counter Precedence Graph

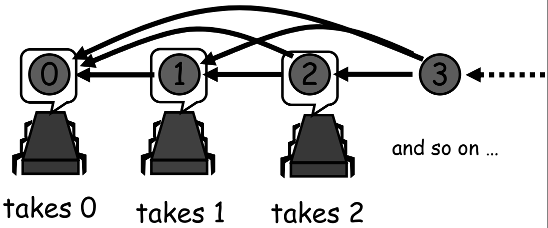


- Timestamping = move tokens on graph
- Atomically
  - read others' tokens
  - move mine
- Ignore tie-breaking for now

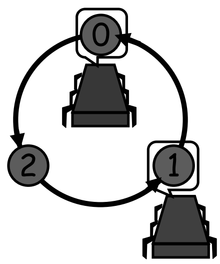
Unbounded Counter Precedence Graph

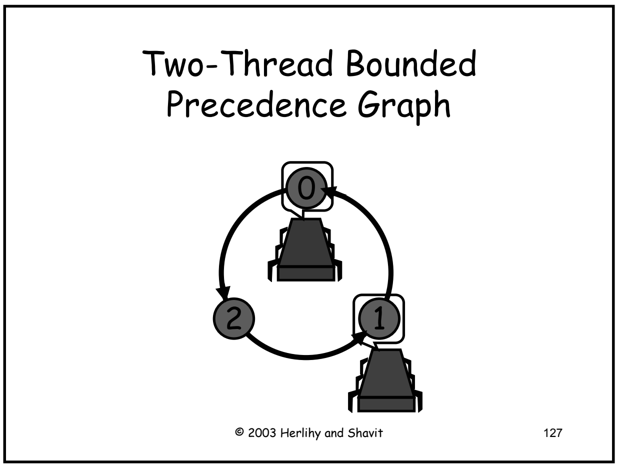


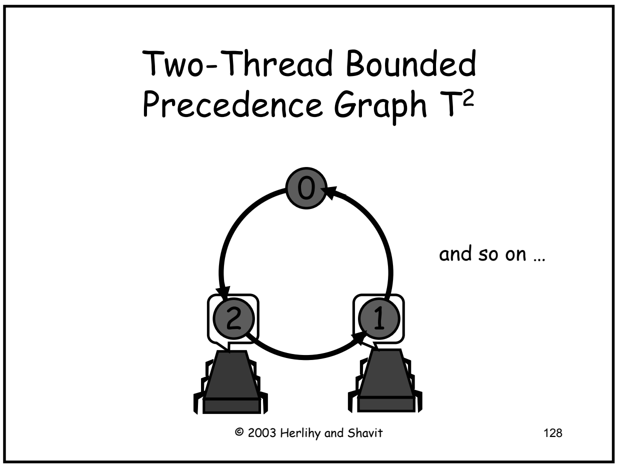
Unbounded Counter Precedence Graph

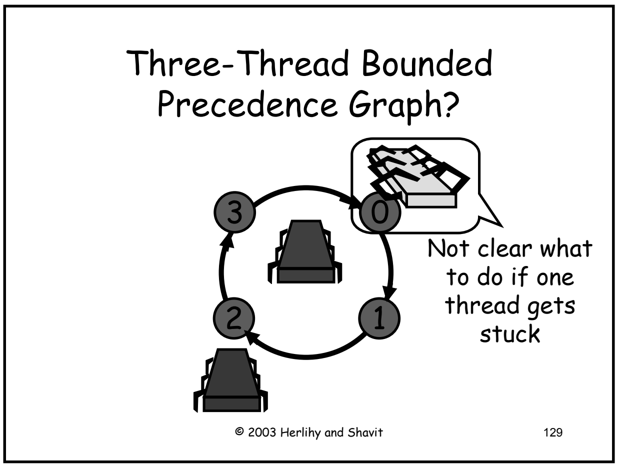


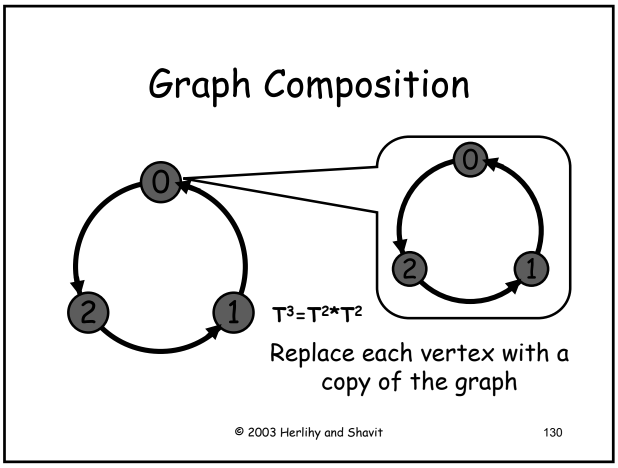
Two-Thread Bounded Precedence Graph

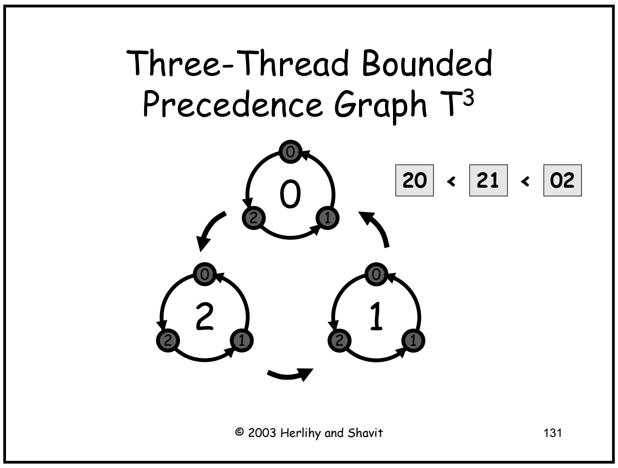


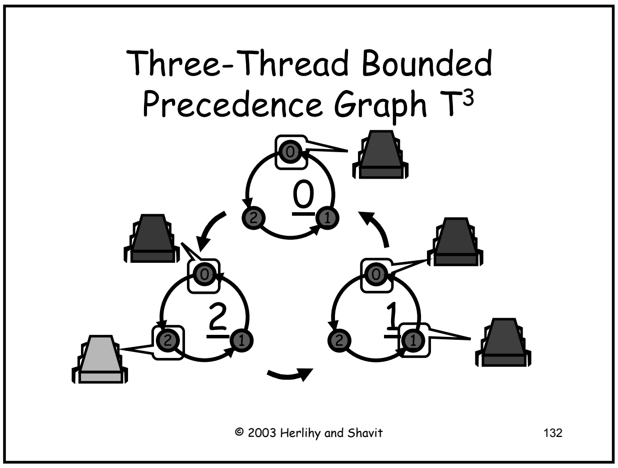

Two-Thread Bounded
Precedence Graph
0
2
1
© 2003 Herlihy and Shavit
127


Two-Thread Bounded
Precedence Graph T^2
0
and so on …
2
1
© 2003 Herlihy and Shavit
128


Three-Thread Bounded
Precedence Graph?
3
0
Not clear what
to do if one
thread gets
stuck
2
1
© 2003 Herlihy and Shavit
129


Graph Composition
0
0
2
1
2
1
$T^3=T^2*T^2$
Replace each vertex with a
