Introduction

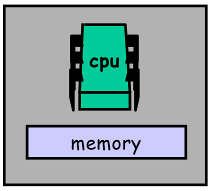
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Outline

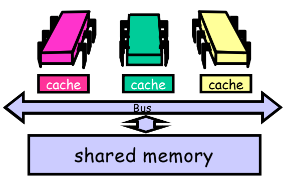
- The Concurrency Speedup
- Java Threading Basics
- Classical Problems of Concurrency
- Basic Locks

The Concurrency Speedup

The Uniprocesor

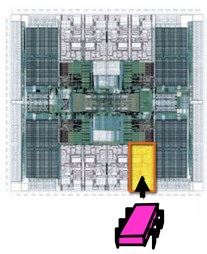


The Shared Memory Multiprocessor (SMP)



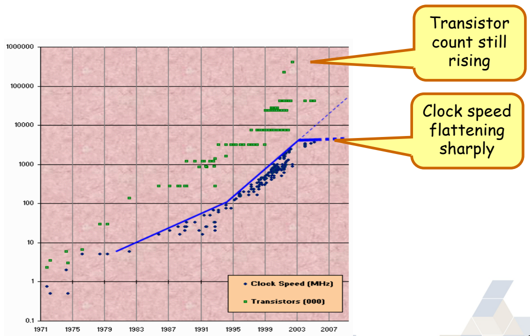
The Multicore Processor (CMP)

All on the same chip

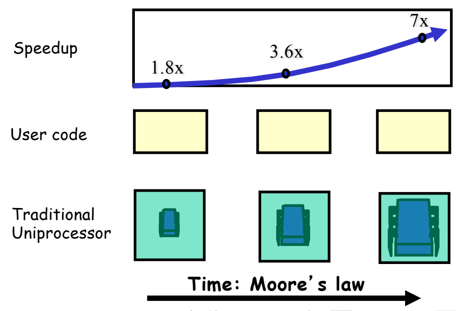


Sun T2000 Niagara

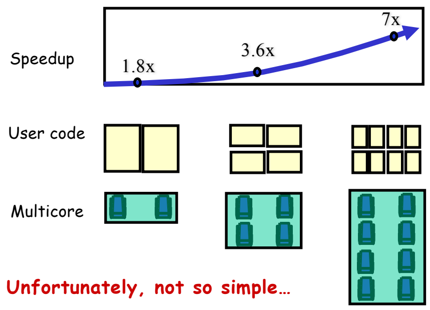
Moore’s Law



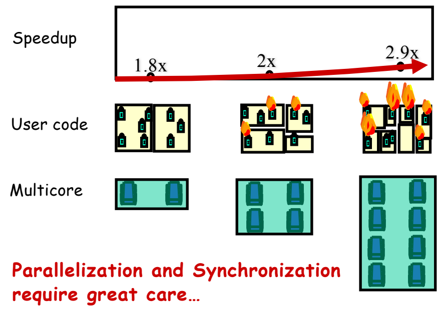
Traditional Scaling Process



Multicore Scaling Process



Real-World Scaling Process

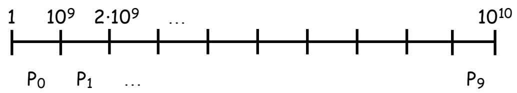


Real-World Scaling Process

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

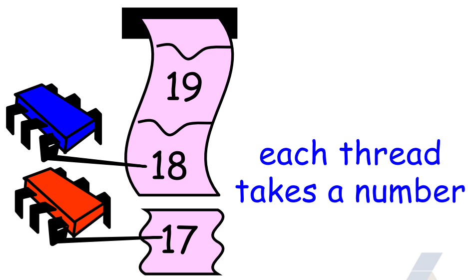
Procedure for Thread *i*

```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10⁹+1, j<(i+1)*10⁹; j++) {
    if (isPrime(j))
      print(j);
  }
}
```


Issues

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

Shared Counter



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

$$\text{Speedup} = \frac{\text{OldExecutionTime}}{\text{NewExecutionTime}}$$

…of computation given n CPUs instead of 1

Amdahl's Law

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

Amdahl's Law

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

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

Process Concept

- All multiprogramming OSs are built around the concept of processes.
- Process – a program in execution; an instance of a running program; the entity that can be assigned to, and executed on, a processor.
- Program is a *passive entity*, process is an *active entity*.
- A process includes three segments:
  1. Program: code/text.
  2. Data: program variables and heap.
  3. Stack: for procedure calls and parameter passing.

