Programmation répartie

» Carole Delporte

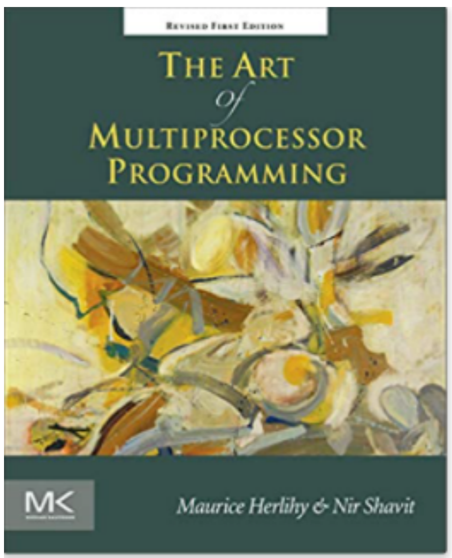
» Moodle :

  » IFECY 140 Programmation Répartie

» programmation :java

» Controle des connaissances: 50% TP et 50% exam

» Bibliographie



» The art of Multiprocessor programming

» Herlihy & Shavit

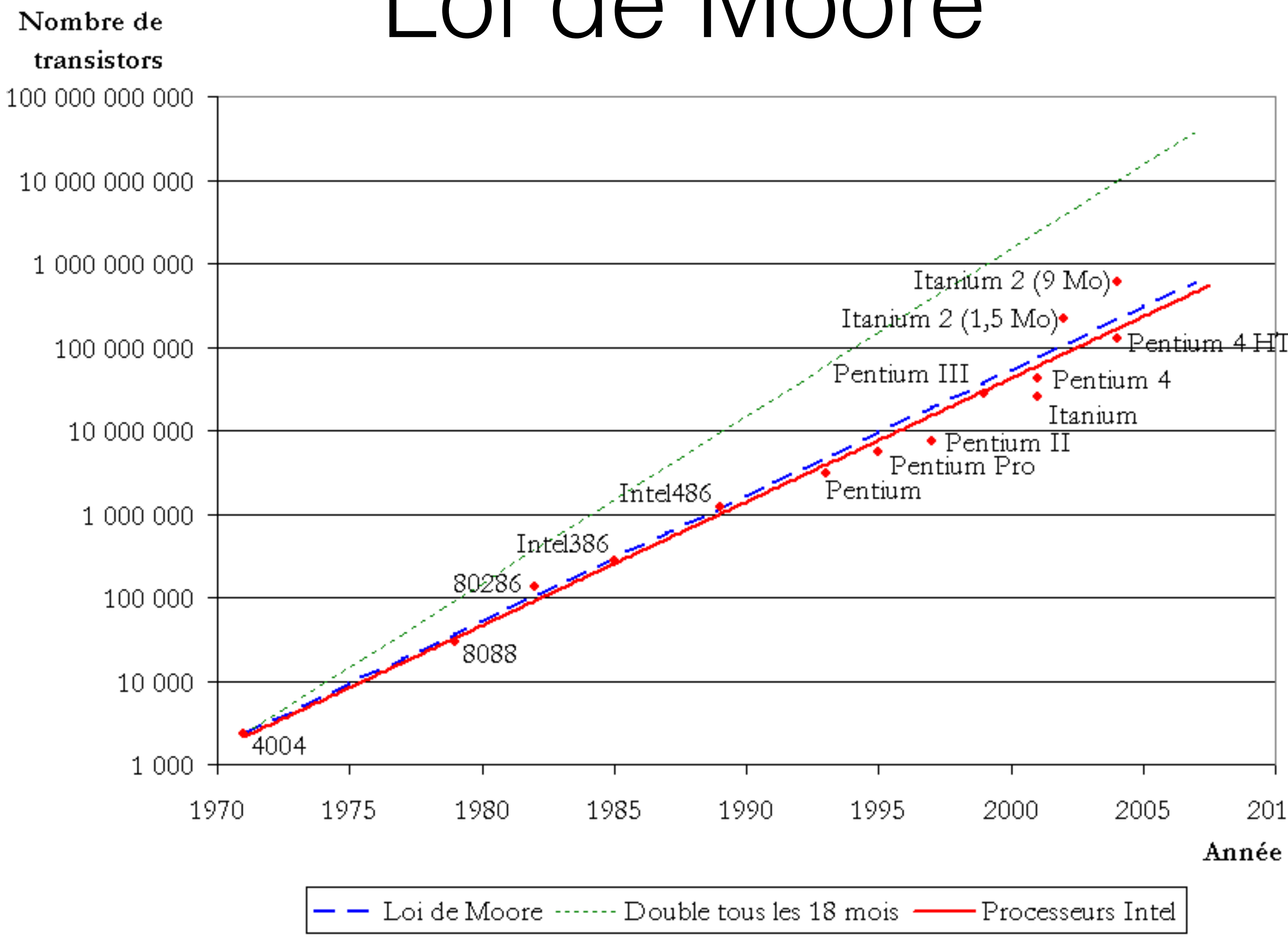
Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

Introduction

Loi de Moore

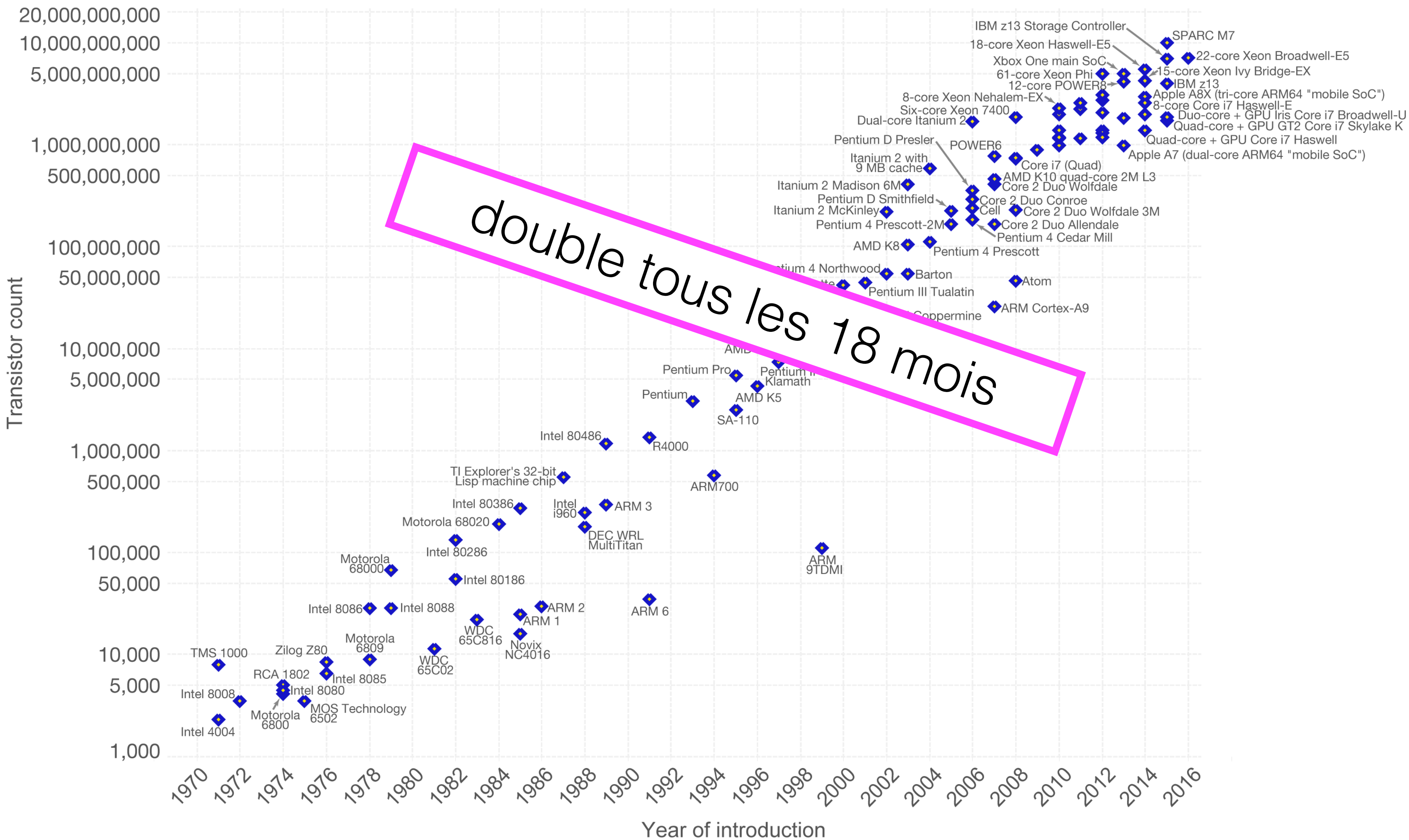


Nombre de transistors
100 000 000 000
10 000 000 000
1 000 000 000
100 000 000
10 000 000
1 000 000
100 000
10 000
1 000
1970
1975
1980
1985
1990
1995
2000
2005
2010
Année
Itanium 2 (9 Mo)
Itanium 2 (1,5 Mo)
Pentium 4 HT
Pentium III
Pentium 4
Itanium
Pentium II
Pentium Pro
Pentium
Intel486
Intel386
80286
8088
4004
Loi de Moore
Double tous les 18 mois
Processeurs Intel

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

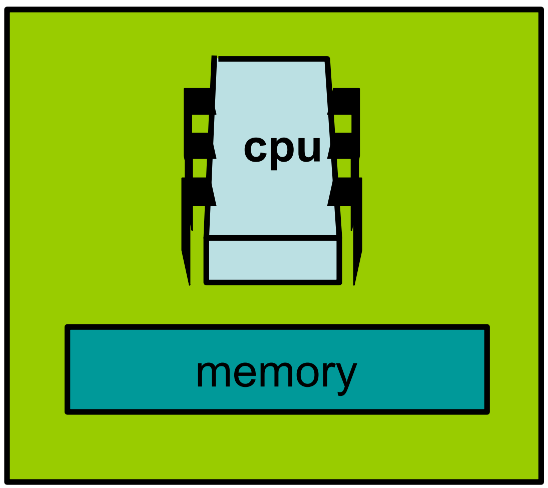
Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



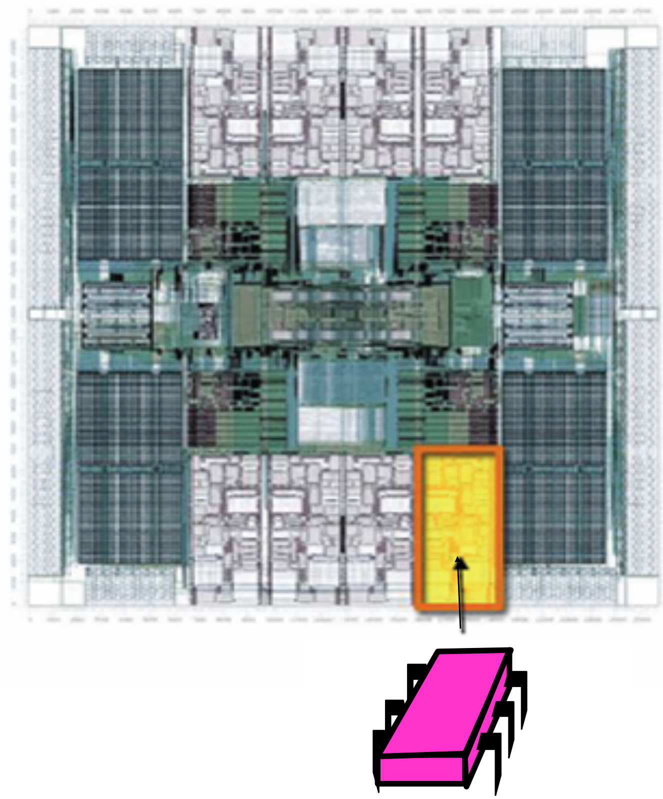
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Uniprocesseur

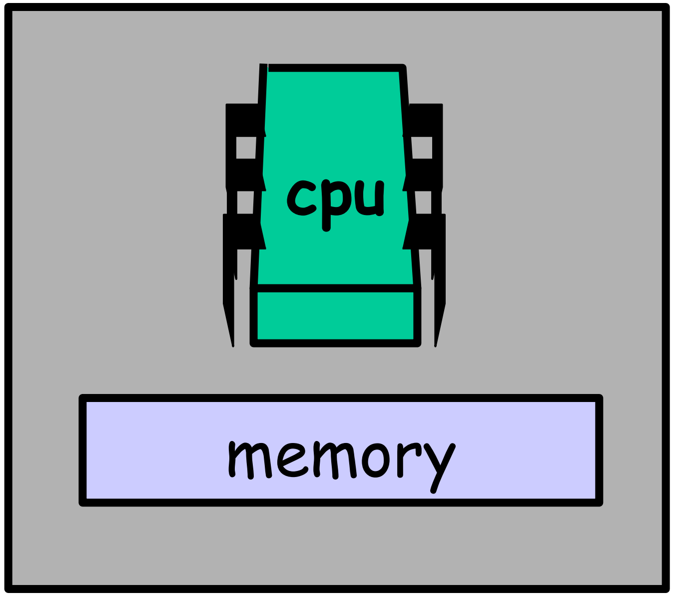


Mémoire partagée

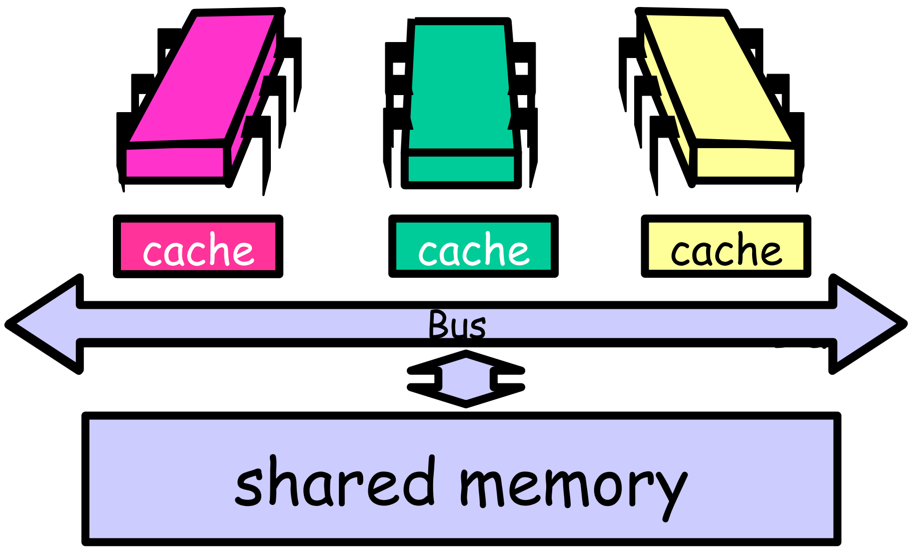
multicore



Still on some of your desktops: The Uniprocessor

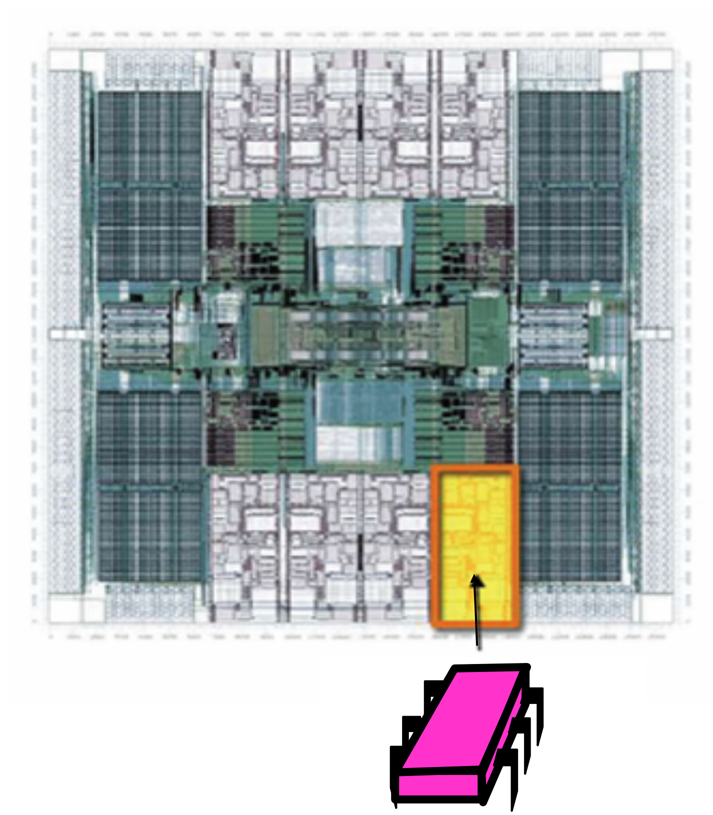


In the Enterprise: The Shared Memory Multiprocessor (SMP)