copy of the graph
© 2003 Herlihy and Shavit
130

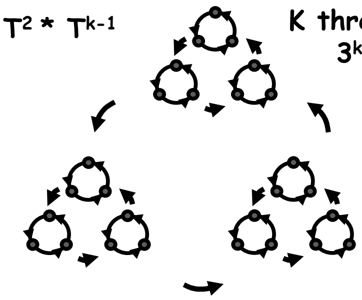

Three-Thread Bounded
Precedence Graph T^3
20 < 21 < 02
0
2
1
© 2003 Herlihy and Shavit
131


Three-Thread Bounded
Precedence Graph T^3
0
2
1
© 2003 Herlihy and Shavit
132

In General

$T^k = T^2 * T^{k-1}$

K threads need 3^k nodes



Deep Philosophical Question

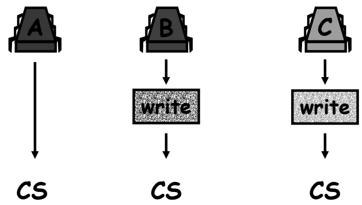
- The Bakery Algorithm is
  - Succinct,
  - Elegant, and
  - Fair.
- Q: So why isn't it practical?
- A: Well, you have to read N distinct object fields

Theorem

At least N multi-reader/single-writer registers are needed to solve deadlock-free mutual exclusion.

Proof

Each thread must write to some register



Can't tell whether A is in critical section

Upper Bound

- You need at least N MRSW registers
- Bakery algorithm
  - Uses 2N MRSW registers
- So the bound is (pretty) tight
- But what if we use MRMW registers?
  - Like the Filter algorithm?