Process Characteristics (1)

- Unit of resource ownership – process is allocated:
  - an address space to hold the process image.
  - control of some resources (files, I/O devices...).
- Unit of dispatching – process is an execution path through one or more programs:
  - execution may be interleaved with other process.
  - the process has an execution state and a dispatching priority.

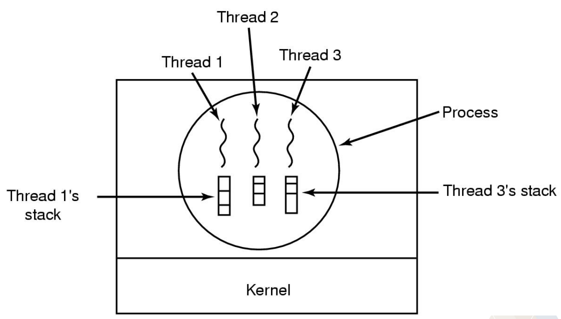
Process Characteristics (2)

- These two characteristics are treated independently by some recent OSs:
  1. The unit of resource ownership is usually referred to as a Task or (for historical reasons) also as a Process.
  2. The unit of dispatching is usually referred to a Thread or a Light-Weight Process (LWP).
- A traditional Heavy-Weight Process (HWP) is equal to a task with a single thread.
- Several threads can exist in the same task.
  - ***Multithreading*** - The ability of an OS to support multiple, concurrent paths of execution within a single process.

Processes and Threads (1)

- Process Items (shared by all threads of task):
  - address space which holds the process image
  - global variables
  - protected access to files, I/O and other resources
- Thread Items:
  - an execution state (Running, Ready, etc.)
  - program counter, register set
  - execution stack
  - some per-thread static storage for local variables
  - saved thread context when not running

Each thread has its own stack

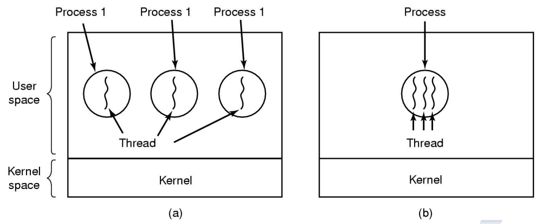


Tasks/Processes and Threads (2)

Per process items	Per thread items
Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information	Program counter Registers Stack State

The Process vs. Thread Model

(a) Three processes each with one thread.
(b) One process with three threads.



Processes vs. Threads

- Creating and managing processes is generally regarded as an expensive task (fork system call).
- Making sure all the processes peacefully co-exist on the system is not easy (as concurrency transparency comes at a price).
- Threads can be thought of as an "execution of a part of a program (in user-space)".
- Rather than make the OS responsible for concurrency transparency, it is left to the individual application to manage the creation and scheduling of each thread.

Thread operation latencies (μs)

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Source: Anderson, T. et al, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM TOCS, February 1992.

Classical Problems of Concurrency

Classical Problems of Concurrency

- There are many of them – let's briefly see three famous problems:

1. Critical Section
2. Producer-Consumer
3. Readers and Writers

The Critical-Section Problem

- *n* processes competing to use some shared data.
- No assumptions may be made about speeds or the number of CPUs.
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.