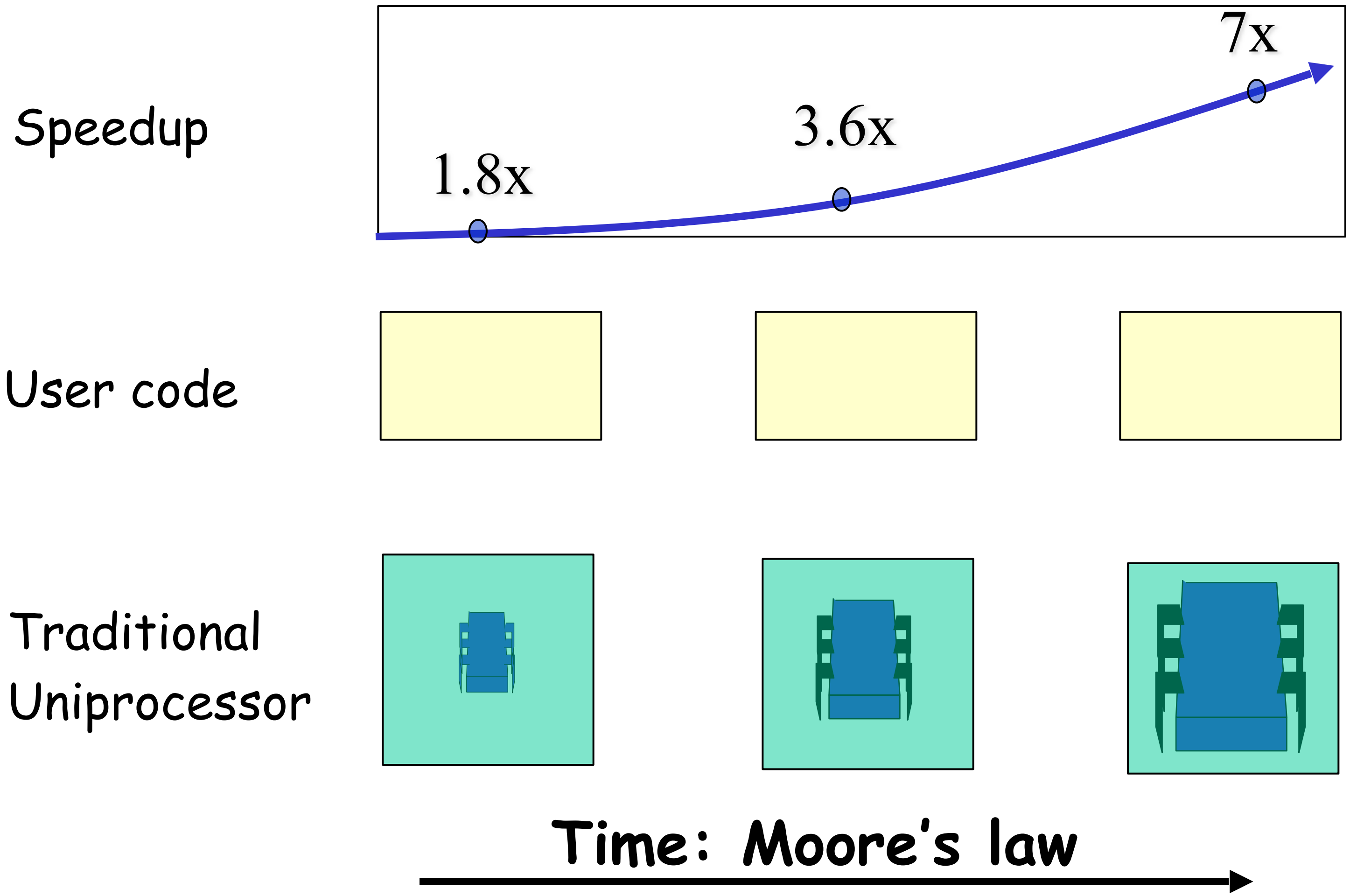
Your New Desktop:
The Multicore Processor
(CMP)

All on the same chip

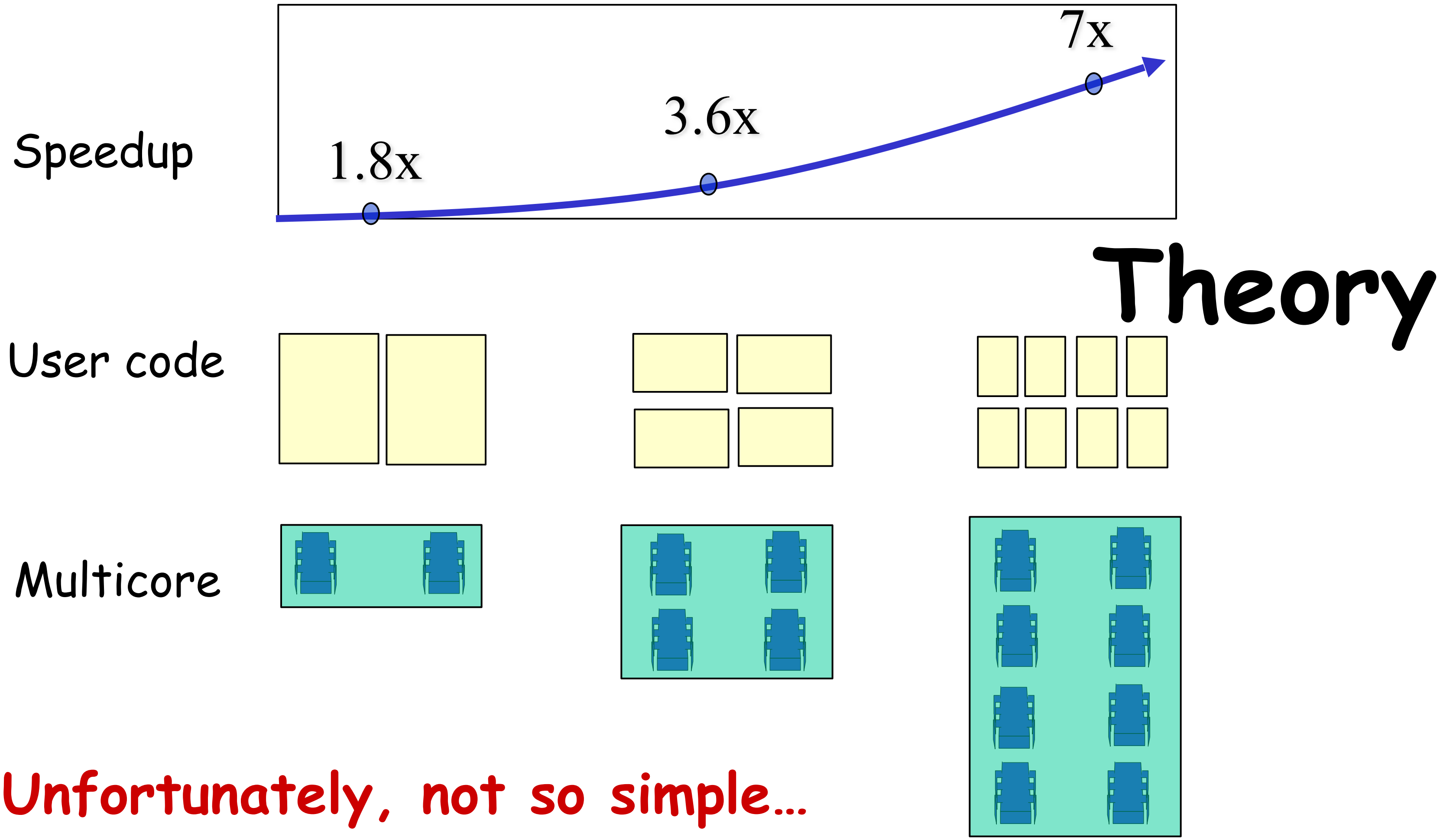


Sun T2000 Niagara

Traditional Scaling Process

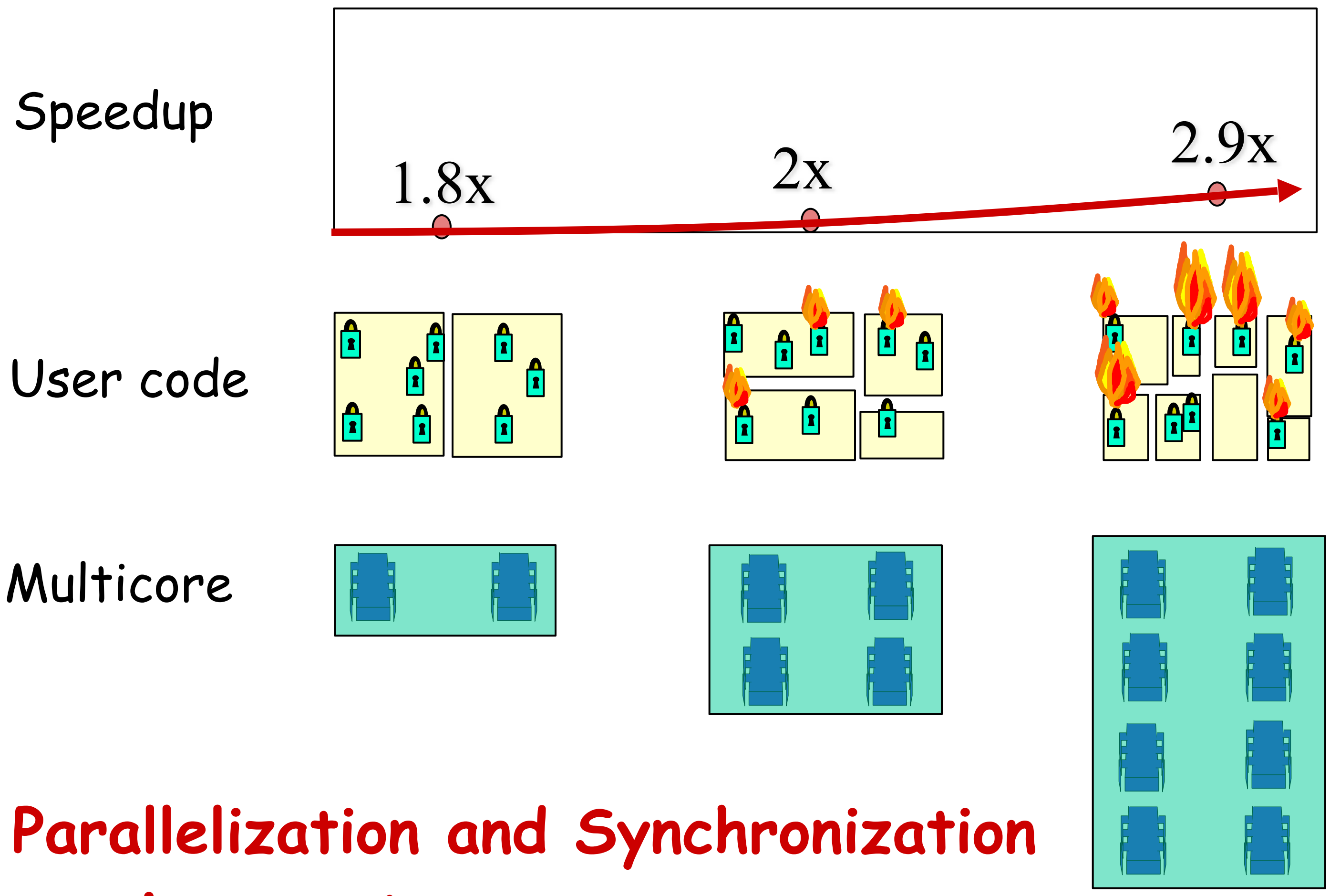


Multicore Scaling Process



Unfortunately, not so simple…

Real-World Scaling Process

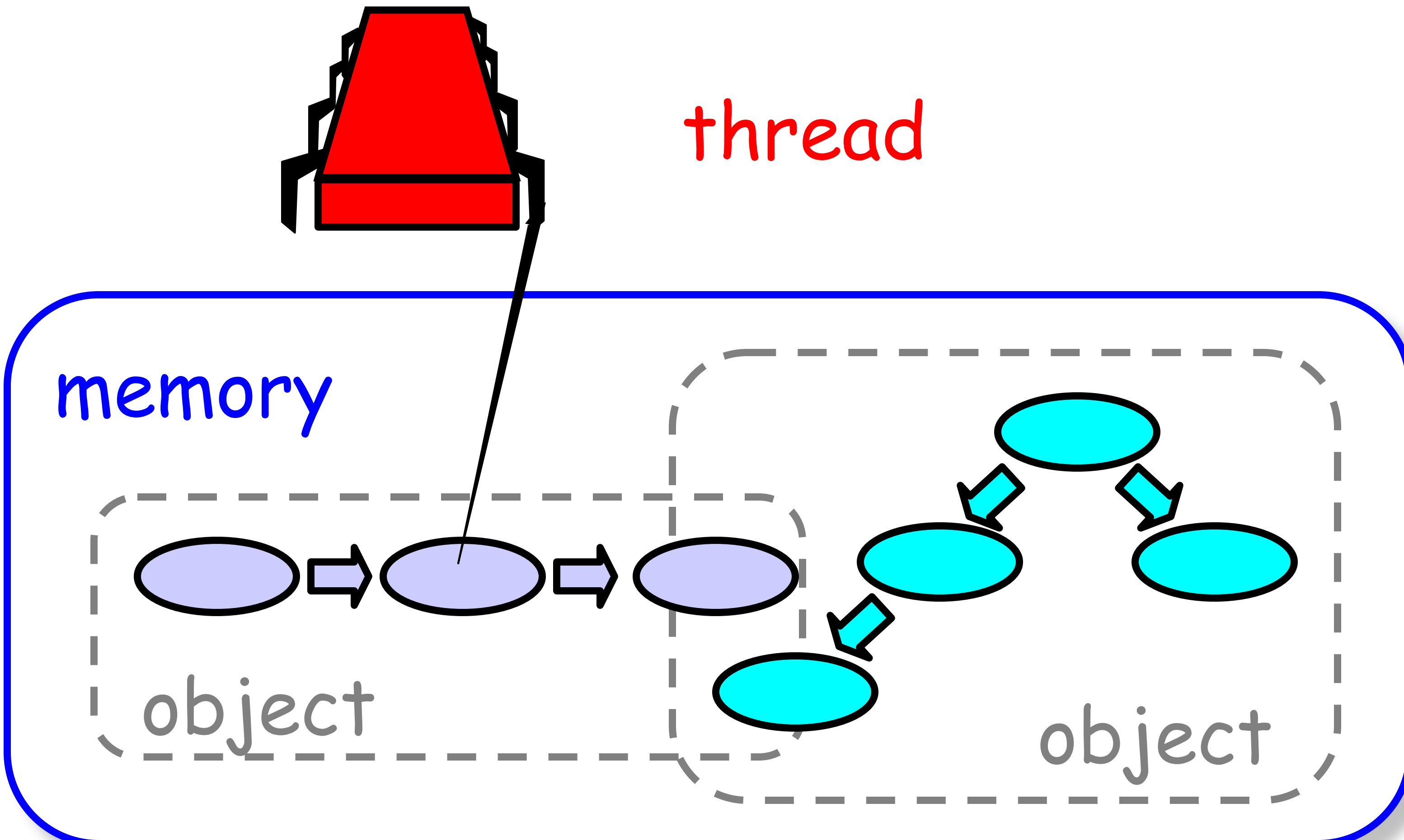


Parallelization and Synchronization require great care…

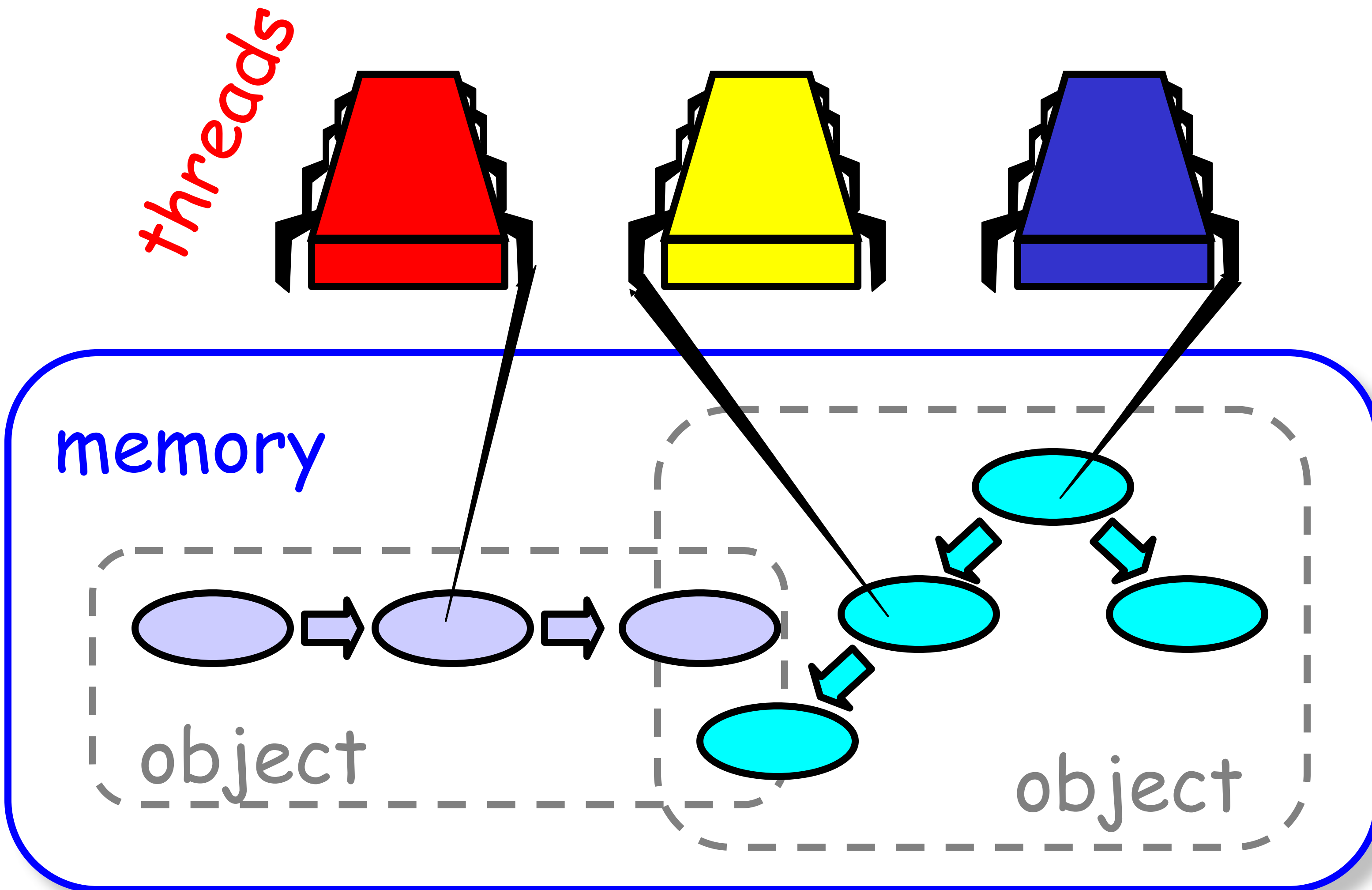
Multicore Programming: Course Overview

- Fundamentals
  - Models, algorithms, impossibility
- Real-World programming
  - Architectures
  - Techniques

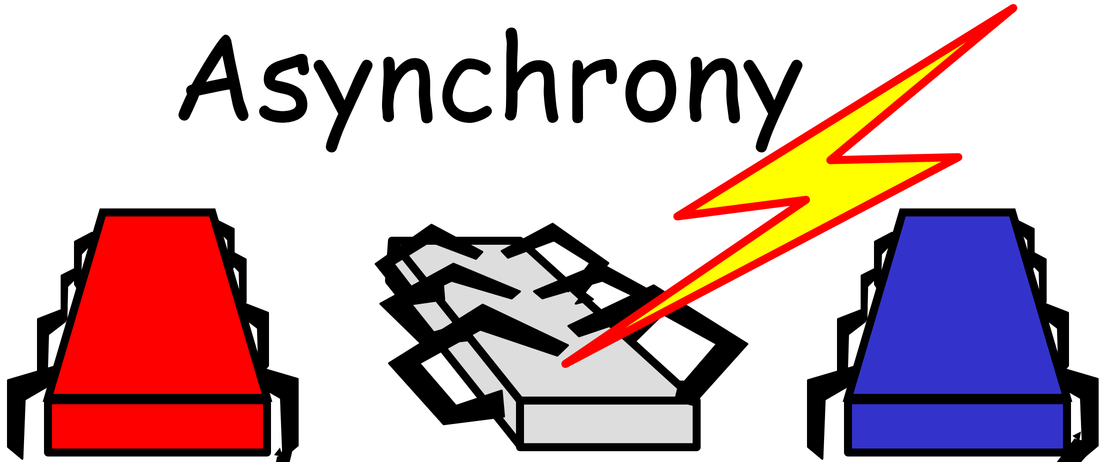
Sequential Computation



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
- Single shared memory
- Objects live in memory
- Unpredictable asynchronous delays

Road Map

- We are going to focus on principles first, then practice. We want to understand what we can and cannot compute before we try and write code.
  - Start with idealized models
  - Look at simplistic problems
  - Emphasize correctness over pragmatism
  - "Correctness may be theoretical, but incorrectness has practical impact"

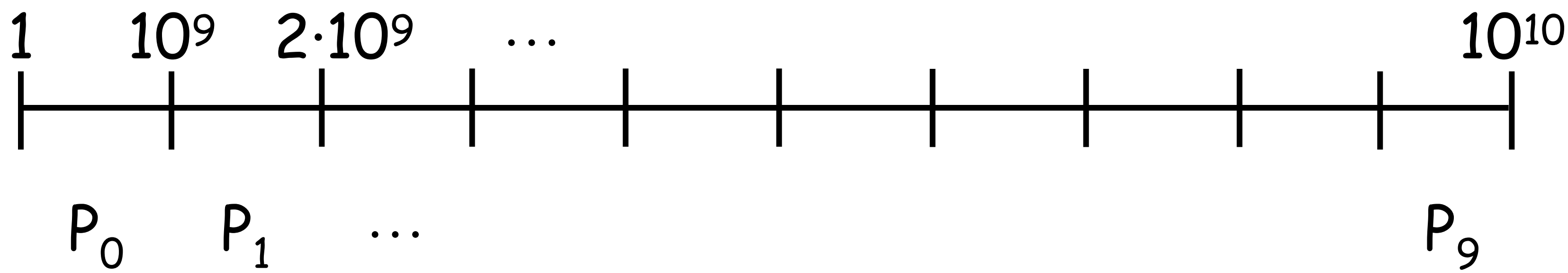
Concurrency Jargon

- Hardware
  - Processors
- Software
  - Threads, processes
- Sometimes OK to confuse them, sometimes not.