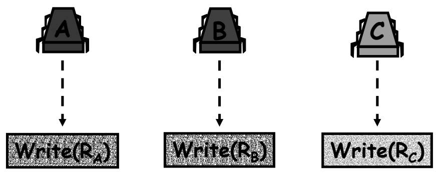
Bad News Theorem

At least N multi-reader/multi-writer registers are needed to solve deadlock-free mutual exclusion.

Let Prove:Theorem

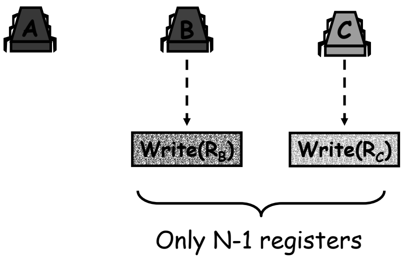
Deadlock-free mutual exclusion for 3 threads requires at least 3 multi-reader multi-writer fields

Covering State

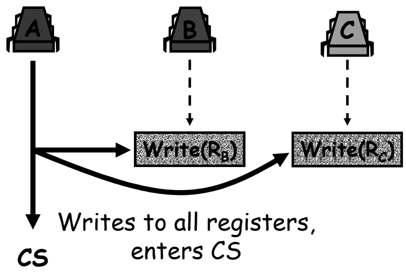


- All registers about to be written
- CS looks empty to all threads

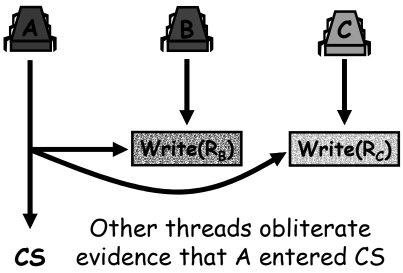
Proof: Assume



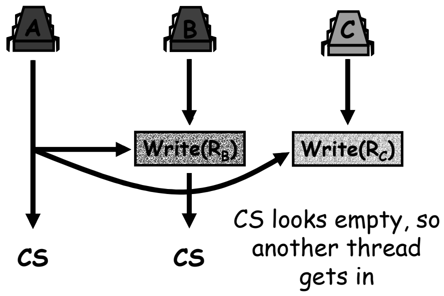
Solo Execution



Covering State



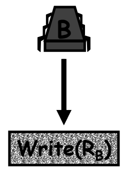
Mutual Exclusion Fails



Proof Strategy

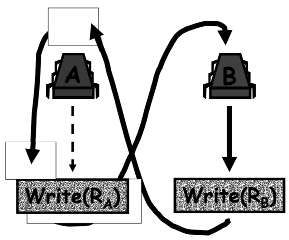
- Proved: In a covering state, you need 3 distinct fields
- Claim: a covering state is reachable from any state where CS is empty

Covering State for One Register



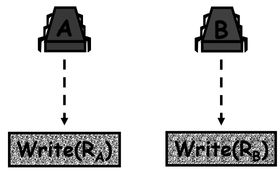
B has to write to some register to enter CS, so stop it just before

Covering State



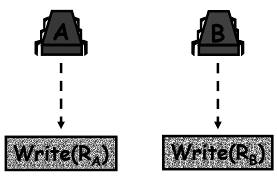
- If we run B through CS 3 times, B must return twice to some register, say R_B

Covering State



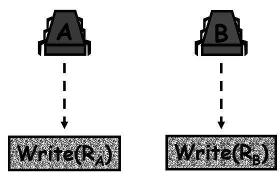
- Start with B covering register R_B
- Run A until it is about to write to uncovered R_A
- Are we done?

Covering State



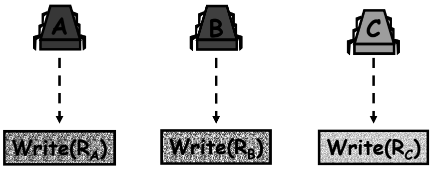
- A could have written to R_B
- CS no longer looks empty to some thread

Covering State



- Run B obliterating traces of A in register R_B
- Run B again until it is about to write to R_B
- Now we are done

Inductively We Can Show



- There is a covering state
  - Where k threads not in CS
  - Cover k distinct registers
  - k=N-1 delivers proof

Mutual Exclusion in Practice

- Shared FIFO queue
- Written in standard Java™

Mutual Exclusion in Practice

- Shared FIFO queue
- Written in standard Java™

Lock-Based Queue

```
public class Queue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public synchronized void enq(Item x) {
  while (this.tail-this.head == QSIZE)
   this.wait();
  this.items[this.tail++ % QSIZE] = x;
  this.notifyAll();
  }
  …
}}
```


