Object-oriented programming and data-structures

CS/ENGRD 2110
SUMMER 2018

Lecture 16: Concurrency

http://courses.cs.cornell.edu/cs2110/2018su

CPU Central Processing Unit. Simplified view

The CPU is the part of the computer that executes instructions.

Java: x= x + 2;

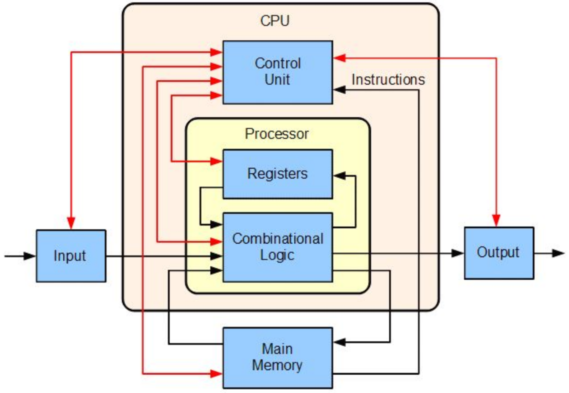
Suppose variable x is at Memory location 800, Instructions at 10

Machine language:

10: load register 1, 800

11: Add register 1, 2

12: Store register 1, 800



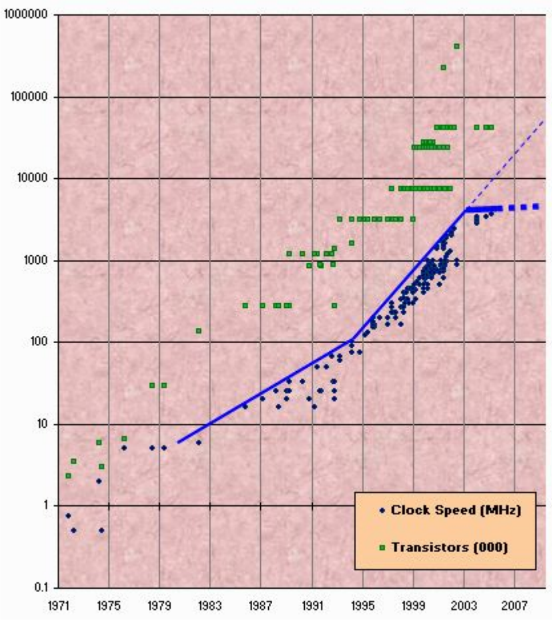
Basic uniprocessor-CPU computer. Black lines indicate data flow, red lines indicate control flow

Clock rate

- Clock rate "frequency at which CPU is running" Higher the clock rate, the faster instructions are executed.

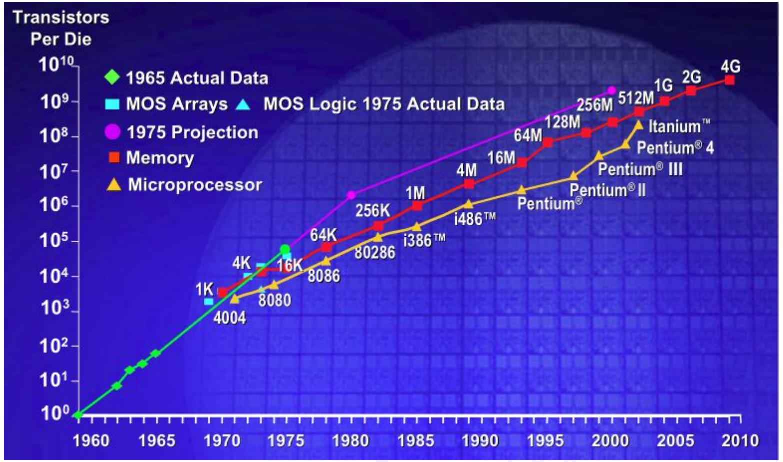
- First CPUs: 5-10 Hz (cycles per second)
- Today MacBook Pro 3.5GHz

- Your OS can control the clock rate, slow it down when idle, speed up when more work to do



Why multicore?

- Moore's Law: Computer speeds and memory densities nearly double each year

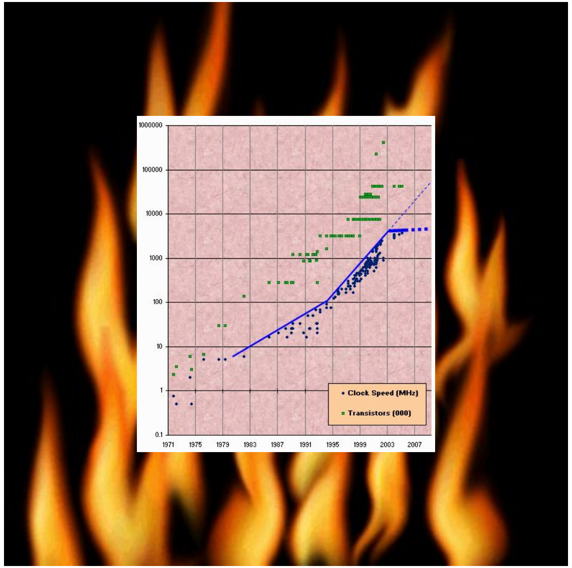


But a fast computer runs hot

- Power dissipation rises as square of the clock rate

- Chips were heading toward melting down!