What It Means

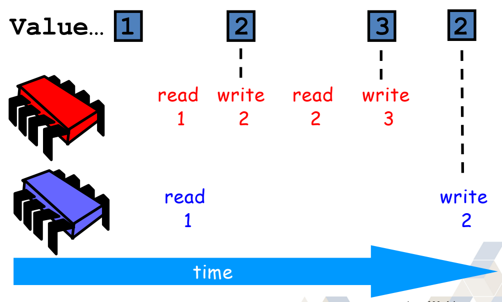
```
public class Counter {
  private long value;

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


What It Means

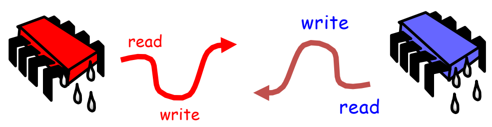
```
public class Counter {
  private long value;

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

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


Not so good…



Is this problem inherent?



If we could only glue reads and writes…

An Aside: Java™

```
public class Counter {
  private long value;

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

Synchronized block

Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:
  1. **Mutual Exclusion**
  2. **Progress**
  3. **Bounded Waiting**
- We can check on all three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.

Types of solutions to CS problem

- Software solutions – e.g. Locks/Monitors
  - algorithms who's correctness does not rely on any other assumptions.
- Hardware solutions – e.g. Atomic Instructions
  - rely on some special machine instructions.
- Operating System solutions – e.g. Semaphores/IPC
  - provide some functions and data structures to the programmer through system/library calls.

Initial Attempts to Solve Problem

- General structure of process P_i (other is P_j) –

```
do {
    entry section
      critical section
    leave section
      remainder section
} while (TRUE);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables
  - **boolean flag[2]**; initially **flag [0] = flag [1] = FALSE**
  - **flag [i] = TRUE** $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

```
do {
    while (flag[j]);
    flag[i] = TRUE;
      critical section
    flag [i] = FALSE;
      remainder section
} while (TRUE);
```

- Satisfies progress, but not mutual exclusion and bounded waiting requirements.

Algorithm 2

- Shared variables
  - **boolean flag[2]**; initially **flag [0] = flag [1] = FALSE**
  - **flag [i] = TRUE** $\Rightarrow$ P_i wants to enter its critical section
- Process P_i

```
do {
    flag[i] = TRUE;
    while (flag[j]);
    critical section
    flag [i] = FALSE;
      remainder section
} while (TRUE);
```

- Satisfies mutual exclusion, but not progress and bounded waiting (?) requirements.

Algorithm 3

- Shared variables:
  - **int turn**; initially **turn = 0**
  - **turn = i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

```
do {
    while (turn != i);
      critical section
    turn = j;
    remainder section
} while (TRUE);
```

- Satisfies mutual exclusion and bounded waiting, but not progress.

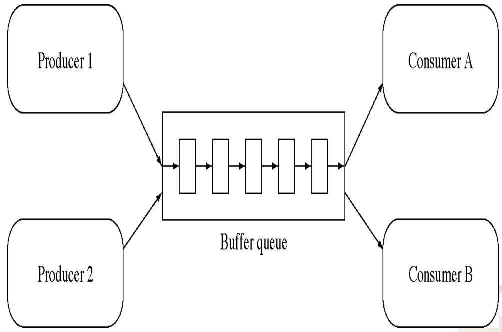
Algorithm 4

- Combined shared variables of algorithms 1 and 2/3.
- Process P_i

```
do {
    flag [i] = TRUE;
    turn = j;
    while (flag [j] and turn == j);
        critical section
    flag [i] = FALSE;
        remainder section
} while (TRUE);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Multiple Producers and Consumers



Producer/Consumer (P/C) Dynamics

- A producer process produces information that is consumed by a consumer process.
- At any time, a producer activity may create some data.
- At any time, a consumer activity may want to accept some data.
- The data should be saved in a buffer until they are needed.
- If the buffer is finite, we want a producer to block if its new data would overflow the buffer.
- We also want a consumer to block if there are no data available when it wants them.

P/C Bounded-Buffer Problem

- We need 3 semaphores:

1. A semaphore **mutex** (initialized to 1) to have mutual exclusion on buffer access.
2. A semaphore **full** (initialized to 0) to synchronize producer and consumer on the number of consumable items.
3. A semaphore **empty** (initialized to n) to synchronize producer and consumer on the number of empty spaces.

Bounded-Buffer – Semaphores

- Shared data

semaphore full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1

Bounded-Buffer – Producer Process