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 primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10^9+1, j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

```
public class ThreadID {
   private static volatile int nextID=0;
   private static class ThreadLocalID extends ThreadLocal<Integer>{
      protected synchronized Integer initialValue(){
         return nextID ++;
      }
      }
   private static ThreadLocalID threadID =new ThreadLocalID();
   public static int get(){
      return threadID.get();
   }
   public static void set (int index){
      threadID.set(index);

      }
}
```

Issues

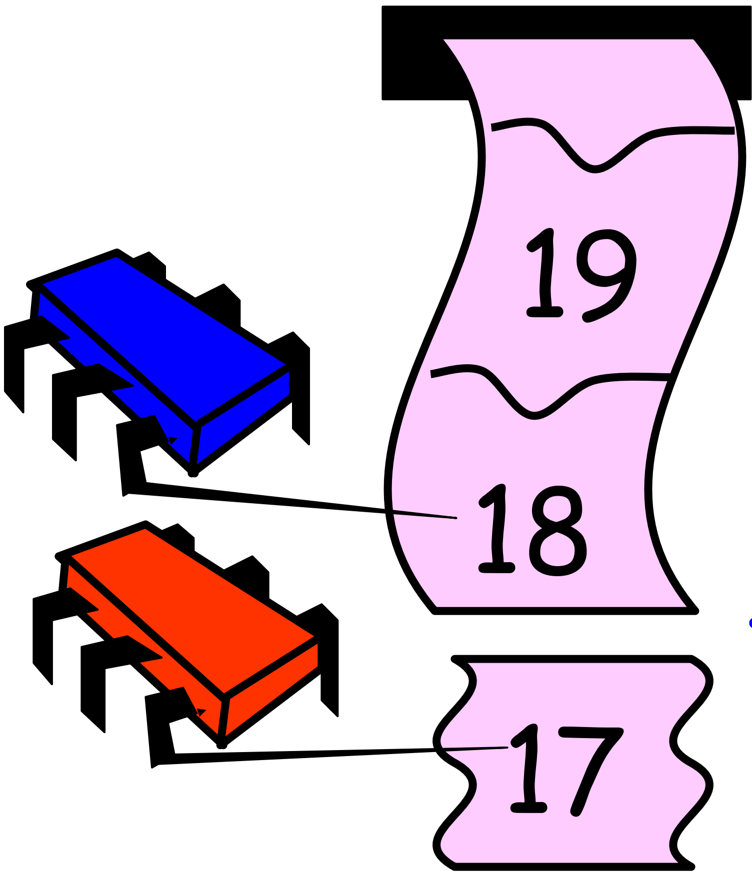
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

Issues

- Higher ranges have fewer primes
- Yet 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

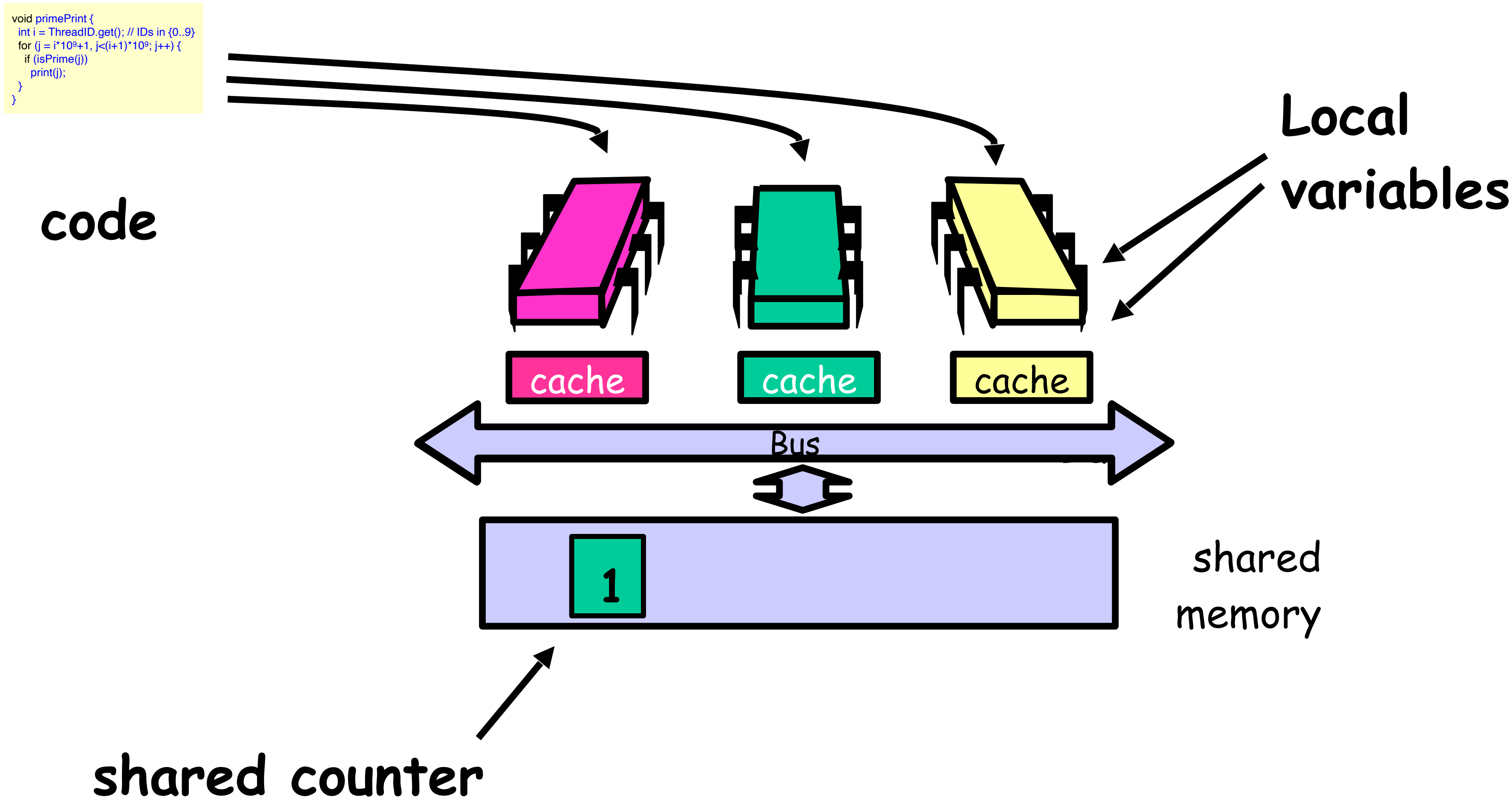
```
counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
        If (j < 10^10)
          if (isPrime(j))print(j);  }
}
```

Procedure for Thread i

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

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
      If (j < 10^10)
        if (isPrime(j))print(j);  }
}
```

Shared counter object

Where Things Reside



Procedure for Thread i

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

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    If (j < 10^10)
          if (isPrime(j))print(j);
  }
}
```

Stop when every value taken

Procedure for Thread i

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

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    If (j < 10^10)
          if (isPrime(j)) print(j);
  }
}
```

Increment & return each new value

Counter Implementation

```
public class Counter {
  private long value;

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

Counter Implementation

```
public class Counter {
  private long value;

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

OK for single thread,
not for concurrent threads

What It Means

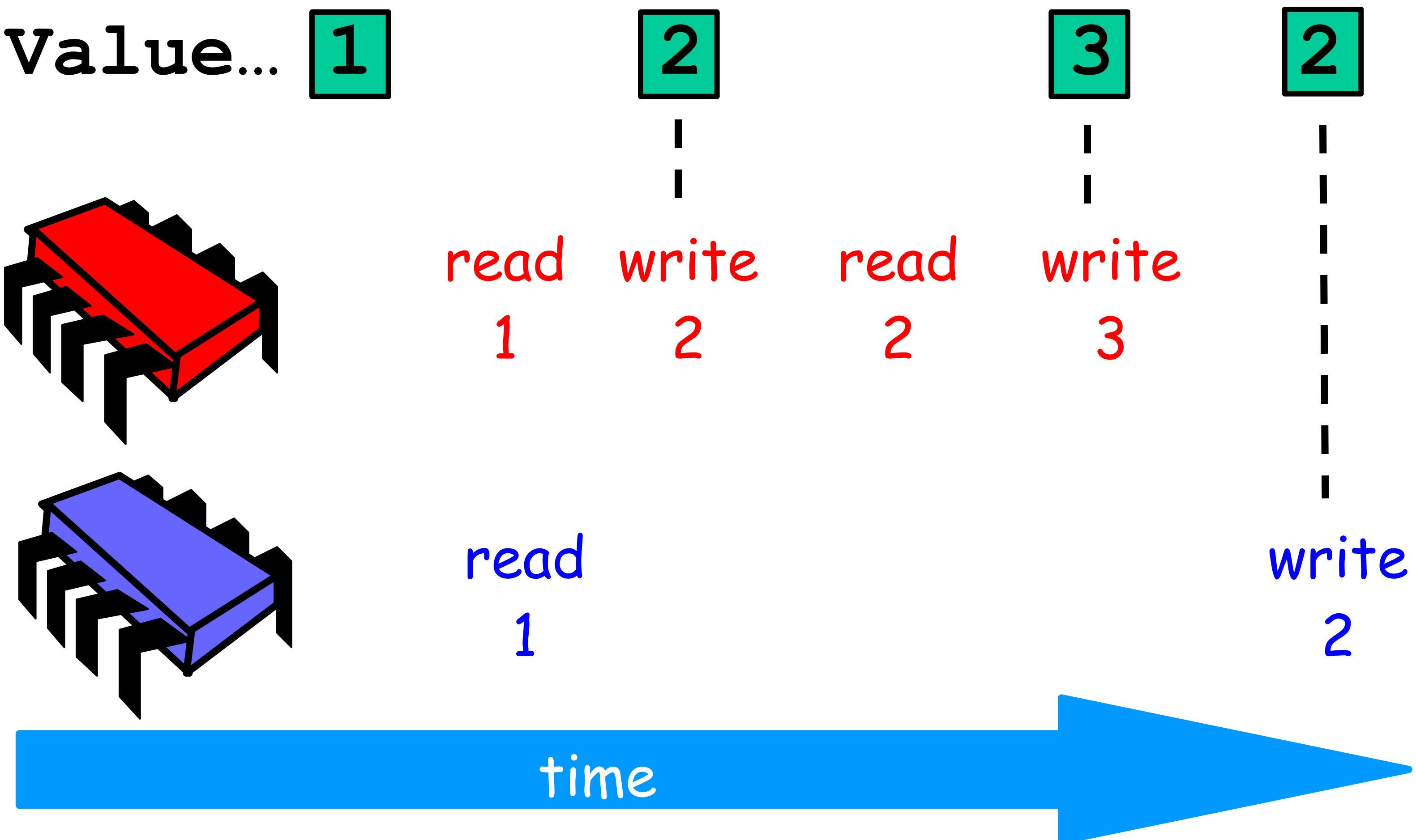
```
public class Counter {
  private long value;

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

What It Means

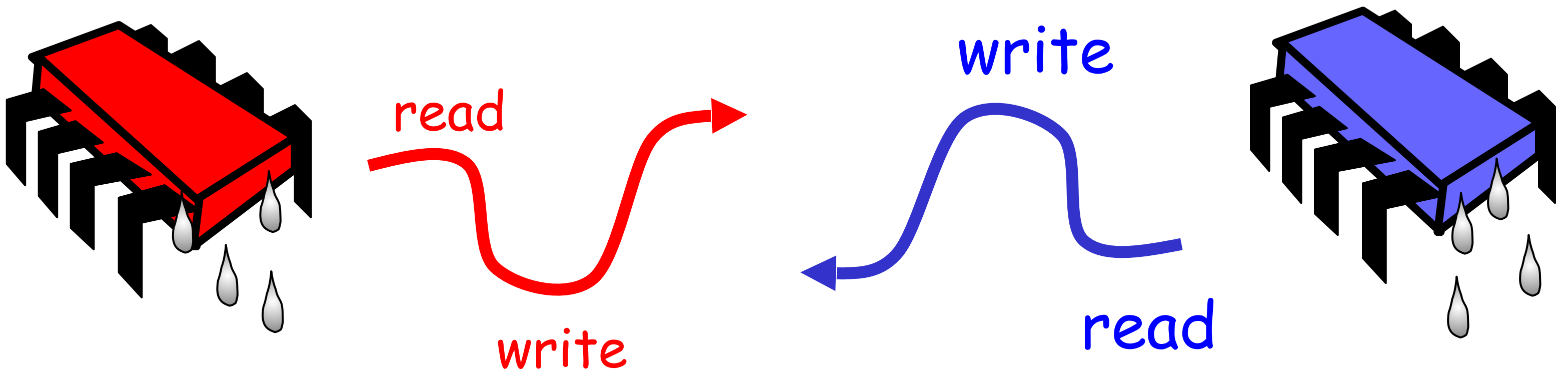
```
public class Counter {
  private long value;

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

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

Not so good…



Is this problem inherent?



If we could only glue reads and writes…

Challenge

```
public class Counter {
  private long value;

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

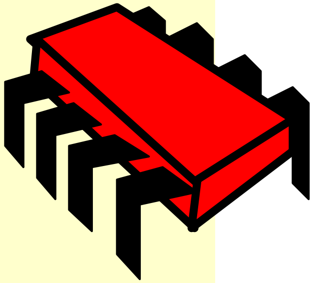
Challenge

```
public class Counter {
  private long value;

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

Make these steps atomic (indivisible)

Hardware Solution

```
public class Counter {
  private long value;

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



ReadModifyWrite() instruction

An Aside: Java™

```
public class Counter {
  private long value;

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

An Aside: Java™

```
public class Counter {
  private long value;