- Put more CPUs on a chip: with four CPUs on one chip, even if we run each at half speed we can perform more overall computations!



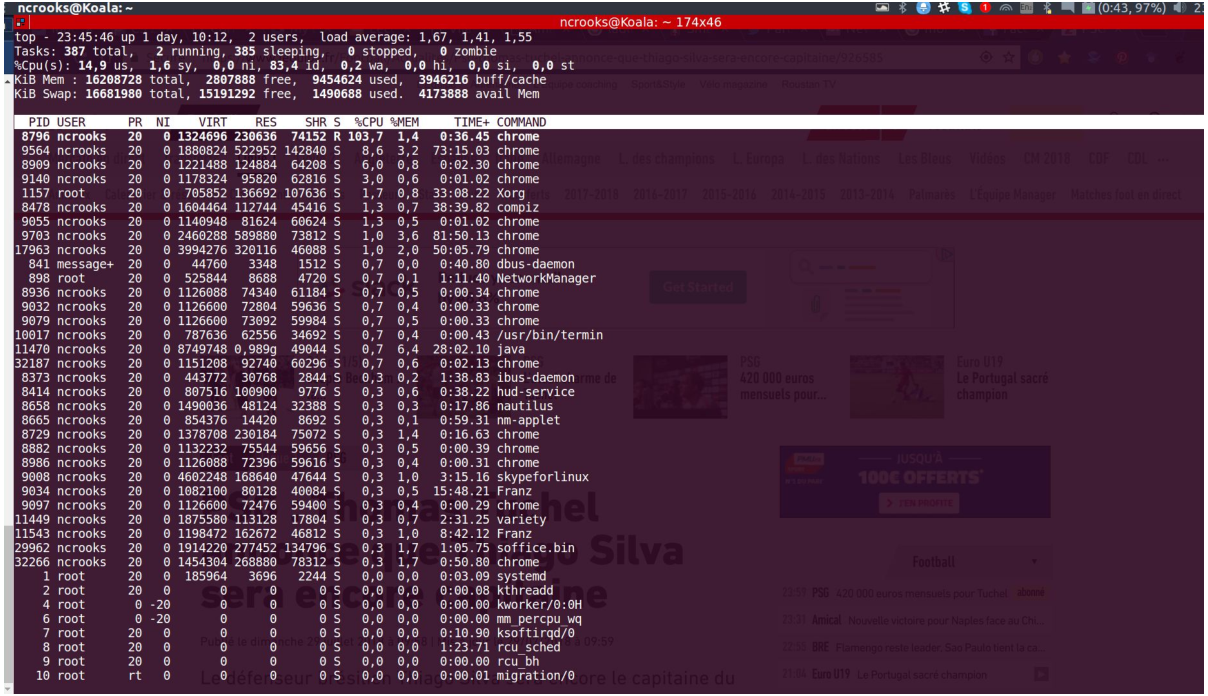
Today: Not one CPU but many

Processing Unit is called a core.

- Modern computers have "multiple cores" (processing units)
  - Instead of a single CPU (central processing unit) on the chip 5-10 common. Intel has prototypes with 80!
- We often run many programs at the same time
- Even with a single core (processing unit), your program may have more than one thing "to do" at a time
  - Argues for having a way to do many things at once

Running processes on my laptop

>100 processes are competing for time. Here's some of them:



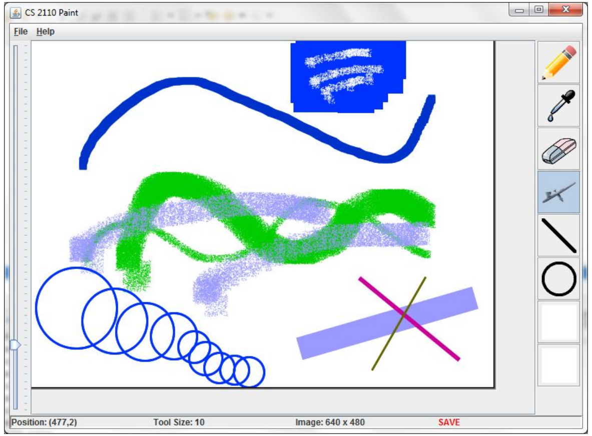
```
ncrooks@Koala: ~ 174x46
top - 23:45:46 up 1 day, 10:12,  2 users,  load average: 1,67, 1,41, 1,55
Tasks: 387 total,   2 running, 385 sleeping,   0 stopped,   0 zombie
%Cpu(s): 14,9 us,  1,6 sy,  0,0 ni, 83,4 id,  0,2 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 16208728 total,  2807888 free,  9454624 used,  3946216 buff/cache
KiB Swap: 16681980 total, 15191292 free,  1490688 used.  4173888 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 8796 ncrooks   20   0 1324696 230636  74152 R 103,7  1,4   0:36.45 chrome
 9564 ncrooks   20   0 1880824 522952 142840 S   8,6  3,2  73:15.03 chrome
 8909 ncrooks   20   0 1221488 124884  64208 S   6,0  0,8   0:02.30 chrome
 9140 ncrooks   20   0 1178324  95820  62816 S   3,0  0,6   0:01.02 chrome
 1157 root      20   0  705852 136692 107636 S   1,7  0,8  33:08.22 Xorg
 8478 ncrooks   20   0 1604464 112744  45416 S   1,3  0,7  38:39.82 compiz
 9055 ncrooks   20   0 1140948  81624  60624 S   1,3  0,5   0:01.02 chrome
 9703 ncrooks   20   0 2460288 589880  73812 S   1,0  3,6  81:50.13 chrome
17963 ncrooks   20   0 3994276 320116  46088 S   1,0  2,0  50:05.79 chrome
  841 message+  20   0   44760   3348   1512 S   0,7  0,0   0:40.80 dbus-daemon
  898 root      20   0  525844   8688   4720 S   0,7  0,1   1:11.40 NetworkManager
 8936 ncrooks   20   0 1126088  74340  61184 S   0,7  0,5   0:00.34 chrome
 9032 ncrooks   20   0 1126600  72804  59636 S   0,7  0,4   0:00.33 chrome
 9079 ncrooks   20   0 1126600  73092  59984 S   0,7  0,5   0:00.33 chrome
10017 ncrooks   20   0  787636  62556  34692 S   0,7  0,4   0:00.43 /usr/bin/termin
11470 ncrooks   20   0 8749748 0,989g  49044 S   0,7  6,4  28:02.10 java
32187 ncrooks   20   0 1151208  92740  60296 S   0,7  0,6   0:02.13 chrome
 8373 ncrooks   20   0  443772  30768   2844 S   0,3  0,2   1:38.83 ibus-daemon
 8414 ncrooks   20   0  807516 100900   9776 S   0,3  0,6   0:38.22 hud-service
 8658 ncrooks   20   0 1490036  48124  32388 S   0,3  0,3   0:17.86 nautilus
 8665 ncrooks   20   0  854376  14420   8692 S   0,3  0,1   0:59.31 nm-applet
 8729 ncrooks   20   0 1378708 230184  75072 S   0,3  1,4   0:16.63 chrome
 8882 ncrooks   20   0 1132232  75544  59656 S   0,3  0,5   0:00.39 chrome
 8986 ncrooks   20   0 1126088  72396  59616 S   0,3  0,4   0:00.31 chrome
 9008 ncrooks   20   0 4602248 168640  47644 S   0,3  1,0   3:15.16 skypeforlinux
 9034 ncrooks   20   0 1082100  80128  40084 S   0,3  0,5  15:48.21 Franz
 9097 ncrooks   20   0 1126600  72476  59400 S   0,3  0,4   0:00.29 chrome
11449 ncrooks   20   0 1875580 113128  17804 S   0,3  0,7   2:31.25 variety
11543 ncrooks   20   0 1198472 162672  46812 S   0,3  1,0   8:42.12 Franz
29962 ncrooks   20   0 1914220 277452 134796 S   0,3  1,7   1:05.75 soffice.bin
32266 ncrooks   20   0 1454304 268880  78312 S   0,3  1,7   0:50.80 chrome
    1 root      20   0  185964   3696   2244 S   0,0  0,0   0:03.09 systemd
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.08 kthreadd
    4 root       0 -20       0      0      0 S   0,0  0,0   0:00.00 kworker/0:0H
    6 root       0 -20       0      0      0 S   0,0  0,0   0:00.00 mm_percpu_wq
    7 root      20   0       0      0      0 S   0,0  0,0   0:10.90 ksoftirqd/0
    8 root      20   0       0      0      0 S   0,0  0,0   1:23.71 rcu_sched
    9 root      20   0       0      0      0 S   0,0  0,0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S   0,0  0,0   0:00.01 migration/0
```

Programs can have several "threads of execution"

We often run many programs at the same time
And each program may have several "threads of execution"