```
do {
    …
    produce an item in nextp
    …
    wait(empty);
    wait(mutex);
    …
    add nextp to buffer
    …
    signal(mutex);
    signal(full);
} while (TRUE);
```

Bounded-Buffer – Consumer Process

```
do {
    wait(full)
    wait(mutex);
        …
    remove an item from buffer to nextc
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in nextc
        …
} while (TRUE);
```

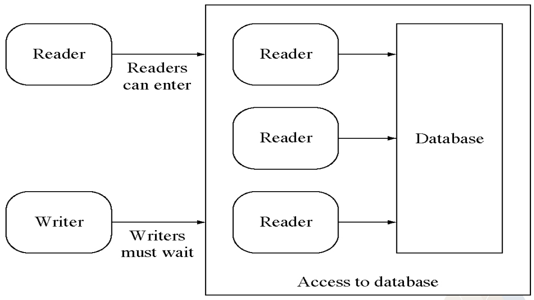
Readers-Writers Problem

- A data set/repository is shared among a number of concurrent processes:
  - Readers – only read the data set; they do **not** perform any updates.
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

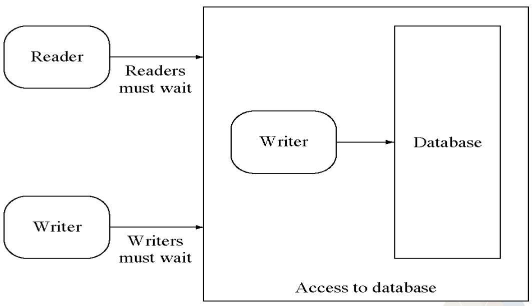
Readers-Writers Dynamics

- Any number of reader activities and writer activities are running.
- At any time, a reader activity may wish to read data.
- At any time, a writer activity may want to modify the data.
- Any number of readers may access the data simultaneously.
- During the time a writer is writing, no other reader or writer may access the shared data.

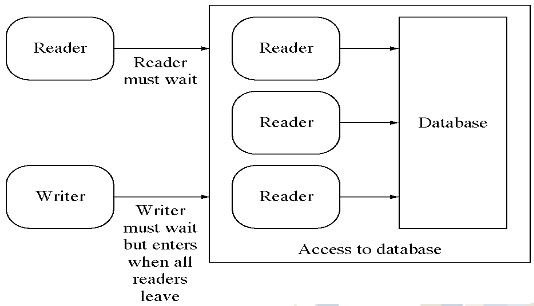
Readers-Writers with active readers



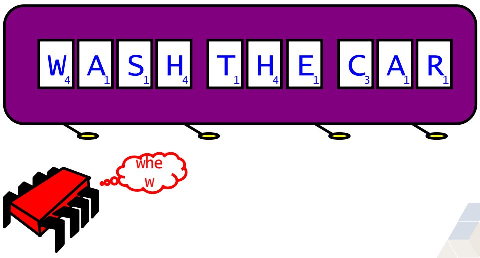
Readers-Writers with an active writer



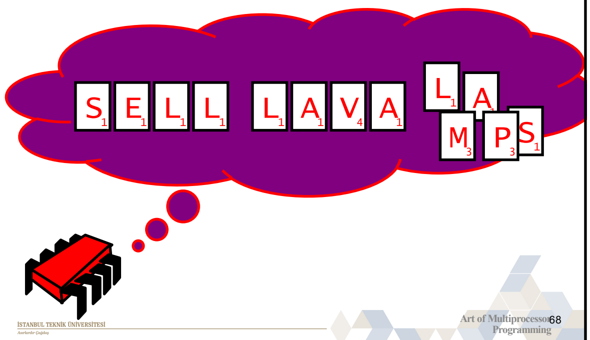
Should readers wait for waiting writer?



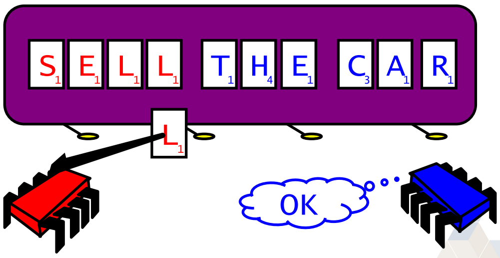
To post a message



Let's send another message



Uh-Oh



Readers-Writers problem

- There are various versions with different readers and writers preferences:

1. The **first** readers-writers problem, requires that no reader will be kept waiting unless a writer has obtained access to the shared data.
2. The **second** readers-writers problem, requires that once a writer is ready, no new readers may start reading.
3. In a solution to the **first** case writers may starve; In a solution to the **second** case readers may starve.

First Readers-Writers Solution (1)

- **readcount** (initialized to 0) counter keeps track of how many processes are currently reading.
- **mutex** semaphore (initialized to 1) provides mutual exclusion for updating readcount.
- **wrt** semaphore (initialized to 1) provides mutual exclusion for the writers; it is also used by the first or last reader that enters or exits the CS.

First Readers-Writers Solution (2)

- Shared data

semaphore mutex, wrt;
int readcount;

Initially

mutex = 1, wrt = 1, readcount = 0

First Readers-Writers – Writer Process