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

Synchronized block

An Aside: Java™

```
public class Counter {
  private long value;

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

Mutual Exclusion

Formalizing the Problem

- Two types of formal properties in asynchronous computation:
- Safety Properties
  - Nothing bad happens ever
- Liveness Properties
  - Something good happens eventually

Formalizing our Problem

- Mutual Exclusion
  - never simultaneously
  - This is a **safety** property
- No Deadlock
  - if only one wants in, it gets in
  - if both want in, one gets in.
  - This is a **liveness** property

Formalizing our Problem

- No starvation (many flavours)
  - if one wants in, it eventually gets in
  - if one wants in, it gets in after at most n processes enter in sc before it

  - This is a **liveness** property

- Mutual Exclusion
- Producers/Consumers
- Readers/Writers

- Waiting

- Concurrent data structure
  - specification?
  - implementation non blocking, wait free ….

Why do we care?

- We want as much of the code as possible to execute concurrently (in parallel)
- A larger sequential part implies reduced performance
- Amdahl's law: this relation is not linear…

Amdahl's Law

$$\text{Speedup} = \frac{\textbf{Temps séquentiel}}{\textbf{Temps concurrent}}$$

…of computation given **n** CPUs instead of **1**

Amdahl's Law

(normalized) time 1 for a single processor to complete the code

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

Amdahl's Law

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

Parallel fraction

Amdahl's Law

Sequential fraction

Parallel fraction

$$\text{Speedup} = \frac{1}{\boxed{1-p} + \frac{\boxed{p}}{n}}$$

Amdahl's Law

Sequential fraction

Parallel fraction

$$\text{Speedup}=\frac{1}{\boxed{1-p}+\frac{\boxed{p}}{\boxed{n}}}$$

Number of processors

Example

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

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?

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?

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?

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

- Making good use of our multiple processors (cores) means
- Finding ways to effectively parallelize our code
  - Minimize sequential parts
  - Reduce idle time in which threads **wait** without

Multicore Programming

- This is what this course is about…
  - The % that is not easy to make concurrent yet may have a large impact on overall speedup

Plan du cours

» Une partie théorique et une partie pratique

» Java - Threads

» Exclusion mutuelle

» Objets concurrents- correction et progression

» Java - java.util.concurrent.atomic

» Objets universels

» Implementation-Performance

Why is Concurrent Programming so Hard?

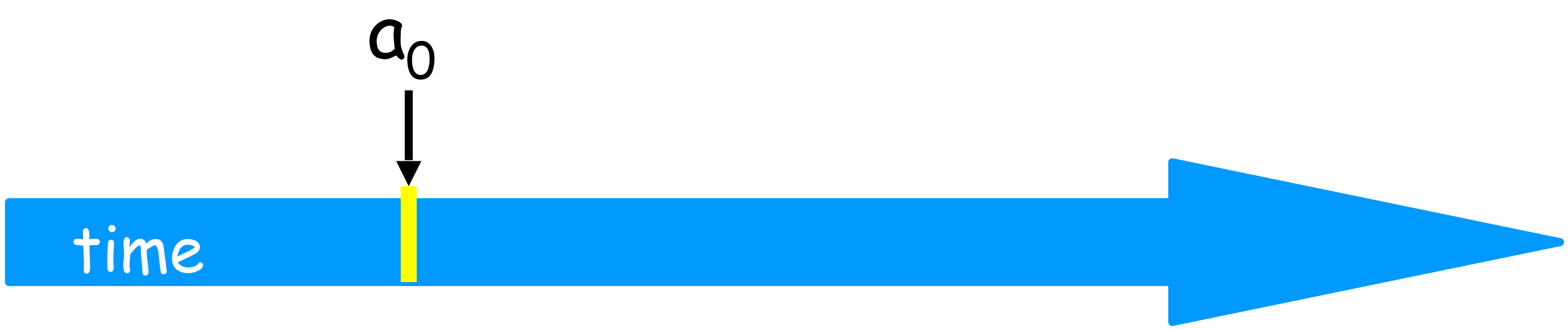
- Try preparing a seven-course banquet
  - By yourself
  - With one friend
  - With twenty-seven friends …
- Before we can talk about programs
  - Need a language
  - Describing time and concurrency

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, like, Nature's way of making sure that everything doesn't happen all at once." (Anonymous, circa 1968)

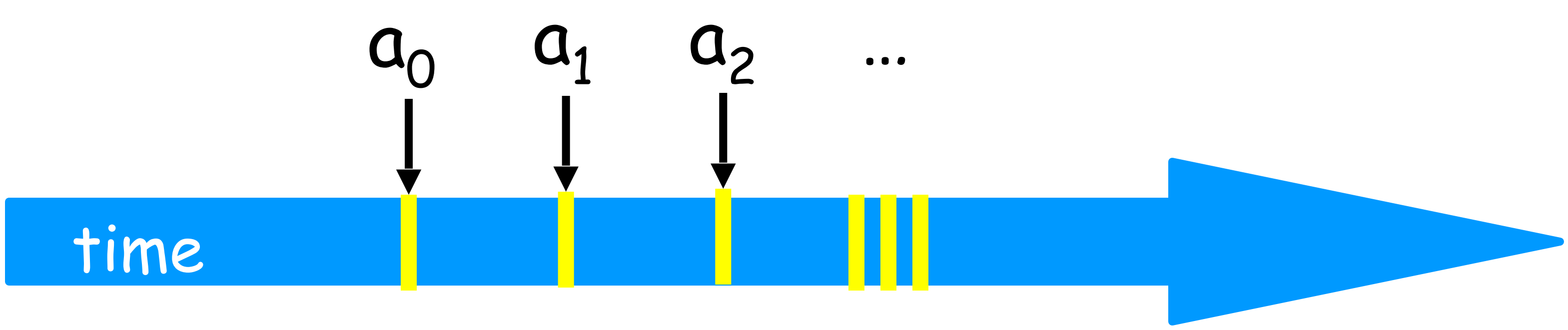
Events

- 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

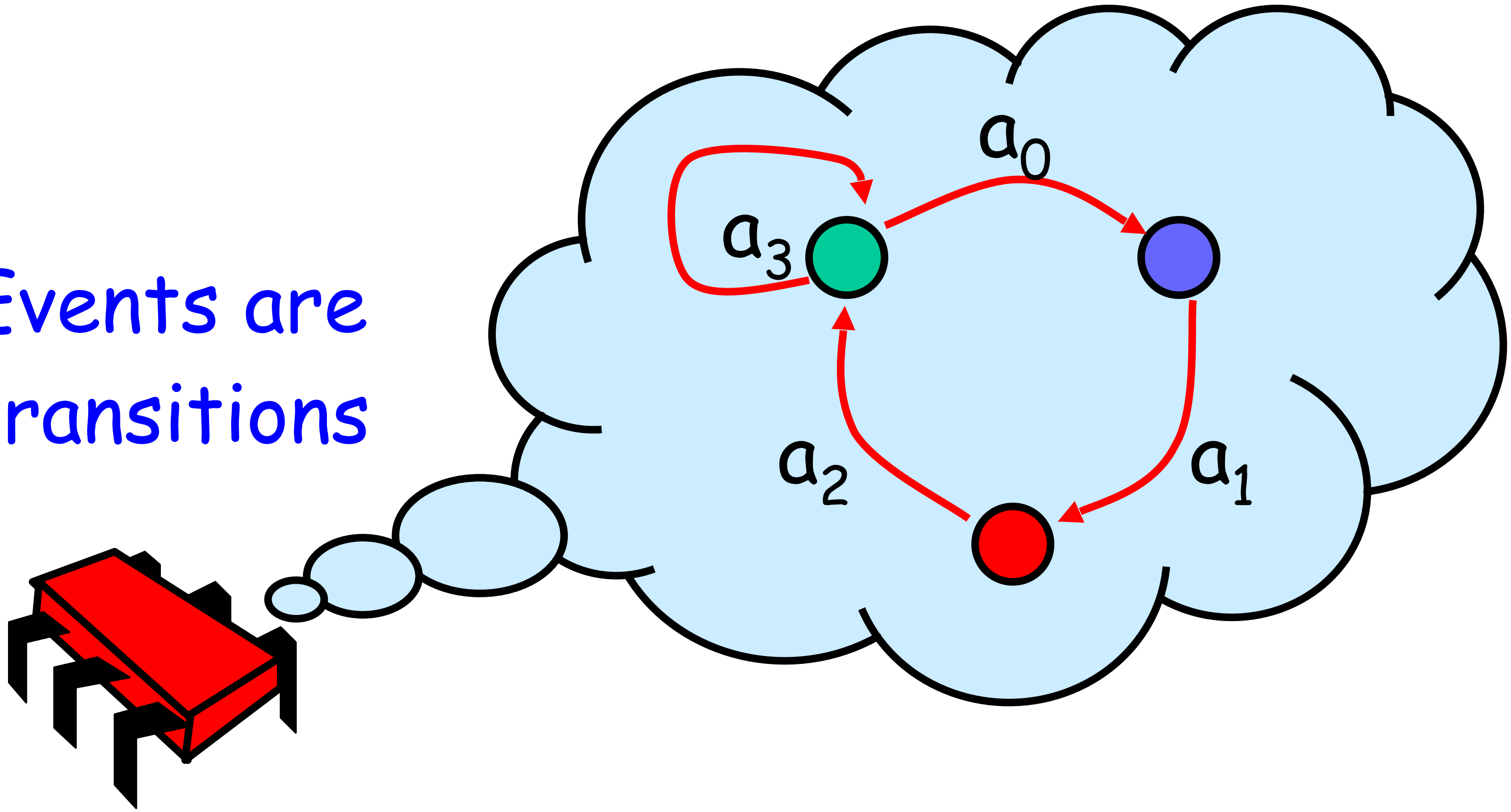


Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …

Threads are State Machines

Events are transitions



States

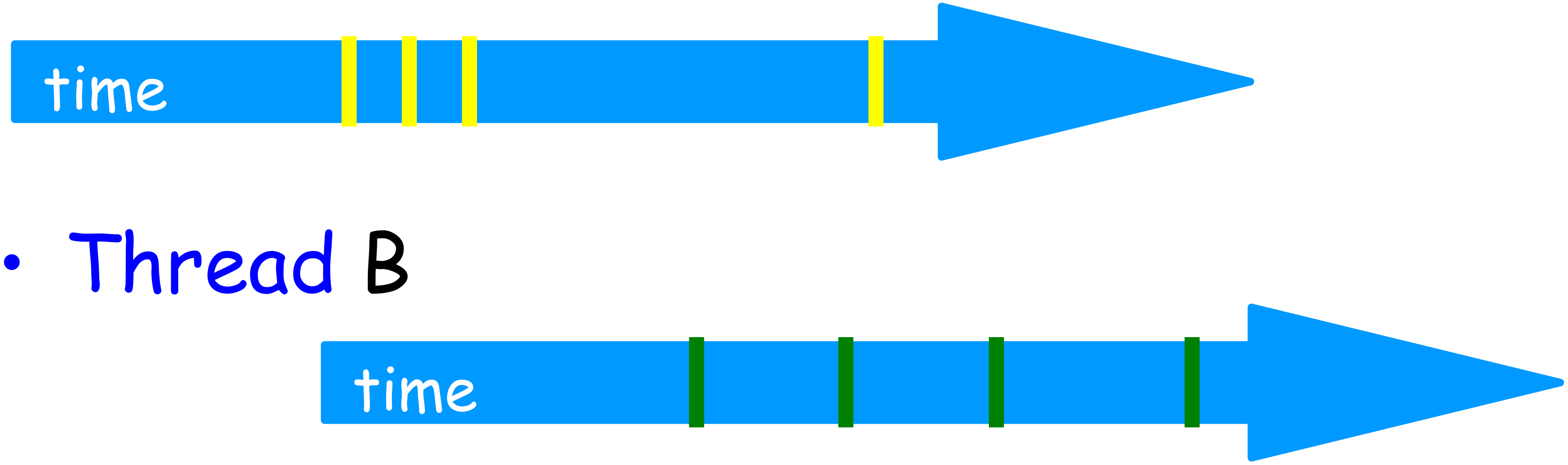
- Thread State
  - Program counter
  - Local variables
- System state
  - Object fields (shared variables)
  - Union of thread states

Concurrency

- Thread *A*

Concurrency

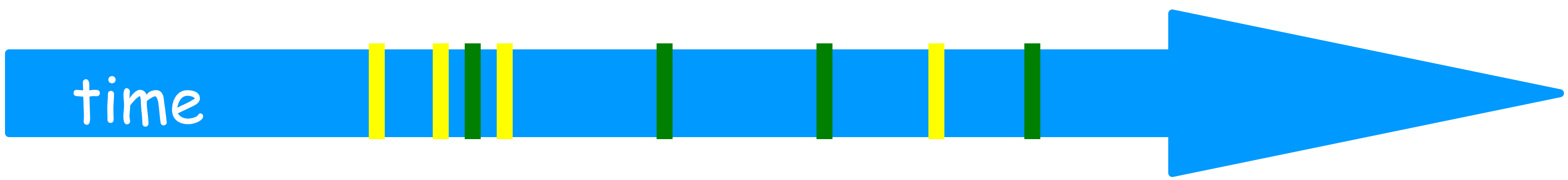
- Thread A



- Thread B

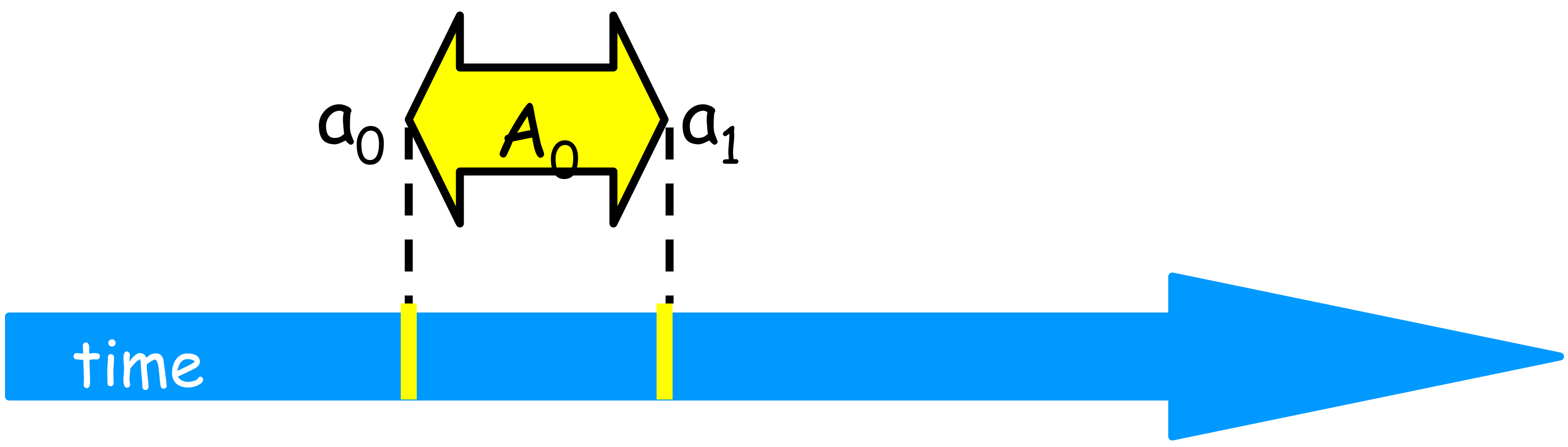
Interleavings

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

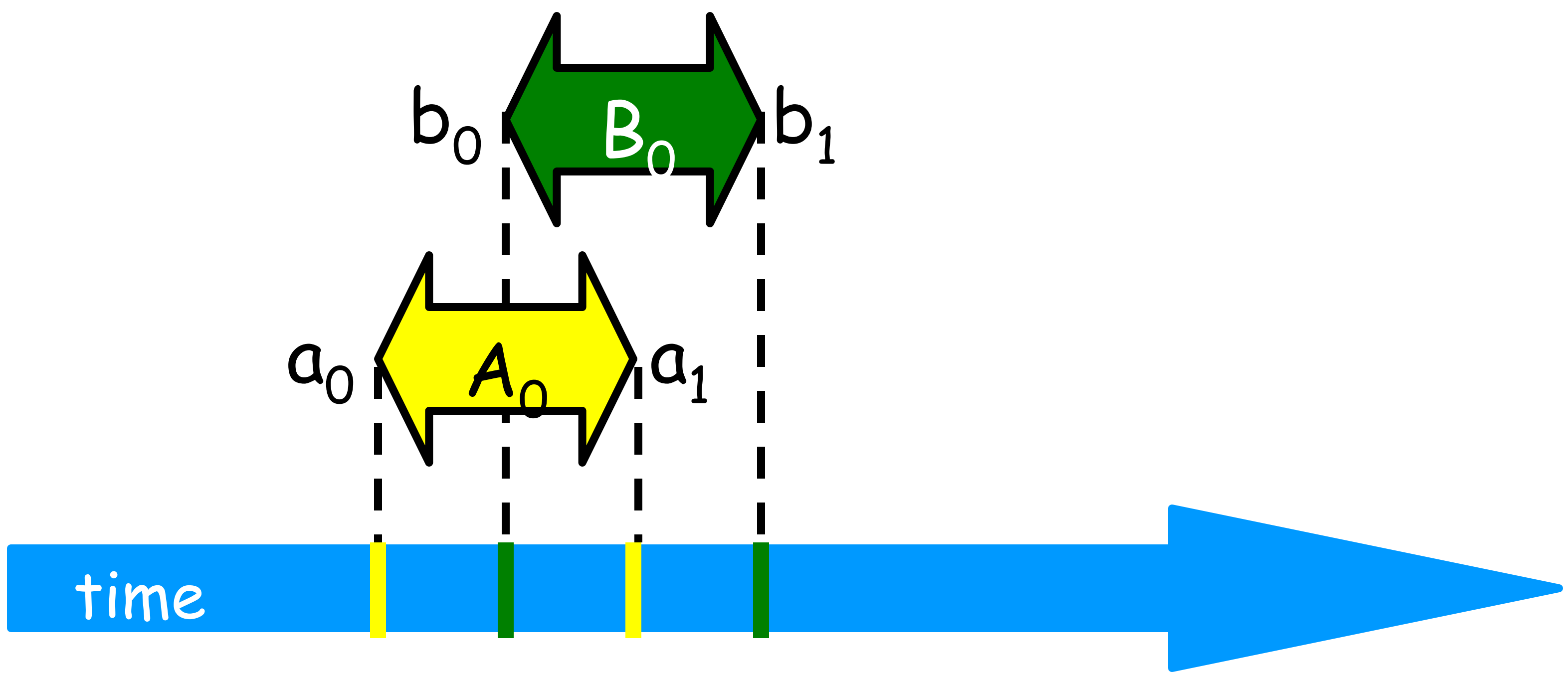


Intervals

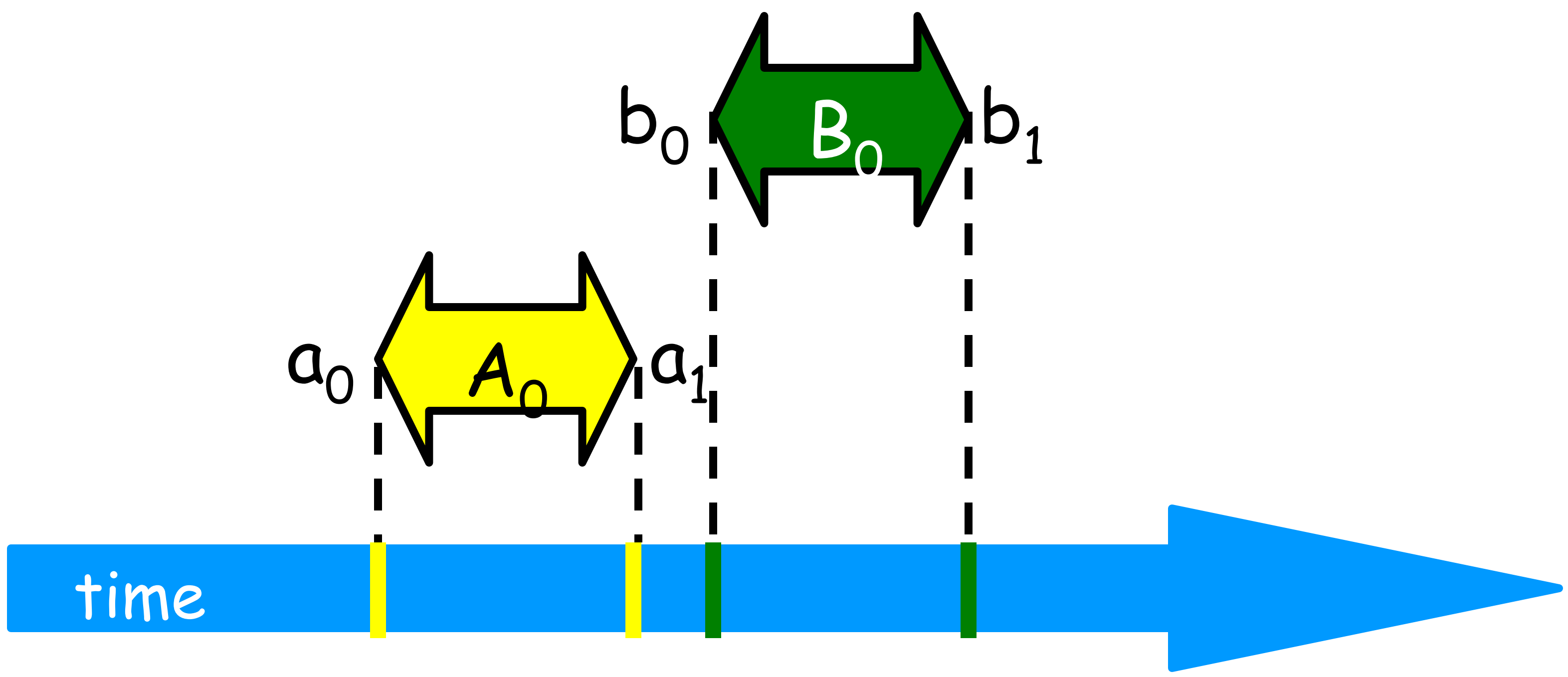
- An interval $A_0 = (a_0, a_1)$ is
  - Time between events a_0 and a_1



Intervals may Overlap

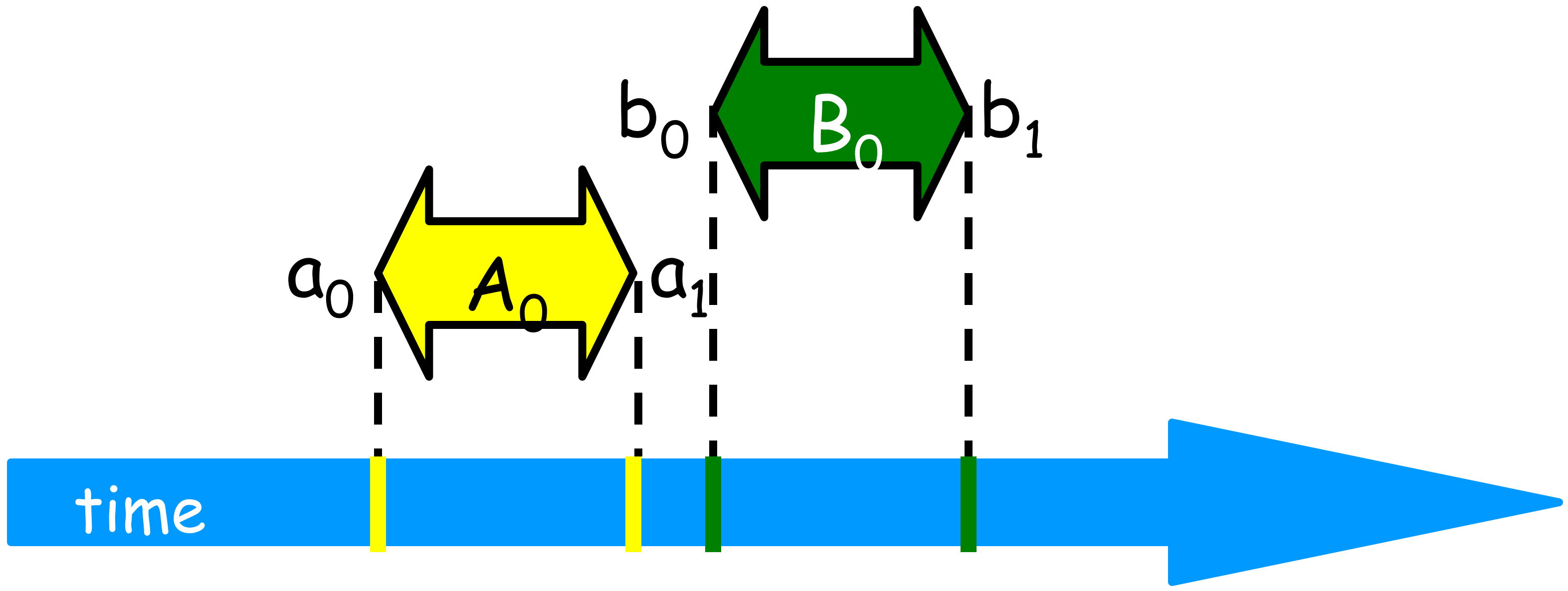


Intervals may be Disjoint

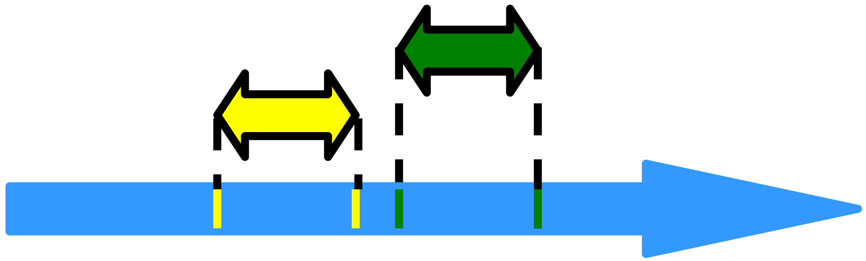


Precedence

Interval A_0 precedes interval B_0

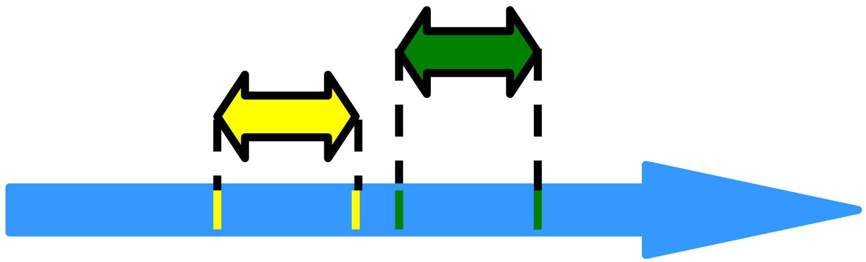


Precedence



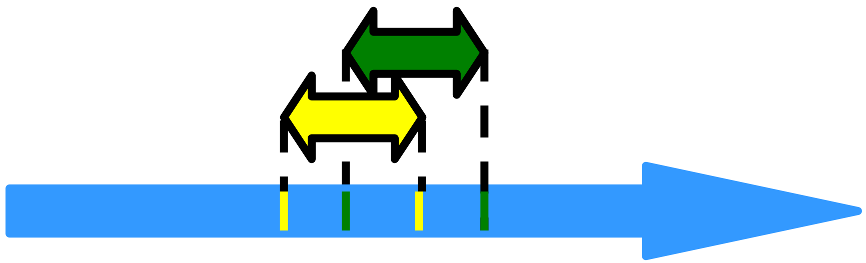
- Notation: $A_0 \rightarrow B_0$
- Formally,
  - End event of A_0 before start event of B_0
  - Also called "happens before" or "precedes"

Precedence Ordering



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

Repeated Events

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

k-th occurrence of event a_0

a_0^k

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

A_0^k

Implementing a Counter