Process graphics **Process user inputs**



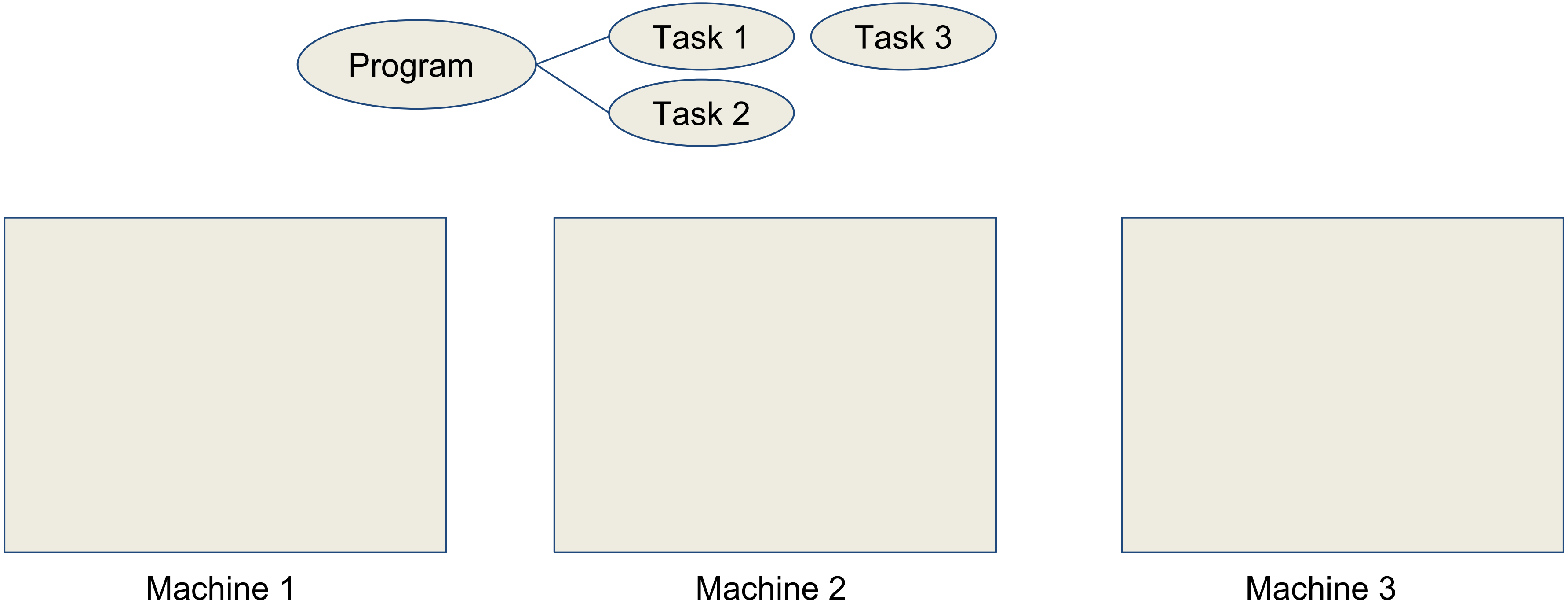
Distributed Systems

A **distributed system** is one in which the failure of a computer you didn't even know existed can render your own computer

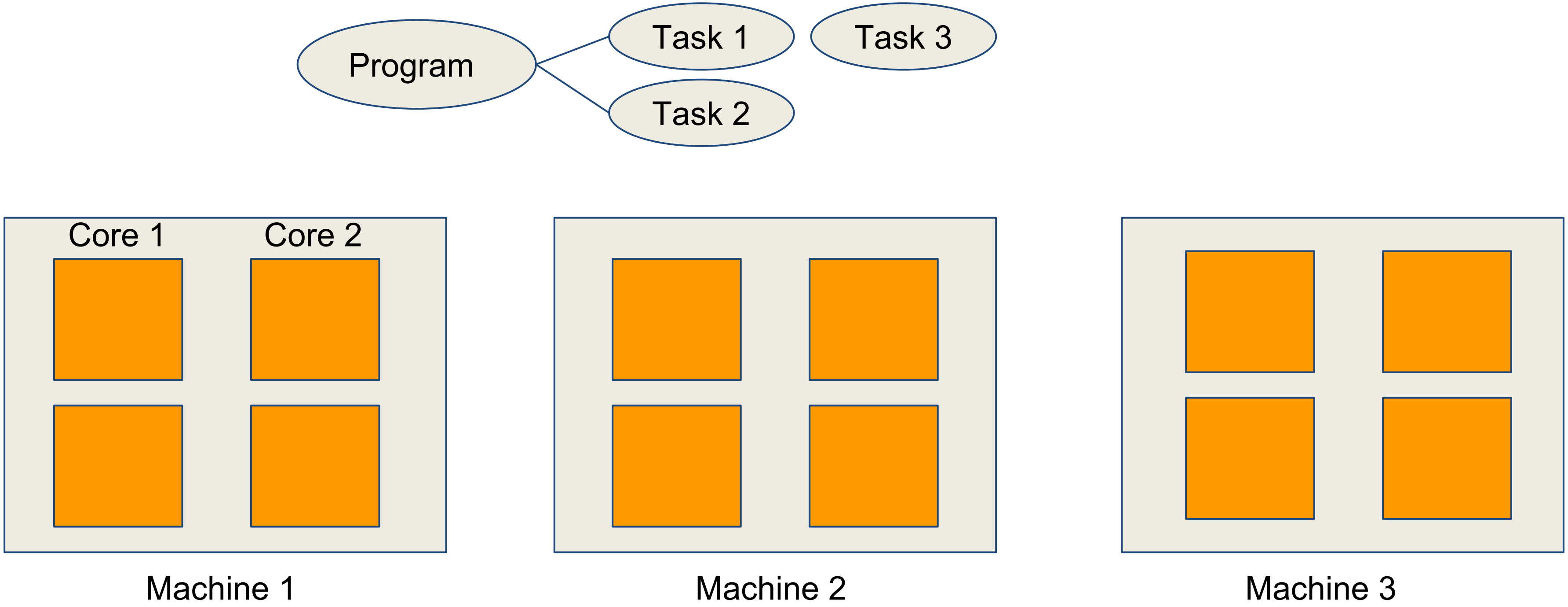
Modern systems like Google, Facebook run applications that are distributed across **thousands of machines** in **large datacenters**

My own personal record is 342 machines.