```
do {
  wait(wrt);
          …
      writing is performed
          …
   signal(wrt);
} while(TRUE);
```

First Readers-Writers – Reader Process

```
do {
      wait(mutex);
        readcount++;
        if (readcount == 1)
            wait(wrt);
      signal(mutex);
              …
          reading is performed
              …
     wait(mutex);
        readcount--;
        if (readcount == 0)
          signal(wrt);
     signal(mutex);
} while(TRUE);
```

Java Threading Basics

Threads Overview

- Java supports threads
  - Threads execute within a single JVM
  - Native threads map a single Java thread to an OS thread
  - Green threads adopt the thread library approach (threads are invisible to the OS)
  - On a multiprocessor system, native threads are required to get true parallelism (but this is still implementation dependent)

Threads Overview

- There are various ways in which concurrency can be introduced by
  - an API for explicit thread creation or thread forking
  - a high-level language construct such as PAR (occam), tasks (Ada), or processes (Modula)
- Integration with OOP, various models:
  - asynchronous method calls
  - early return from methods
  - futures
  - active objects

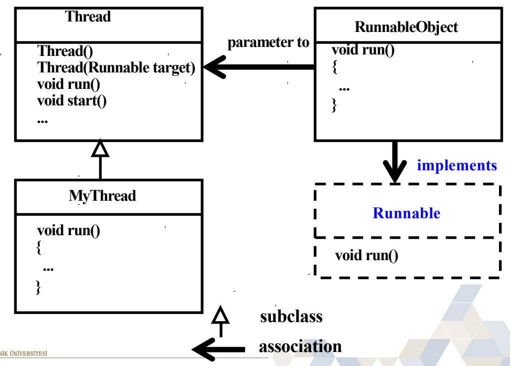
Concurrency in Java

- Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads are created
- However to avoid all threads having to be child classes of `Thread`, it also uses a standard interface

```
public interface Runnable {
  public void run();
}
```

- Hence, any class which wishes to express concurrent execution must implement this interface and provide the `run` method
- Threads do not begin their execution until the `start` method in the `Thread` class is called

Threads in Java



Thread Identification

- The identity of the currently running thread can be found using the `currentThread` method
- This has a static modifier, which means that there is only one method for all instances of `Thread` objects
- The method can always be called using the `Thread` class

```
public class Thread extends Object
             implements Runnable {
  ...

  public static Thread currentThread();
  ...
}
```

Threads in Java

```
public class PrintNumbers {
   public static void printNumbers() {
      for(int i=0; i<1000; i++) {
         System.out.println(
               Thread.currentThread().getId()+": " + i);
      }
   }
}
```


Threads in Java

```
public class Thread1 extends Thread {

   @Override
   public void run() {
      System.out.println("Thread1 ThreadId: " +
         Thread.currentThread().getId());
      // do our thing