```
public class Counter {
  private long value;

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

Make these steps indivisible using 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;
  }}
```

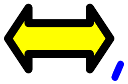
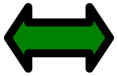
Critical section

Mutual Exclusion

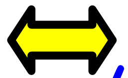
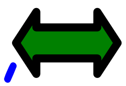
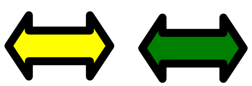
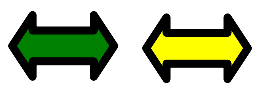
Mutual Exclusion

- Let CS_i^k, $\Leftrightarrow$, be thread i's k-th critical section execution

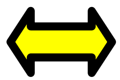
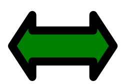
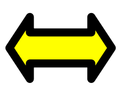
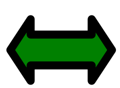
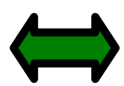
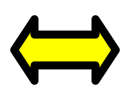
Mutual Exclusion

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

Mutual Exclusion

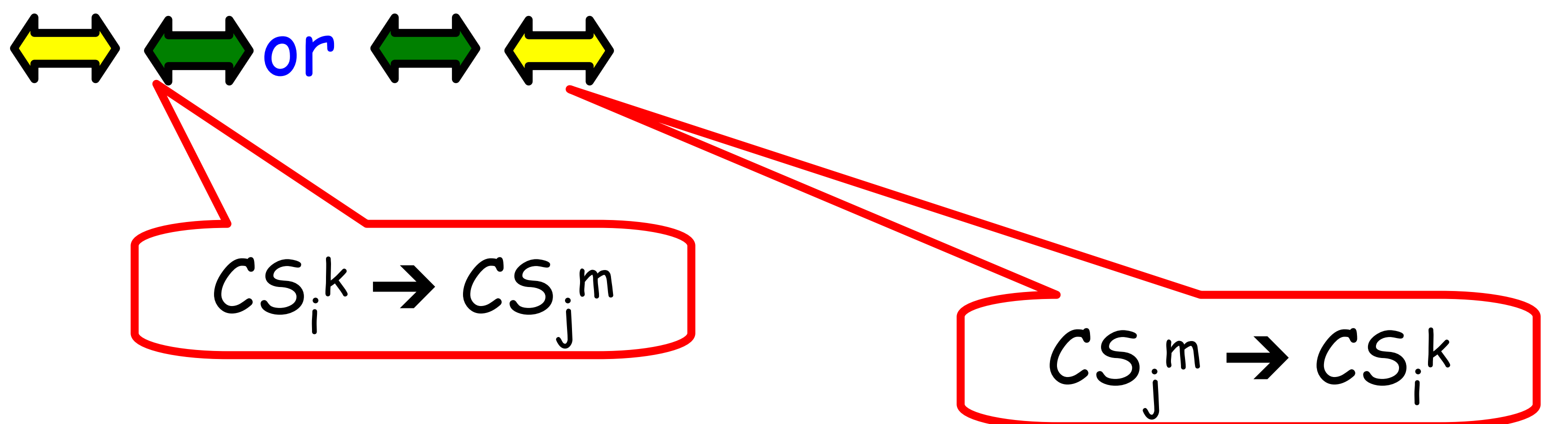
- Let CS_i^k, , be thread i's k-th critical section execution
- And CS_j^m, , be j's m-th execution
- Then either
  -  or 

Mutual Exclusion

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

$CS_i^k \rightarrow CS_j^m$

Mutual Exclusion

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

Deadlock-Free

- If some thread 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

Starvation-Free

- If some thread calls lock()
  - It will eventually return
- Individual threads make progress

Two-Thread vs n -Thread Solutions

- Two-thread solutions first
  - Illustrate most basic ideas
  - Fits on one slide
- Then n-Thread solutions

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

JAVA like

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

Set my flag

LockOne

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

Set my flag

Wait for other flag to go false

LockOne Satisfies Mutual Exclusion

- Assume CS_A^j overlaps CS_B^k
- Consider each thread's last (j-th and k-th) read and write in the lock() method before entering
- Derive a contradiction

From the Code

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

- $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false) \rightarrow CS_A$
- $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false) \rightarrow CS_B$

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

- $\text{read}_A(\text{flag[B]==false}) \rightarrow \text{write}_B(\text{flag[B]=true})$

- $\text{read}_B(\text{flag[A]==false}) \rightarrow \text{write}_A(\text{flag[B]=true})$

Combining

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

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

Combining

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

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

Combining

- Assumptions:
  - $\text{read}_A(\text{flag[B]==false}) \rightarrow \text{write}_B(\text{flag[B]=true})$
  - $\text{read}_B(\text{flag[A]==false}) \rightarrow \text{write}_A(\text{flag[A]=true})$
- From the code
  - $\text{write}_A(\text{flag[A]=true}) \rightarrow \text{read}_A(\text{flag[B]==false})$
  - $\text{write}_B(\text{flag[B]=true}) \rightarrow \text{read}_B(\text{flag[A]==false})$

Combining

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

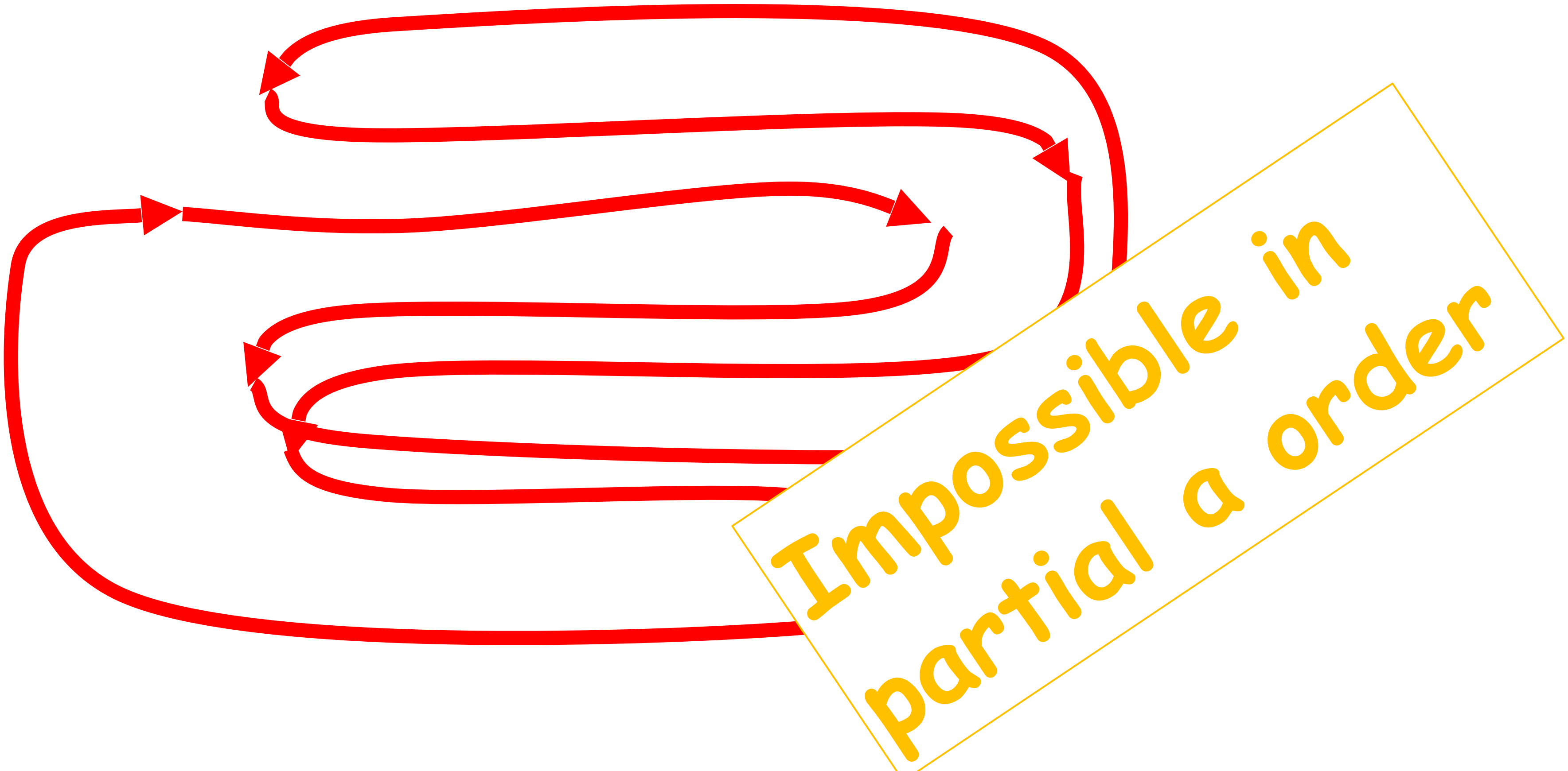
Combining

- Assumptions:
  - $\text{read}_A(\text{flag}[B]==\text{false}) \rightarrow \text{write}_B(\text{flag}[B]=\text{true})$
  - $\text{read}_B(\text{flag}[A]==\text{false}) \rightarrow \text{write}_A(\text{flag}[A]=\text{true})$
- From the code
  - $\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B]==\text{false})$
  - $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_B(\text{flag}[A]==\text{false})$

Combining

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

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
  - Cannot be both 0 and 1

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

- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

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

From the Code

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

- $\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow$

  $\text{write}_A(\text{victim}=A) \rightarrow$

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

  $\text{read}_A(\text{victim}) \rightarrow \text{CS}_A$

- Idem for B

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

- **(*A* was the last thread to write to the victim field)**

- **$\text{write}_B(\text{victim}=B) \rightarrow \text{write}_A(\text{victim}=A)$**

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

- $write_B(flag[B]=true) \rightarrow$ (code)

  $write_B(victim=B) \rightarrow$ (hyp ordre)

  $write_A(victim=A) \rightarrow$ (A en CS)

  $read_A(flag[B]=false)$

- Contradict: no other write to Flag[B] was performs

  before the Cs exec

Starvation 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;
}
```

Deadlock Free

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

- Thread blocked
  - only at while loop
  - only if it is the victim
- One or the other must not be the victim