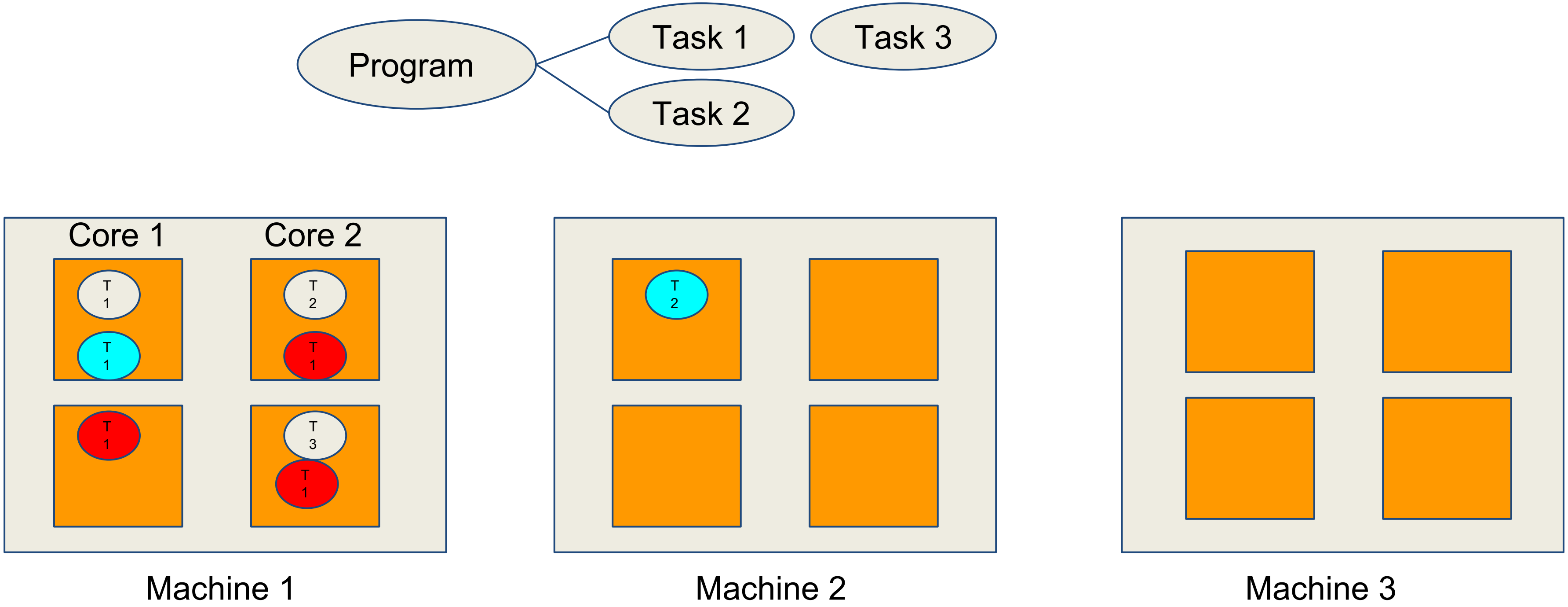
Abstract View



Abstract View



Abstract View



Concurrency

- *Concurrency* refers to a single program in which several processes, called threads, are running simultaneously
  - Special problems arise
  - They see the same data and hence can interfere with each other, e.g. one process modifies a complex structure like a heap while another is trying to read it
    - In CS2110, we'll look at:
      - Race Conditions
      - Deadlocks
- We'll refer to any sequential execution chunk as a **task**

Thread

- A **thread** is an object that "independently computes"
  - Needs to be created, like any object
  - Then "started" --causes some method to be called. It runs side by side with other threads in the same program; they see the same global data

- The actual executions could occur on different CPU cores, but don't have to
  - We can also simulate threads by *multiplexing* a smaller number of cores over a larger number of threads

Java Class Thread

- Threads are instances of class Thread
  - Can create many, but they do consume space & time

- The Java Virtual Machine creates the thread that executes your main method.

- Threads have a priority
  - Higher priority threads are executed preferentially
  - By default, newly created threads have initial priority equal to the thread that created it (but priority can be changed)

Java Class Thread

- Threads are objects in Java, just like everything else

- There's two ways to create a thread:
  - By **extending the class Thread**
  - By **implementing the interface Runnable**

- Which one do you think is better?

Creating a new Thread (Method 1)


```
class MaxThread extends Thread {
    private int[] array;

    MaxThread(int[] array) {
        this.array = array;
    }

    @Override public void run() {
        // computes max of array
        ...
    }
}
```

overrides
Thread.run()

Call **run()** directly?
no new thread is used:
Calling thread will run it

```
MaxThread p= new MaxThread(array);
p.start();
```

Do this and
Java invokes **run()** in new thread

Creating a new Thread (Method 2)


```
class MaxRun implements Runnable {
     private int[] array;

    MaxRun(int[] array) {
        this.array = array ;
    }

    public void run() {
        //compute max of an array
        ...
    }
}
```

```
MaxRun p= new MaxRun(array);
new Thread(p).start();
```