      PrintNumbers.printNumbers();
   }
}
```


Threads in Java

```
   static public void main(String[] args) {
      System.out.println("Main ThreadId: " +
            Thread.currentThread().getId());

      for(int i=0; i<3; i++)
         new Thread1().start(); // don't call run!


      printNumbers();
   }
```


Threads in Java

```
public class Thread2 implements Runnable {

   @Override
   public void run() {
      System.out.println("Thread2 ThreadId: " +
         Thread.currentThread().getId());

         PrintNumbers.printNumbers();
   }
}
```

Threads in Java

```java
static public void main(String[] args) {
   System.out.println("Main ThreadId: " +
         Thread.currentThread().getId());

   for(int i=0; i<3; i++)
     new Thread(new Thread2()).start();

   printNumbers();
}
```


Threads in Java

```java
static public void main(String[] args) {
   System.out.println("Main ThreadId: " +
      Thread.currentThread().getId());
   new Thread(new Runnable() {
      @Override
      public void run() {
         System.out.println("Thread3 ThreadId: " +
               Thread.currentThread().getId());

         printNumbers();
      }
   }).start();

}
```


Threads in Java

```java
public class PrintNumbers {
   public static void main(String[] args) {
      Runnable task1 = () -> {
            for(int i=0; i<1000; i++)
              System.out.println(Thread.currentThread()
                                   .getId()+": " + i);

          };

          Thread[] t = new Thread[12];
          for(int i=0;i<t.length;i++)
              t[i] = new Thread(task1);
         }
}
```


A Thread Terminates:

- when it completes execution of its `run` method either normally or as the result of an unhandled exception
- (DEPRECATED) via a call to its `stop` method — the `run` method is stopped and the thread class cleans up before terminating the thread (releases locks and executes any finally clauses)
  - the thread object is now eligible for garbage collection.
  - `stop` is inherently unsafe as it releases locks on objects and can leave those objects in inconsistent states; the method is now deprecated and should not be used
- (NOT IMPLEMENTED) by its `destroy` method being called — destroy terminates the thread without any cleanup (not provided by many JVMs, now deprecated)

Daemon Threads

- Java threads can be of two types: user threads or daemon threads
- Daemon threads are those threads which provide general services and typically never terminate
- When a new thread is created it inherits the daemon status of its parent.
- Normal thread and daemon threads differ in what happens when they exit. When the JVM halts any remaining daemon threads are abandoned: finally blocks are not executed, stacks are not unwound - JVM just exits. Due to this reason daemon threads should be used sparingly and it is dangerous to use them for tasks that might perform any sort of I/O.
- The `setDaemon` method must be called before the thread is started

Joining

- One thread can wait (with or without a timeout) for another thread (the target) to terminate by issuing the `join` method call on the target's thread object
- The `isAlive` method allows a thread to determine if the target thread has terminated

Communication in Java

- Via reading and writing to data encapsulated in shared objects protected by simple monitors
- Every object is implicitly derived from the `Object` class which defines a mutual exclusion lock
- Methods in a class can be labeled as synchronized, this means that they can only be executed if the lock can be acquired (this happens automatically)
- The lock can also be acquired via a synchronized statement which names the object
- A thread can wait and notify on a single anonymous condition variable

Synchronization

Synchronization of threads is needed in order to control threads coordination, mainly in order to prevent simultaneous operations on data

For simple synchronization Java provides the synchronized keyword

For more sophisticated locking mechanisms, starting from Java 5, the package java.concurrent.locks provides additional locking options

Synchronization

```
public class SynchronizedCounter {
   private int c = 0;
   public synchronized void increment() { c++; }
   public synchronized void decrement() { c--; }
   public synchronized int value() { return c; }
}
```

The synchronized keyword on a method means that if this is already locked anywhere (on this method or elsewhere) by another thread, we need to wait till this is unlocked before entering the method

Synchronization