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

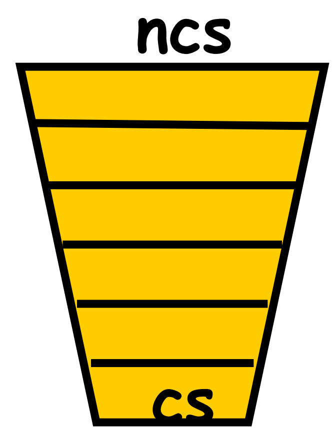
Pas d'atomicité du &&

évaluation des 2 variables indépendamment

The Filter Algorithm for n Threads

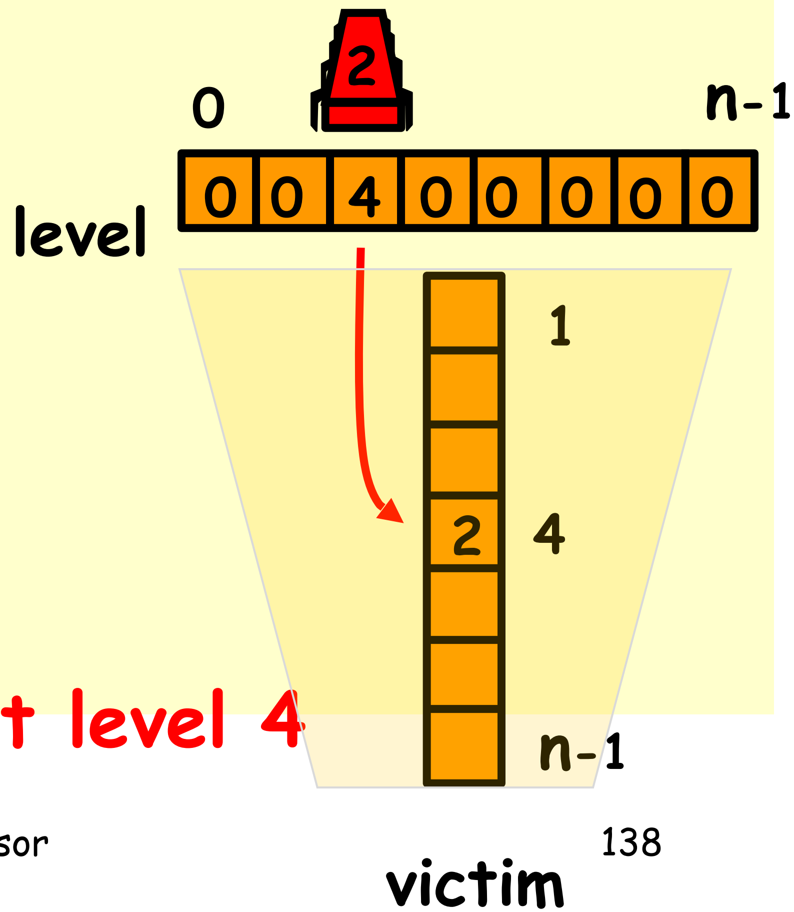
There are **n-1** "waiting rooms" called levels

- 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 = 0; i < n; i++) {
         level[i] = 0;
      }}
     …
}
```



Thread 2 at level 4

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

```
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;
}}
```

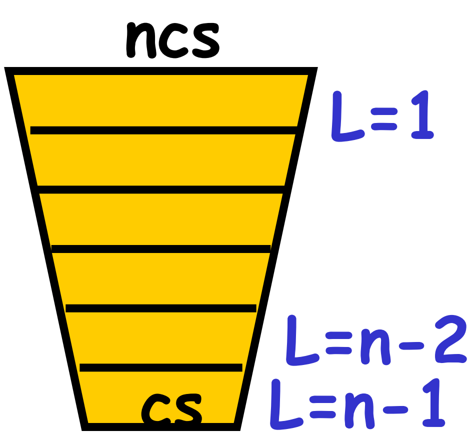
Filter

```
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);
  }}
```

Thread enters level L when it completes the loop

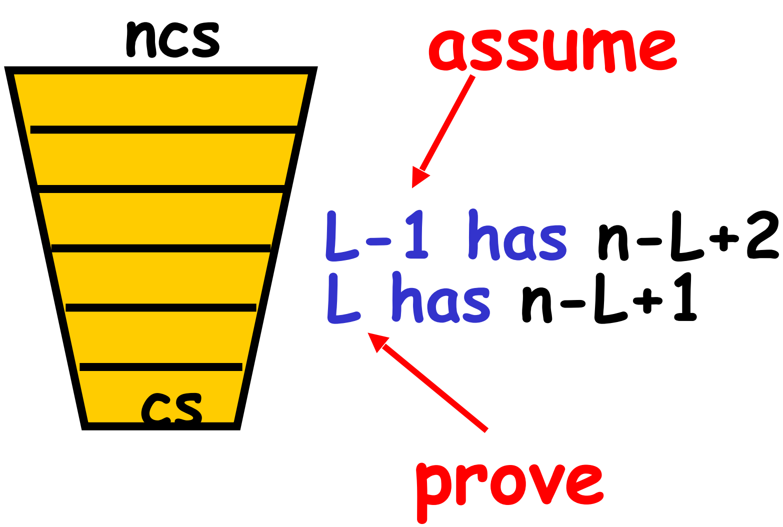
Claim

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

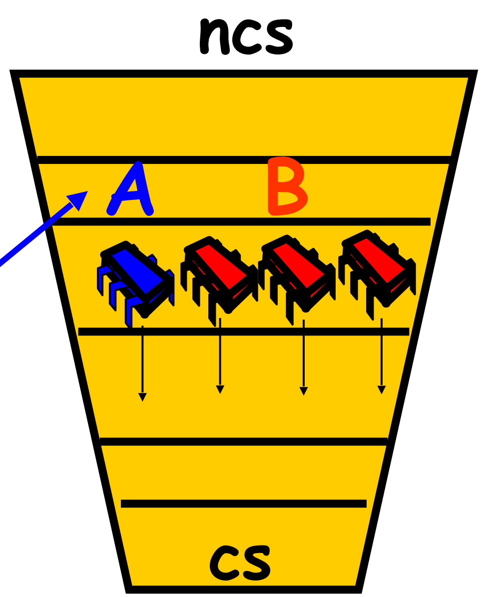


Induction Hypothesis

- No more than n-L+1 at level L
- Induction step: by contradiction
- By ind: no more than n-(L-1)+1 threads at level L-1
- A last to write victim[L]
- B is any other thread at level L



Proof Structure



Assumed to enter L-1

$n-L+2 = 4$

$n-L+2 = 4$

Last to write victim[L]

By way of contradiction all enter L

Show that A must have seen B in level[L] and since victim[L] == A could not have entered

From the Code

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

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

From the Code

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

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

By Assumption

(3) $write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$

By assumption, *A* is the last thread to write **victim[L]**

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

Combining Observations

(1) $write_B(level[B]=L) \rightarrow write_B(victim[L]=B)$

(3) $write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$

(2) $write_A(victim[L]=A) \rightarrow read_A(level[B])$

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

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

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

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

Thus, A read level[B] ≥ L,

A was last to write victim[L],

so it could not have entered level L!

No Starvation

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

Bounded Waiting

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

Bounded Waiting

- Divide lock() method into 2 parts:
  - Doorway interval:
    - Written D_A
    - always finishes in finite steps
  - Waiting interval:
    - Written 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

- Filter Lock satisfies properties:
  - No one starves
  - But very weak fairness

Bakery Algorithm

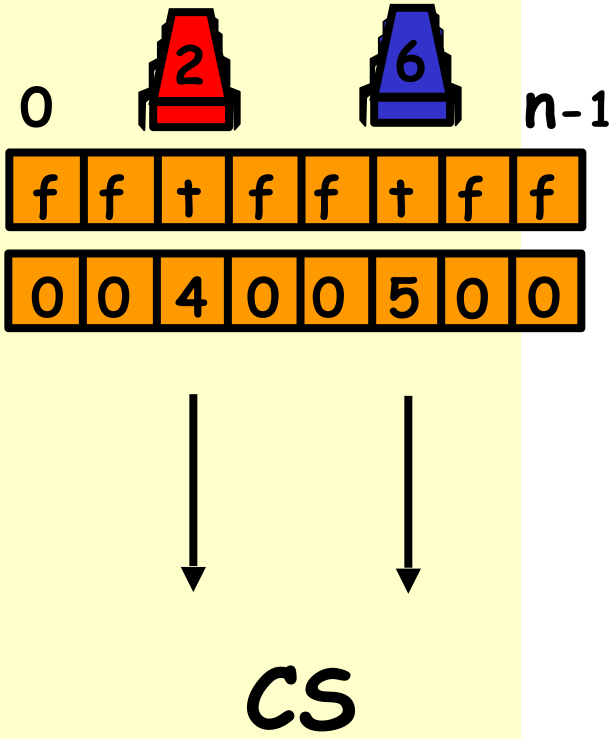
- Provides First-Come-First-Served
- How?
  - Take a "number"
  - Wait until lower numbers have been served
- Lexicographic order
  - $(a,i) > (b,j)$
    - If $a > b$, or $a = b$ and $i > j$

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

```
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

```
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));
 }
```

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[i],i) > (label[k],k));
 }
```

Doorway

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[i],i) > (label[k],k));
 }
```

I'm interested

Bakery Algorithm

Take increasing label (read labels in some arbitrary order)

```
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));
 }
```

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[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])+1;
  while (∃k flag[k]
          && (label[i],i) > (label[k],k));
 }
```

Someone is interested

With lower (label,i) in lexicographic order

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

labels are always increasing

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 smaller
- And:
  - $\text{write}_A(\text{label}[A]) \rightarrow$
    $\text{read}_B(\text{label}[A]) \rightarrow$
    $\text{write}_B(\text{label}[B]) \rightarrow$
    $\text{read}_B(\text{flag}[A])$
- So B is locked out while flag[A] is true

```
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));
 }
```

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],A > label[B],B

```
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));
 }
```

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen flag[A] == false

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen flag[A] == false
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$

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 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));
 }
```

Mutex breaks if label[i] overflows

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 Good News

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

The ~~Good~~ Bad News

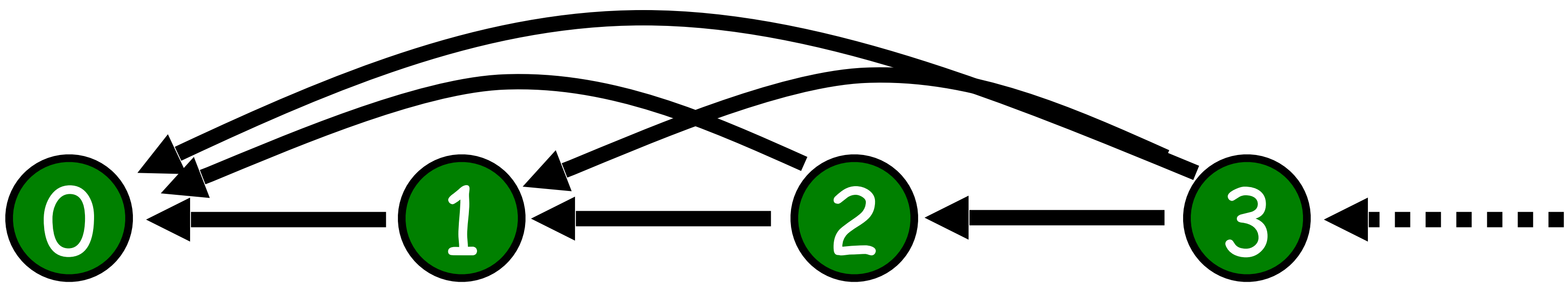
- One can construct a
  - Wait-free (no mutual exclusion)
  - Concurrent
  - Timestamping system
  - That never overflows

This part is hard

Instead …

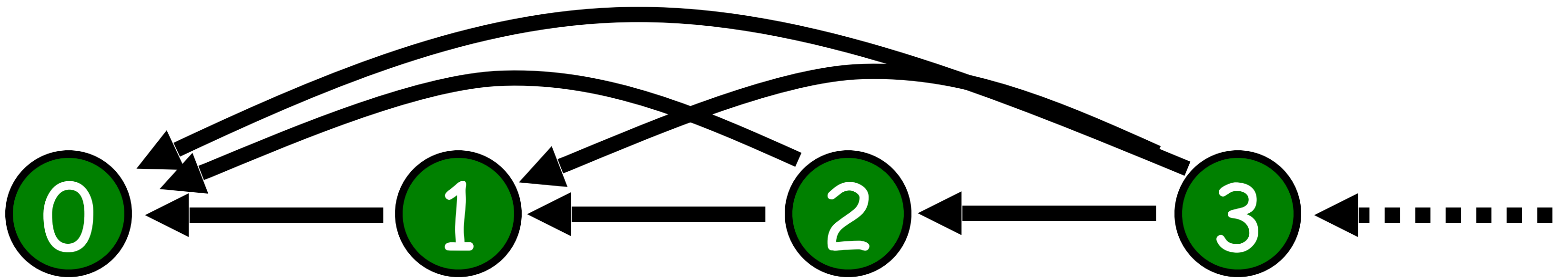
- We construct a Sequential timestamping system
  - Same basic idea
  - But simpler
- Uses mutex to read & write atomically
- No good for building locks
  - But useful anyway

Precedence Graphs



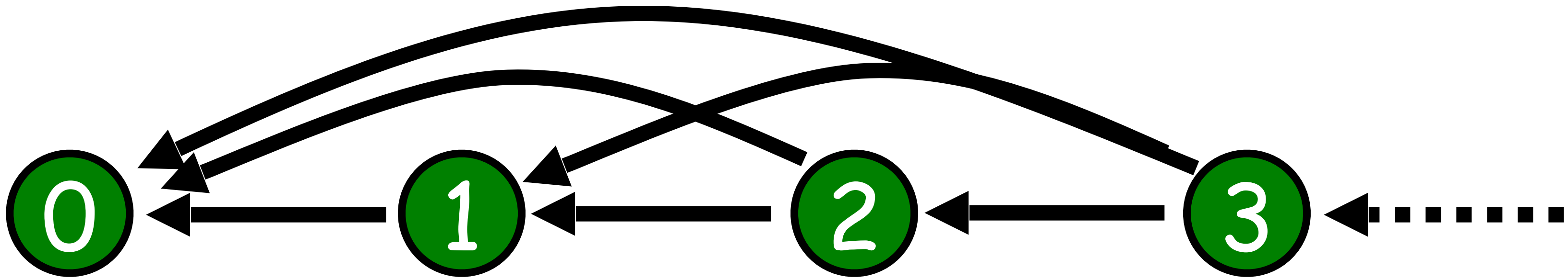
- Timestamps form directed graph
- Edge x to y
  - Means x is later timestamp
  - We say x dominates y

Unbounded Counter Precedence Graph

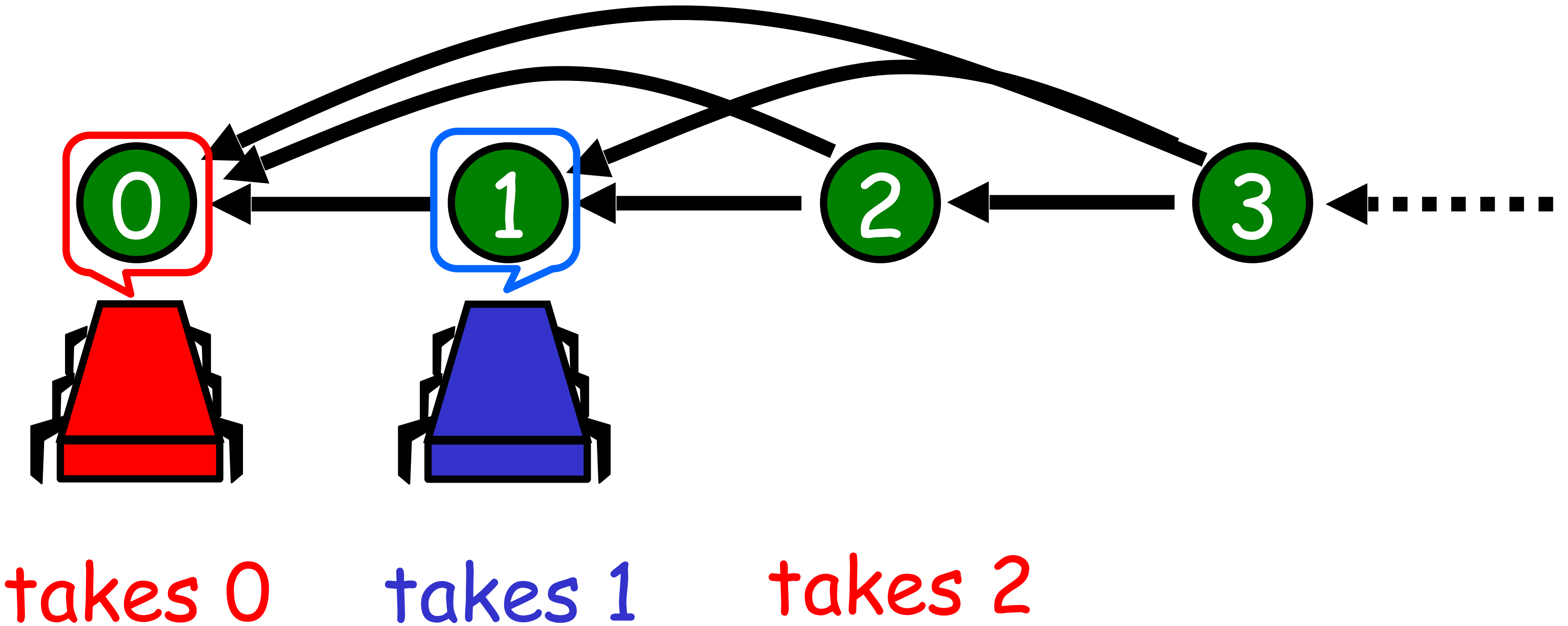


- Timestamping = move tokens on graph
- Atomically
  - read others' tokens
  - move mine
- Ignore tie-breaking for now