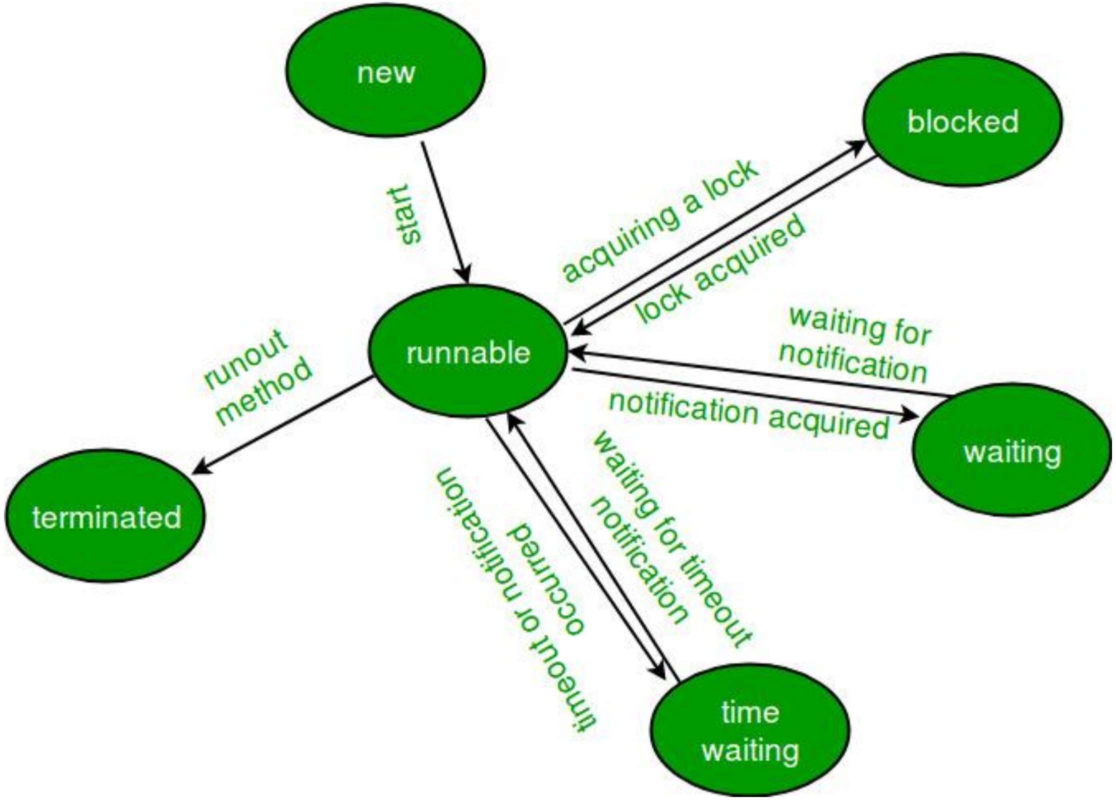
Threads can pause

- When active, a thread is "runnable".
  - It may not actually be "running". For that, a CPU must schedule it. Higher priority threads could run first.

- A thread can pause
  - Call Thread.sleep(k) to sleep for k milliseconds
    - Suspends the execution of a thread
  - Doing I/O (e.g. read file, wait for mouse input, open file) can cause thread to pause
  - Java has a form of locks associated with objects. When threads lock an object, one succeeds at a time.

- A thread can offer another thread the CPU
  - Call yield()

Thread States



How do I wait for threads to finish?

- Calling **join()** on a thread will cause another thread to wait until the first thread is finished

- Can be used to determine when the output of a computation is ready!
  - For instance, let's modify our run method to store the final max value in to an additional result array that is **shared** across all threads

- Want to know when it's safe to check the result!

How do I wait for threads to finish?

- Calling **join()** on a thread will cause another thread to wait until the first thread is finished

- Can be used to determine when the output of a computation is ready!
  - For instance, let's modify our run method to store the final max value in to an additional result array that is **shared** across all threads

- Want to know when it's safe to check the result!

```
MaxRun p1= new MaxRun(array1, result, 0);
MaxRun p2= new MaxRun(array2, result, 1);
MaxRun p3= new MaxRun(array3, result, 2);
MaxRun p4= new MaxRun(array4, result, 3);
Thread t1 = new Thread(p1).start();
Thread t2 = new Thread(p2).start();
Thread t3 = new Thread(p3).start();
Thread t4 = new Thread(p4).start();
t1.join();
t2.join();
t3.join();
t4.join();

System.out.println("Result " +
result[0] + " " + result[1] + " "…);
```

How do I wait for threads to finish?

- Calling **join()** on a thread will cause another thread to wait until the first thread is finished

- Can be used to determine when the output of a computation is ready!
  - For instance, let's modify our run method to store the final max value in to an additional result array that is **shared** across all threads

- Want to know when it's safe to check the result!

```
MaxRun p1= new MaxRun(array1, result, 0);
MaxRun p2= new MaxRun(array2, result, 1);
MaxRun p3= new MaxRun(array3, result, 2);
MaxRun p4= new MaxRun(array4, result, 3);
Thread t1 = new Thread(p1).start();
Thread t2 = new Thread(p2).start();
Thread t3 = new Thread(p3).start();
Thread t4 = new Thread(p4).start();
t1.join();
t2.join();
t3.join();
t4.join();

System.out.println("Result " +
result[0] + " " + result[1] + " " …);
```

Memory Consistency Errors

- Threads often operate on **shared data**

- If not careful, however, concurrent access to shared data can **break the correctness of the program**

- Race conditions arise both
  - at the memory level
    - memory consistency errors
  - At the program level
    - Invariants can be violated due to concurrent updates

What if threads share data?