```
public void addName(String name) {
   synchronized(this) {
      lastName = name;
      nameCount++;
   }
   nameList.add(name);
}
```

When synchronizing a block, key for the locking should be supplied (usually would be this)

The advantage of not synchronizing the entire method is efficiency

Synchronization

```
public class TwoCounters {
   private long c1 = 0, c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();
   public void inc1() {
      synchronized(lock1) {
         c1++;
      }
   }
   public void inc2() {
      synchronized(lock2) {
         c2++;
      }
   }
}
```

You must be absolutely sure that there is no tie between c1 and c2

Synchronization

Having a static method be synchronized means that ALL objects of this type are locked on the method and can get in one thread at a time.

The lock is the Class object representing this class.

The performance penalty might be sometimes too high – needs careful attention!

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

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

Announce I'm interested

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 it is the victim
- One or the other must not be the victim

Starvation Free

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

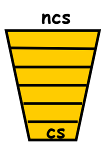
  `flag[j] == true` and
  `victim == i`
- When `j` re-enters
  - it sets `victim` to `j`.
  - So `i` gets in

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

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


The Filter Algorithm for *n* Threads

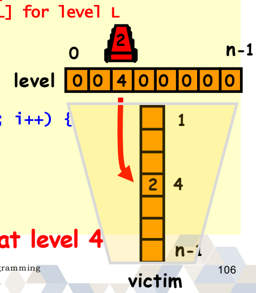
There are ***n-1*** "waiting rooms" called levels

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



Filter

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

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



Filter

```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;

      while ((∃ k != i level[k] >= L) &&
             victim[L] == i );
    }}
  public void unlock() {
    level[i] = 0;
  }}
```


Filter

```
class Filter implements Lock {
  …

  public void lock() {
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;

      while ((∃ k != i) level[k] >= L) &&
             victim[L] == i);
    }}
  public void release(int i) {
    level[i] = 0;
  }}
```

One level at a time

Filter

```
class Filter implements Lock {
  …

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

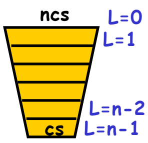

Filter

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



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

Bakery Algorithm

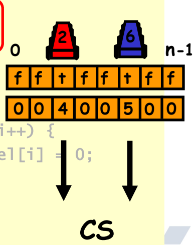
- Provides First-Come-First-Served
- How?
  - Take a "number"
  - Wait until lower numbers have been served
- Lexicographic order
  - (a,i) > (b,j)
    - If a > b, or a = b and i > j

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

Doorway

```
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

I'm interested

```
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));
 }
```

Take increasing label (read labels in some arbitrary order)

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

Bakery Y2^32K 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));
 }
```

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

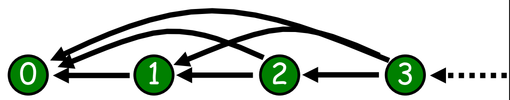
Does Overflow Actually Matter?

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

Timestamps

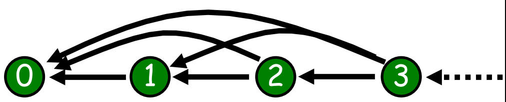
- Label variable is really a timestamp
- Need ability to
  - Read others' timestamps
  - Compare them
  - Generate a later timestamp
- Can we do this without overflow?

Precedence Graphs



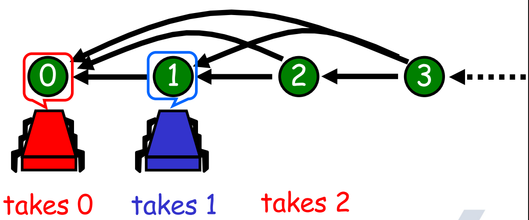
- Timestamps form directed graph
- Edge x to y
  - Means x is later timestamp
  - We say x dominates y

Unbounded Counter Precedence Graph

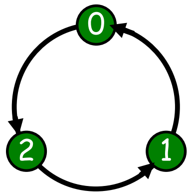


- Timestamping = move tokens on graph
- Atomically
  - read others' tokens
  - move mine
- Ignore tie-breaking for now