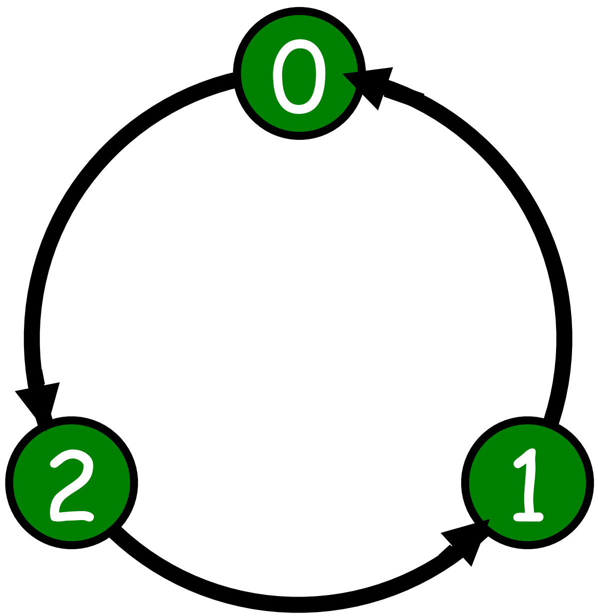
Unbounded Counter Precedence Graph



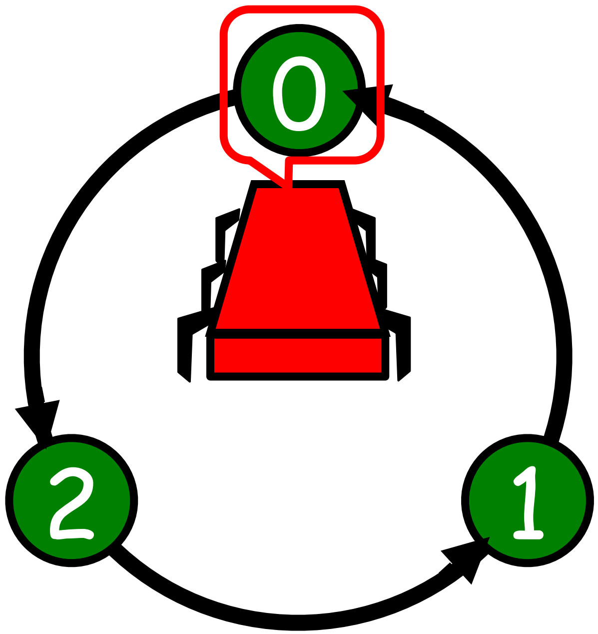
Unbounded Counter Precedence Graph



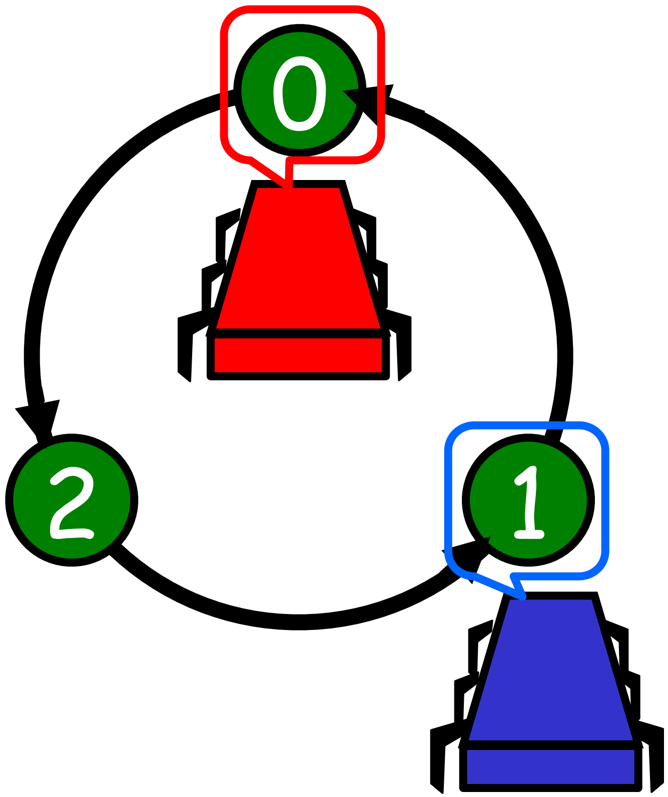
Two-Thread Bounded Precedence Graph



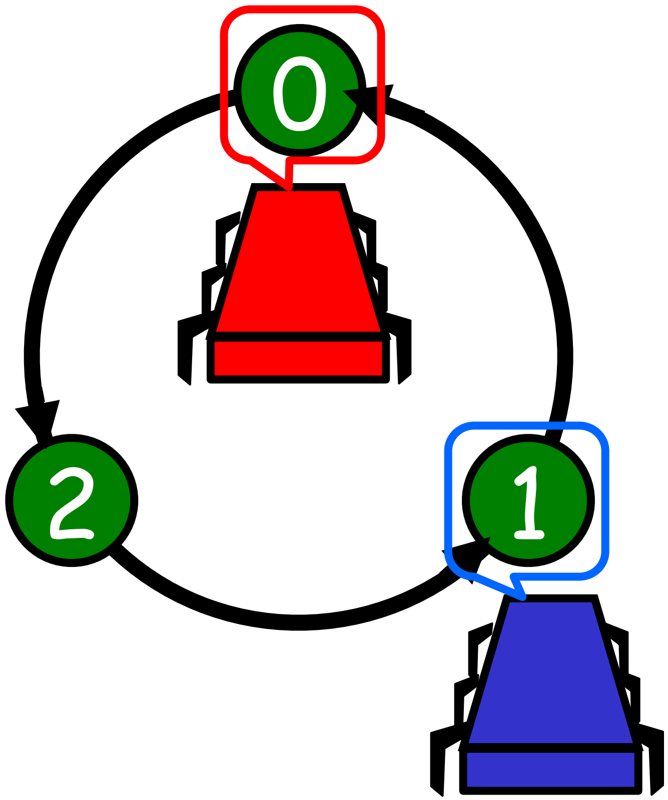
Two-Thread Bounded Precedence Graph



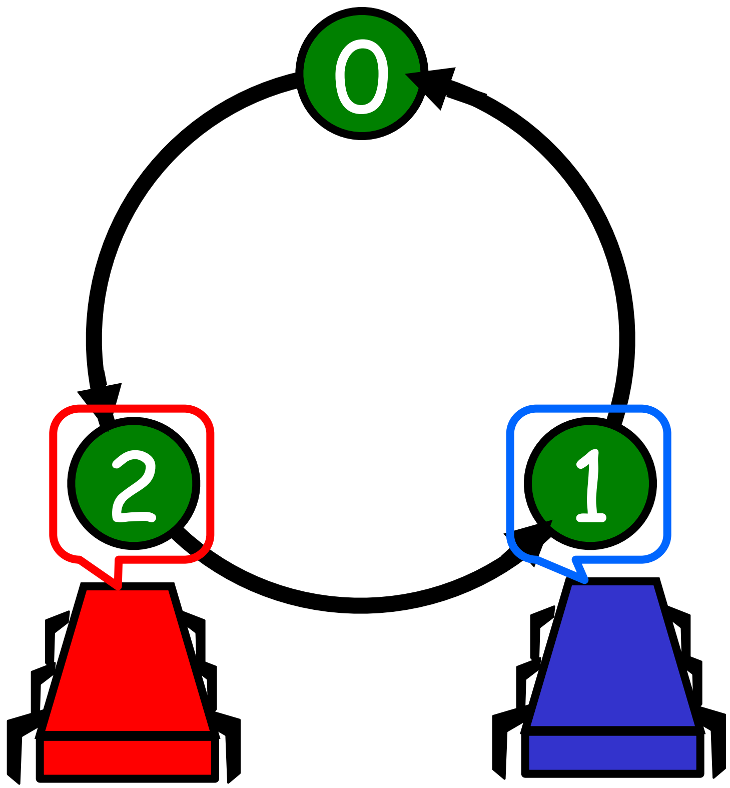
Two-Thread Bounded Precedence Graph



Two-Thread Bounded Precedence Graph

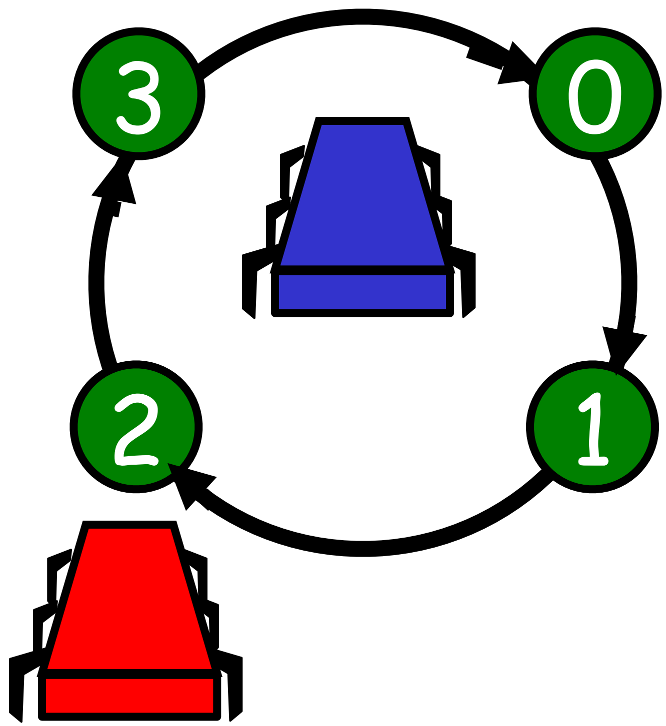


Two-Thread Bounded Precedence Graph T^2

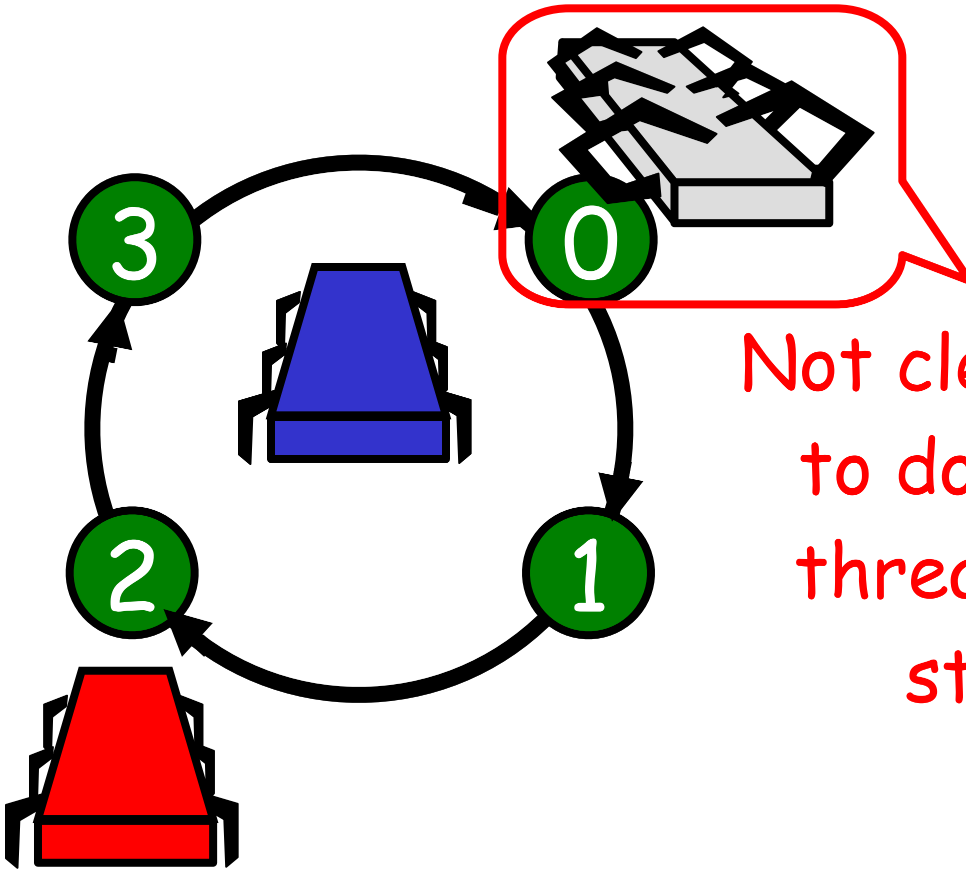


and so on …

Three-Thread Bounded Precedence Graph?

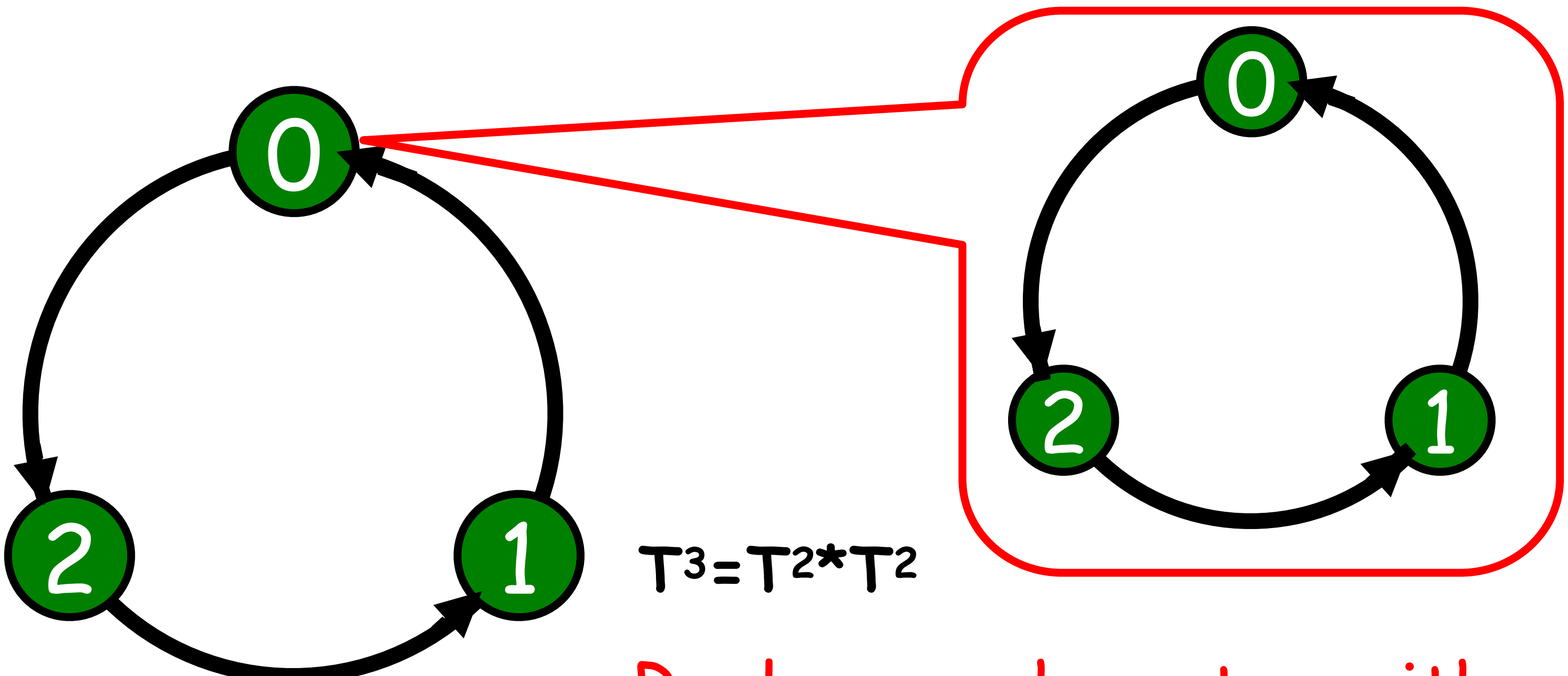


Three-Thread Bounded Precedence Graph?