- Threads often operate on **shared data**

- If not careful, however, concurrent access to shared data can **break the correctness of the program**

- Race conditions arise both
  - at the memory level
    - memory consistency errors
  - At the program level
    - Invariants can be violated due to concurrent updates

- Code is said to be **thread-safe** if it remains correct when accessed concurrently

Race conditions

- A **race condition** arises if two or more processes access the same variables or objects concurrently and at least one does updates

- If the updates are not **atomic**, the end state can be inconsistent
  - An operation is atomic if it happens "all at once" without being interrupted by other events.

Race conditions

- A **race condition** arises if two or more processes access the same variables or objects concurrently and at least one does updates
- If the updates are not **atomic**, the end state can be inconsistent
  - An operation is atomic if it happens "all at once" without being interrupted by other events.
- Very few operations in modern systems are atomic !

Race conditions

- Suppose x is initially 5

Thread t1	Thread t2
x = x + 1 System.out.println(x);	x = x + 1 System.out.println(x);

- What do you think will be the end value?

Race conditions

- Suppose x is initially 5

Thread t1	Thread t2
LOAD x	...
	LOAD x
ADD 1	ADD 1
	STORE x
STORE x	

- ... after finishing, x = 6! We “lost” an update

Race conditions

- Suppose x is initially 5

Thread t1	Thread t2
LOAD x	...
	LOAD x
ADD 1	ADD 1
	STORE x
STORE x	

- Machine level implementation of **increment** is not atomic!

Race conditions

- Suppose x is initially 5

Thread t1	Thread t2
LOAD x	...
	LOAD x
ADD 1	ADD 1
	STORE x
STORE x	

- Second store happens after first store -> we lost an update!

Program Correctness Errors

- What if we want to insert a new element to a linked list?

Program Correctness Errors

- What if we want to insert a new element to a linked list?

```
void add(V v) { // to tail
   Node newNode = new Node(v);
   If (tail!=null) {
          tail.next = newNode;
          newNode.prev = tail;
          tail = newNode;
   }  else {
          head = new Node(v);
          tail = head;
   }

}
```

```
V poll() { // from head
  V v = null;
  If (head!=null) {
      if (head == tail) { // list is one el
          tail = tail.prev;
      }
      elif (head.next == tail) { list is two el
          tail.prev = null
      }
      v = head.value;
      head = head.next;
      if (head!=null) {
          head.prev = null;
      }
  }
  return v;
}
```

Program Correctness Errors

- What if we want to insert a new element to a linked list?

```
void add(V v) { // to tail
   Node newNode = new Node(v);
   If (tail!=null) {
          tail.next = newNode;
          newNode.prev = tail;
          tail = newNode;
   }  else {
          head = new Node(v);
          tail = head);
   }

}
```

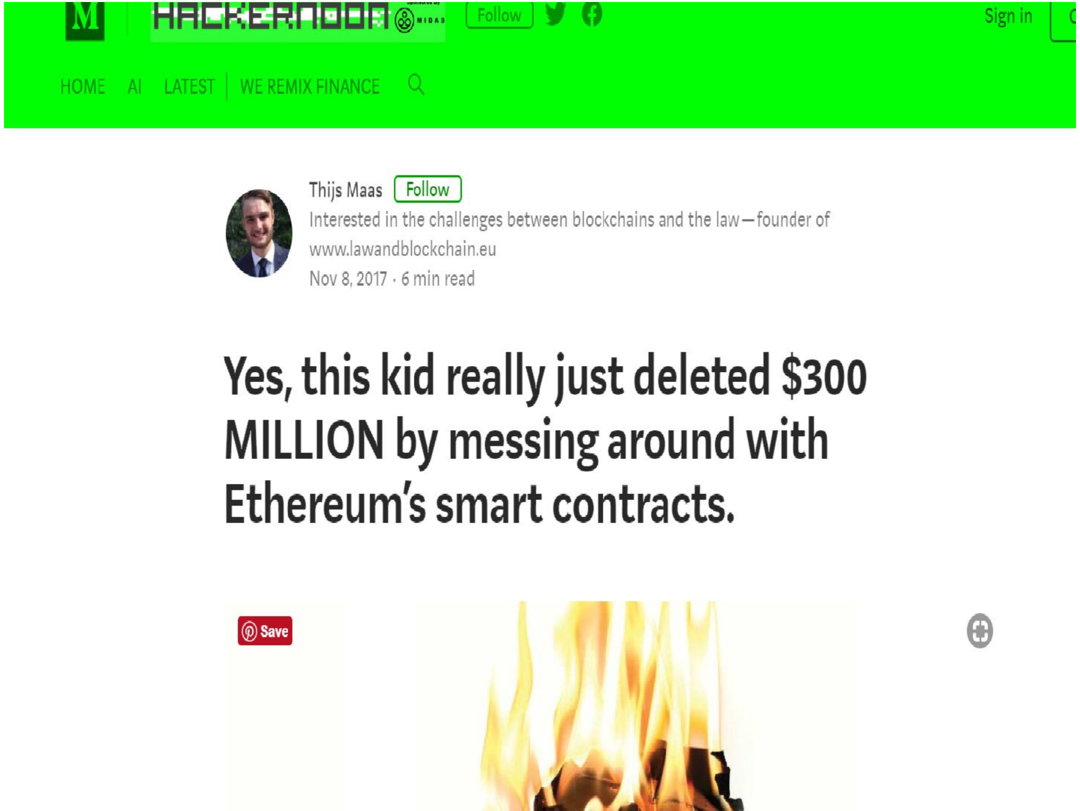
What could go wrong?

```
V poll() { // from head
  V v = null;
  If (head!=null) {
      if (head == tail) { // list is one el
          tail = tail.prev;
      }
      elif (head.next == tail) { list is two el
          tail.prev = null
      }
      v = head.value;
      head = head.next;
      if (head!=null) {
          head.prev = null;
      }
  }
  return v;
}
```

Race conditions

- Race conditions are bad news
  - Race conditions can cause many kinds of bugs, not just the example we see here!
  - Common cause for “blue screens”: null pointer exceptions, damaged data structures
  - Concurrency makes proving programs correct much harder!