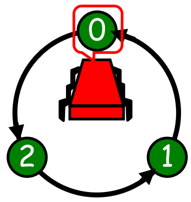
Unbounded Counter Precedence Graph



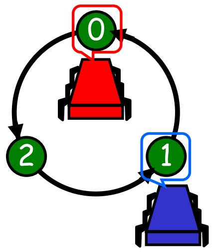
Two-Thread Bounded Precedence Graph



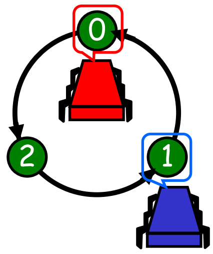
Two-Thread Bounded Precedence Graph



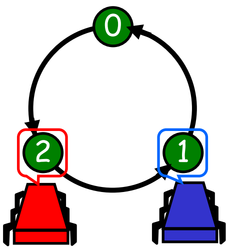
Two-Thread Bounded Precedence Graph



Two-Thread Bounded Precedence Graph

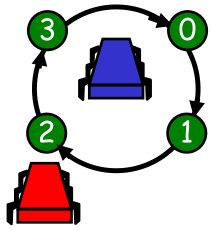


Two-Thread Bounded Precedence Graph T^2

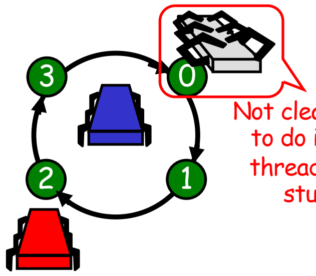


and so on …

Three-Thread Bounded Precedence Graph?

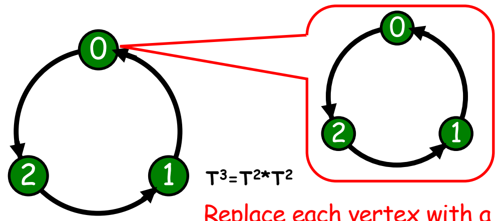


Three-Thread Bounded Precedence Graph?



Not clear what to do if one thread gets stuck

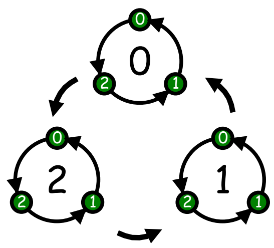
Graph Composition



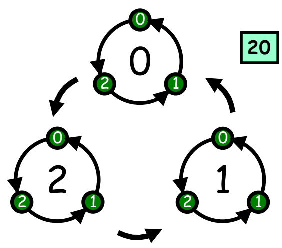
$T^3 = T^2 * T^2$

Replace each vertex with a copy of the graph

Three-Thread Bounded Precedence Graph T^3

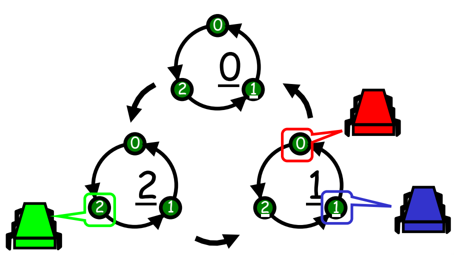


Three-Thread Bounded Precedence Graph T^3



20 < 02 < 12

Three-Thread Bounded Precedence Graph T^3

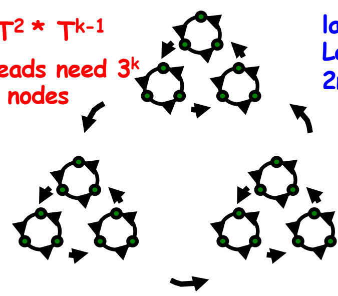


and so on…

In General

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

K threads need 3^k nodes



label size = $\log_2(3^k)$ = 2n

Slides adopted from

- Java Concurrency Framework by Sidartha Gracias
- Java Threads by Amir Kirsh
- Concurrent Programming in Java by Andy Wellings

... and examples from

- Java Concurrency in Practice by Brian Goetz
- Concurrent Programming in Java by Doug Lea