Lock-Based Queue

```
public class Queue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public synchronized void enq(Item x) {
  while (this.tail-this.head == QSIZE)
   this.wait();
  this.items[this.tail++ % QSIZE] = x;
  this.notifyAll();
  }
  …
}}
```

Acquire lock on entry, release on exit

Lock-Based Queue

```
public class Queue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public synchronized void enq(Item x) {
  while (this.tail-this.head == QSIZE)
   this.wait();
  this.items[this.tail++ % QSIZE] = x;
  this.notifyAll();
  }
  …
}}
```

If Queue is full, release lock, sleep, try again

Lock-Based Queue

```
public class Queue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public synchronized void enq(Item x) {
  while (this.tail-this.head == QSIZE)
   this.wait();
  this.items[this.tail++ % QSIZE] = x;
  this.notifyAll();
  }
  …
}}
```

Append the item to the queue

Lock-Based Queue

```
public class Queue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public synchronized void enq(Item x) {
  while (this.tail-this.head == QSIZE)
   this.wait();
  this.items[this.tail++ % QSIZE] = x;
  this.notifyAll();
  }
  …
}}
```

Wake up sleeping dequeuers

Observations

- Each method locks entire queue
- No concurrency between methods
- Is this really necessary?

No

And thereby hangs a tale …

Lock-Free Queue

- Imagine two threads
  - One enqueues only
  - One dequeues only
- Do they need mutual exclusion?

Lock-Free Queue

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  Object[QSIZE] items;

  public void enq(Item x) {
    while (tail-head == QSIZE) {};
    items[tail % QSIZE] = x; tail++;
  }
  public Item deq() {
     while (tail == head) {}
     Item item = items[head % QSIZE]; head++;
     return item;
}}
```


Lock-Free Queue

```
public class LockFreeQueue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public void enq(Item x) {
  while (tail-head == QSIZE) {};
  items[tail % QSIZE] = x; tail++;
  }
 public Item deq() {
  while (tail == head) {}
  Item item = items[head % QSIZE];
  head++; return item;
}}
```

Lock-Free Queue

```
public class LockFreeQueue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public void enq(Item x) {
  while (tail-head == QSIZE) {};
  items[tail % QSIZE] = x; tail++;
  }
 public Item deq() {
  while (tail == head) {}
  Item item = items[head % QSIZE];
  head++; return item;
}}
```

Spin while queue is full

Lock-Free Queue

```
public class LockFreeQueue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public void enq(Item x) {
  while (tail-head == QSIZE) {};
  items[tail % QSIZE] = x; tail++;
  }
 public Item deq() {
  while (tail == head) {}
  Item item = items[head % QSIZE];
  head++; return item;
}}
```

Put object in quue

Lock-Free Queue

```
public class LockFreeQueue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public void enq(Item x) {
  while (tail-head == QSIZE) {};
  items[tail % QSIZE] = x; tail++;
  }
 public Item deq() {
  while (tail == head) {}
  Item item = items[head % QSIZE];
  head++; return item;
}}
```

Increment tail counter

Vive La Différence

- The lock-based Queue
  - *Is coarse-grained synchronization*
  - *Critical section is entire method*
- The lock-free Queue
  - *Is fine-grained synchronization*
  - *Critical section is single machine instruction*

Critical Sections

- Easy way to implement concurrent objects
  - Take sequential object
  - Make each method a critical section
- Like synchronized methods in Java™
- Problems
  - Blocking
  - No concurrency

Amdahl's Law

Sequential fraction

Parallel fraction

$$\textbf{Speedup} = \frac{1}{\boxed{1 - c} + \frac{\boxed{c}}{\boxed{n}}}$$

Number of processors

Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup}=2.17= \frac{1}{1-0.6+\frac{0.6}{10}}$$

Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup}=3.57= \frac{1}{1-0.8+\frac{0.8}{10}}$$

Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup}=5.26= \frac{1}{1-0.9+\frac{0.9}{10}}$$

Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

$$\text{Speedup}=9.17= \frac{1}{1-0.99+\frac{0.99}{10}}$$

The Moral

- Granularity matters
  - Long critical sections vs atomic machine instructions
  - Smaller the granularity, greater the speedup

Mutual Exclusion

Nir Shavit
Multiprocessor Synchronization
Fall 2003