Not clear what to do if one thread gets stuck

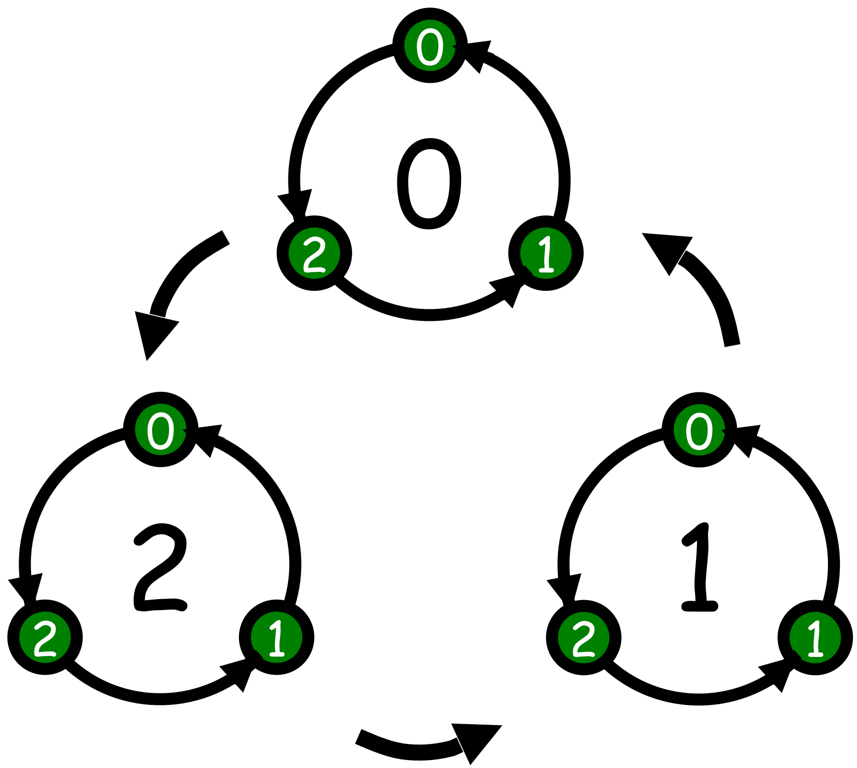
Graph Composition



Replace each vertex with a copy of the graph

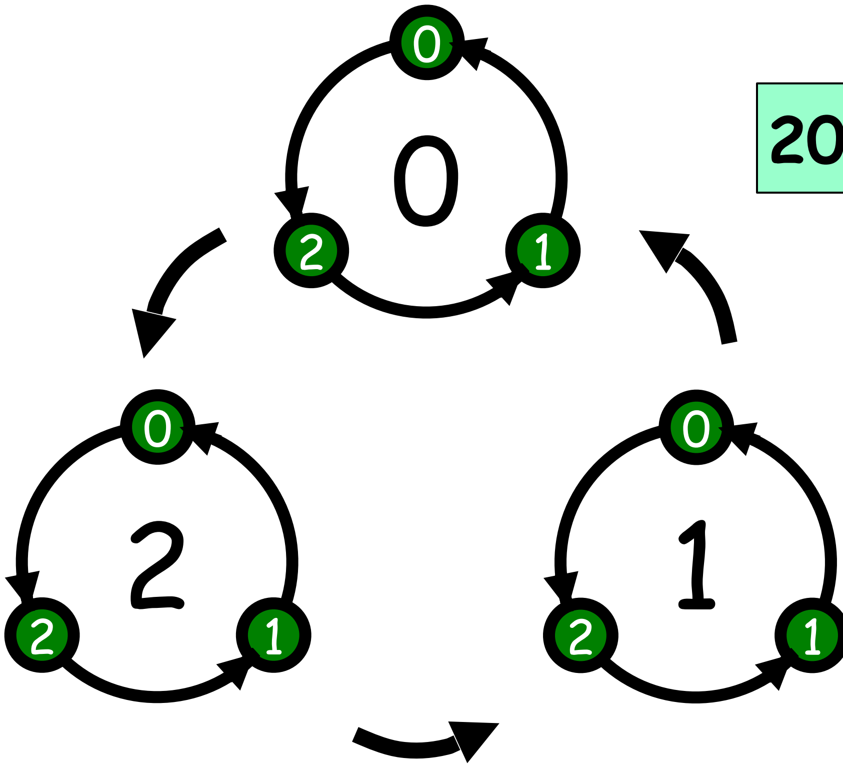
Three-Thread Bounded Precedence Graph

T^3



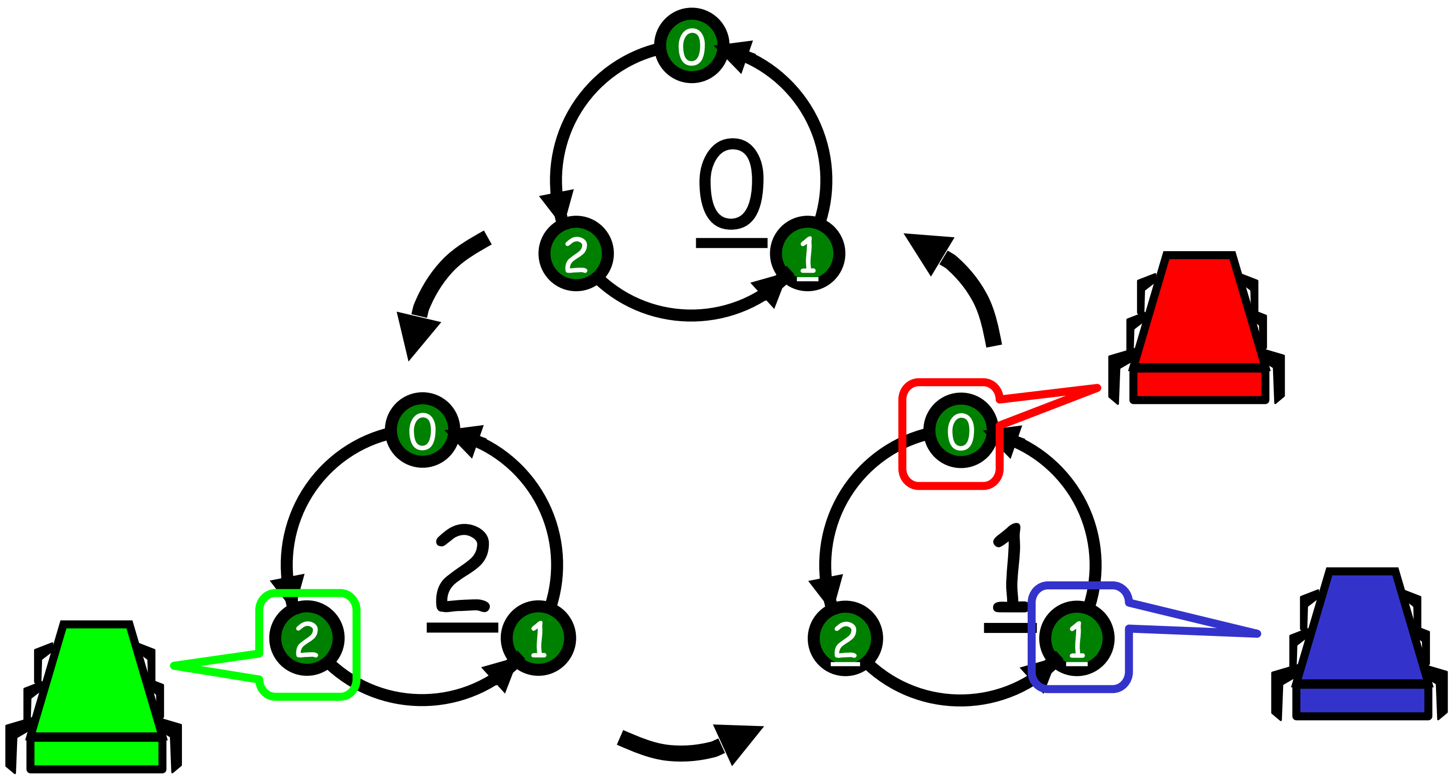
Three-Thread Bounded Precedence Graph

T^3



20 < 02 < 12

Three-Thread Bounded Precedence Graph T^3

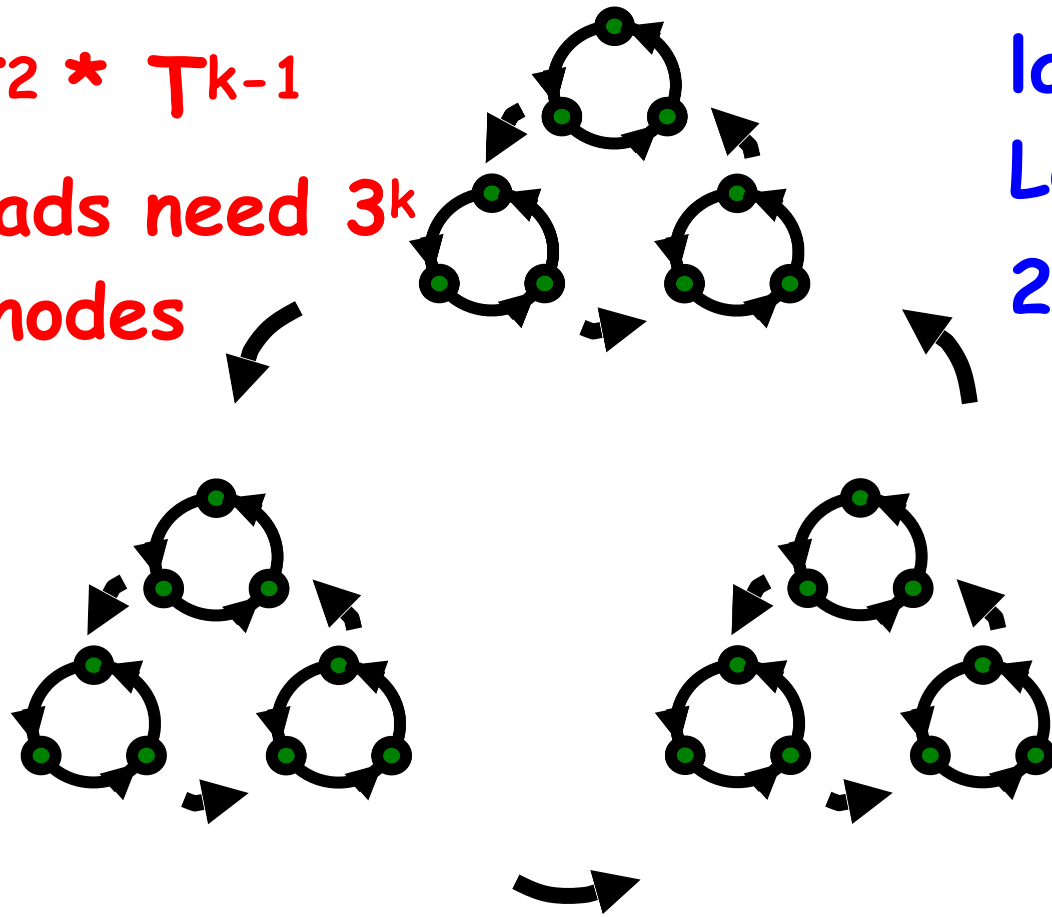


and so on…

In General

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

K threads need 3^k nodes



label size = $Log_2(3^k)$ = 2n

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 variables

Shared Memory

- Shared read/write memory locations called Registers (historical reasons)
- Come in different flavors
  - Multi-Reader-Single-Writer (Flag[])
  - Multi-Reader-Multi-Writer (**Victim**[])
  - Not that interesting: SRMW and SRSW

Theorem

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

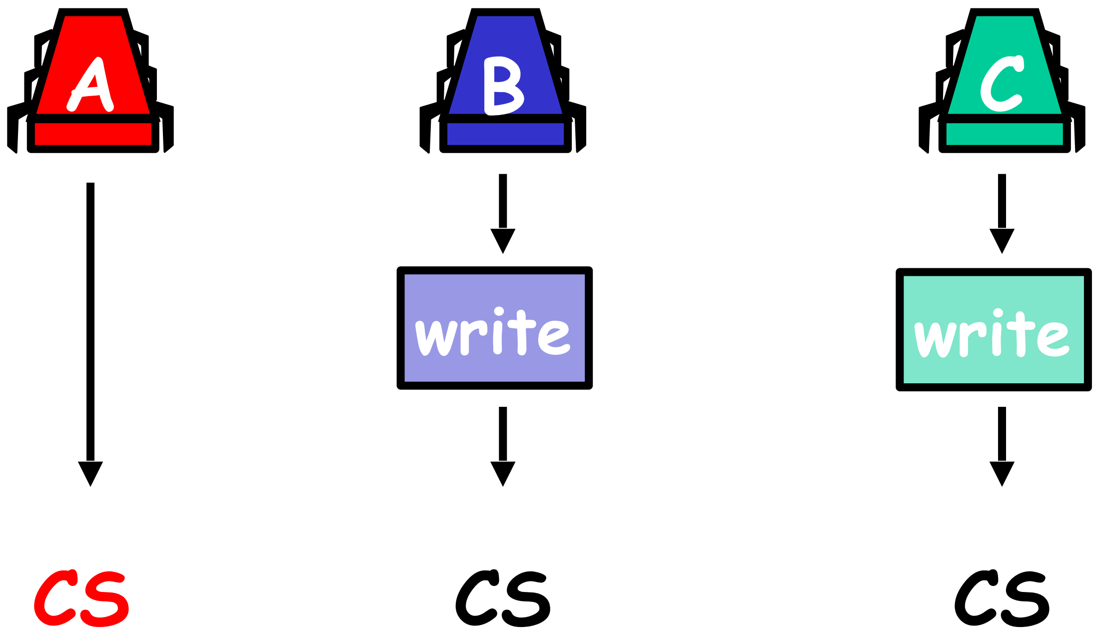
N registers like Flag[]…

Proving Algorithmic Impossibility

- To show no algorithm exists:
  - assume by way of contradiction one does,
  - show a bad execution that violates properties:
  - in our case assume an alg for deadlock free mutual exclusion using < N registers

Proof: Need N-MRSW Registers

Each thread must write to some register



…can't tell whether A is in critical section

Upper Bound

- Bakery algorithm
  - Uses 2N MRSW registers
- So the bound is (pretty) tight
- But what if we use MRMW registers?
  - Like **victim[]** ?

Bad News Theorem

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

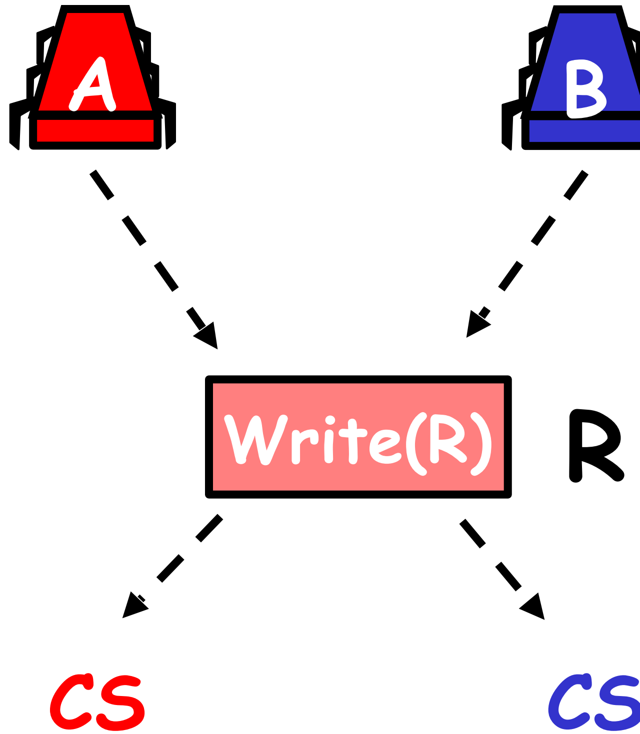
(So multiple writers don't help)

Theorem (First 2-Threads)

Theorem: Deadlock-free mutual exclusion for 2 threads requires at least 2 multi-reader multi-writer registers

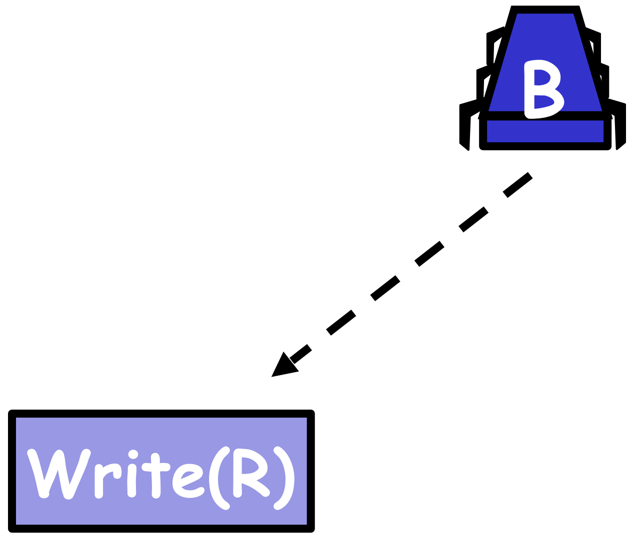
Proof: assume one register suffices and derive a contradiction

Two Thread Execution



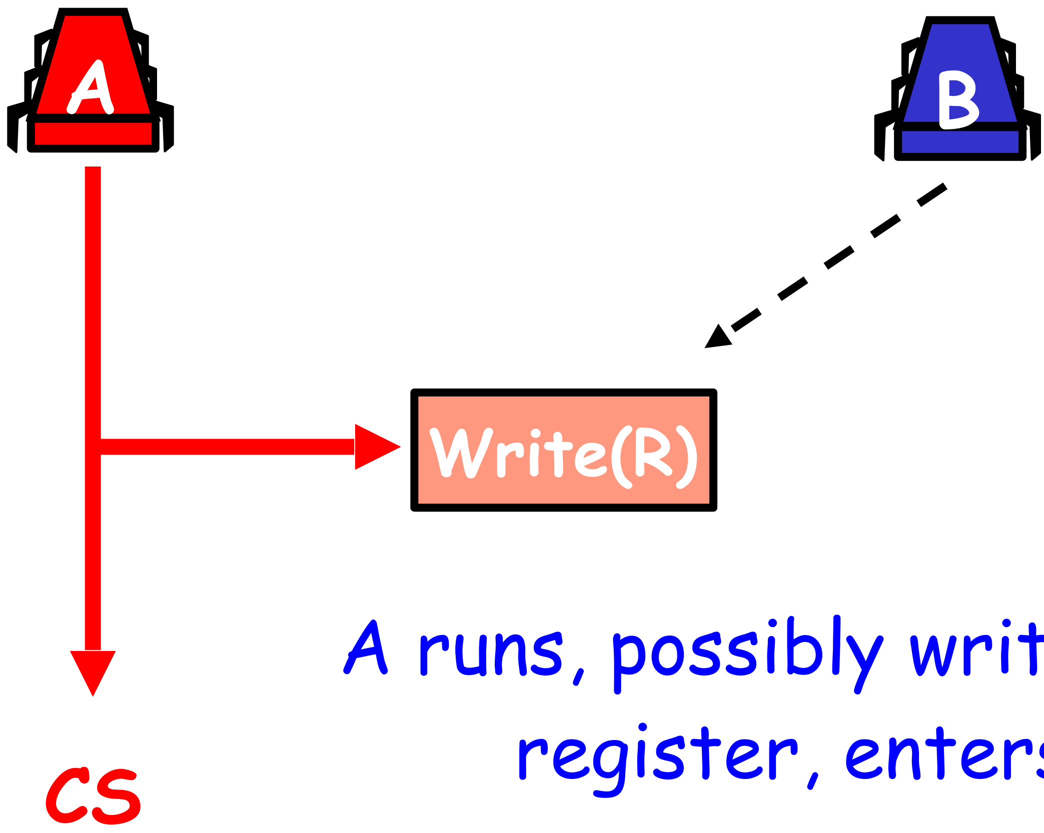
- Threads run, reading and writing R
- Deadlock free so at least one gets in

Covering State for One Register Always Exists



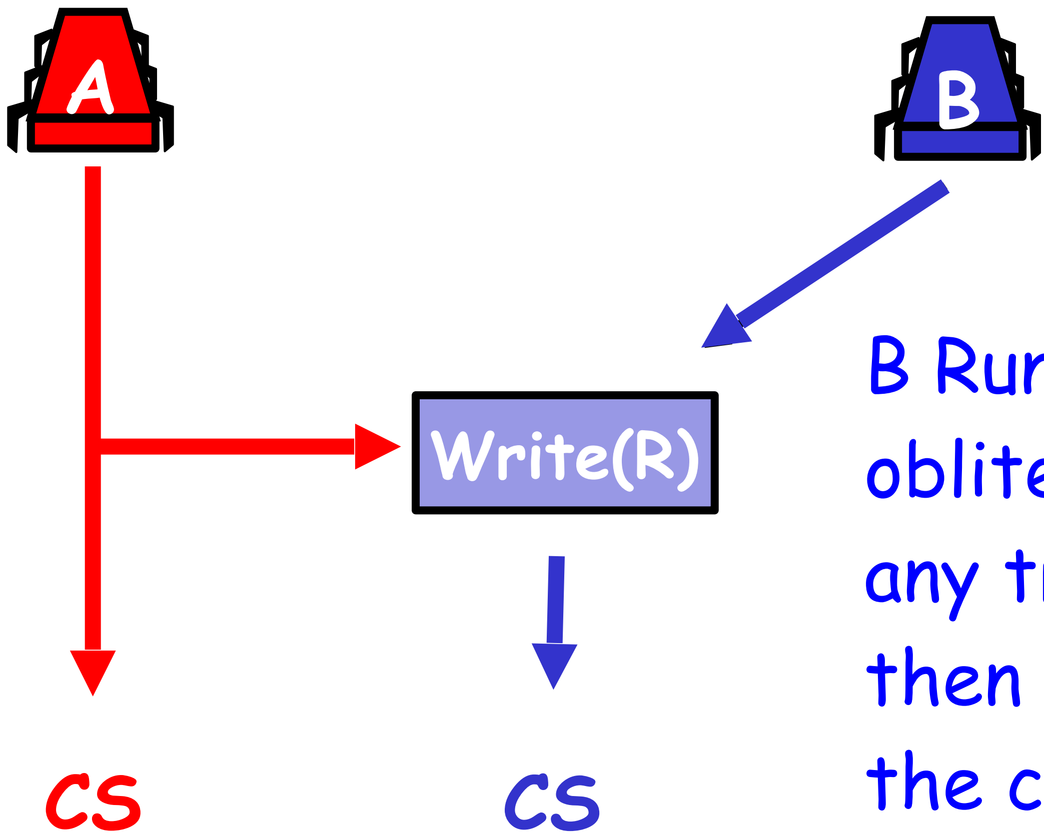
In any protocol B has to write to the register before entering CS, so stop it just before

Proof: Assume Cover of 1



A runs, possibly writes to the register, enters CS

Proof: Assume Cover of 1



B Runs, first obliterating any trace of A, then also enters the critical section

Theorem

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