Race conditions



Race conditions

Synchronization

- To prevent race conditions, one often requires a process to "acquire" resources before accessing them, and only one process can "acquire" a given resource at a time.

- This process is called **synchronization**

- Different languages provide more/less native support for synchronisation. Java provides
  - `Synchronized` primitive
  - Locks
  - Semaphores (don't look at this here)

Synchronized Keyword

- Synchronized keyword in Java acquires exclusive ownership of a given resource

- Exists in two contexts:
  - **Synchronized methods**
  - **Synchronized blocks**

- Every object in Java has an **intrinsic lock** (or **monitor lock**)

Synchronized Methods

- To make a method synchronized, simply add the synchronized keyword to its declaration

```
synchronized void add(V v) { // to tail
   Node newNode = new Node(v);
   If (tail!=null) {
         tail.next = newNode;
         newNode.prev = tail;
         tail = newNode;
   }  else {
         head = new Node(v);
         tail = head;
   }

}
```

Synchronized Methods

- To make a method synchronized, simply add the synchronized keyword to its declaration
- A synchronized method acquires exclusive ownership of the current instance of the object (**monitor lock**)
  - Ownership lasts from the beginning of the method until the end
- No two synchronized methods **on the same object** can execute concurrently

```
synchronized void add(V v) { // to tail
   Node newNode = new Node(v);
   If (tail!=null) {
          tail.next = newNode;
          newNode.prev = tail;
          tail = newNode;
   }  else {
          head = new Node(v);
          tail = head;
   }

}
```

Synchronized Methods

- Synchronized methods are great when modify only a single object

- And when are ok with locking the entire object during execution
  - Ex: currently locking the entire linked list, even if looking at different nodes

```
synchronized void add(V v) { // to tail
    Node newNode = new Node(v);
    If (tail!=null) {
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
    }  else {
            head = new Node(v);
            tail = head;
    }

}
```

Synchronized Blocks

- Unlike synchronized methods, **synchronized blocks** must specify the object that they wish to lock

- As in synchronized methods,
  - Might have to wait if other thread has acquired object.
  - While this thread is executing the synchronized block, the object is *locked*. No other thread can obtain the lock.

```
void add(V v) { // to tail
    Node newNode = new Node(v);
    synchronized (this) {
    If (tail!=null) {
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
    }  else {
            head = new Node(v);
            tail = head;
    }
   }
}
```

Revisiting the DLL

- What if we added a third method: **traverse,** that prints out all the nodes of the DLL
  - Can we achieve better performance using synchronized blocks?

Revisiting the DLL

- What if we added a third method: **traverse,** that prints out all the nodes of the DLL
  - Can we achieve better performance using synchronized blocks?

- What if we locked individual nodes in the DLL instead of locking the DLL itself?
  - How should we lock those?

Revisiting the DLL (head/tail not null)


```
void add(V v) { // to tail
   Node newNode = new Node(v);
   synchronized(head) {
     synchronized(tail) {
      If (tail!=null) {
          tail.next = newNode;
          newNode.prev = tail;
          tail = newNode;
      }  else {
          head = tail;
          tail = new Node(v);
      }
    }
   }
}
```

```
V poll() { // from head
  synchronized (head) {
     synchronized (tail) {
     V v = null;
     If (head!=null) {
      if (head == tail) { // list is one el
          tail = tail.prev;
      }
      elif (head.next == tail) { list is two el
          tail.prev = null
      }
      v = head.value;
      head = head.next;
      if (head!=null) {
          head.prev = null;
      }
  } }}
  return v;
}
```

What happens if null?

- Null objects will throw a null pointer exceptions when calling synchronized
- But if we don’t call synchronize, two threads could try to set head to non-null concurrently!
- What can we do?!

What happens if null?

- Null objects will throw a null pointer exceptions when calling synchronized

- But if we don't call synchronize, two threads could try to set head to non-null concurrently!

- What can we do?!
  - One option: make add/poll acquire a lock on the linked list to check whether head/tail is null
  - If not null, acquire lock on object, then release lock on linked list.

What happens if null?

- Null objects will throw a null pointer exceptions when calling synchronized

- But if we don't call synchronize, two threads could try to set head to non-null concurrently!

- What can we do?!
  - One option: make add/poll acquire a lock on the linked list to check whether head/tail is null
  - If not null, acquire lock on object, then release lock on linked list.
  - **BUT synchronized blocks** can only be nested

What happens if null?

- Null objects will throw a null pointer exceptions when calling synchronized

- But if we don't call synchronize, two threads could try to set head to non-null concurrently!

- What can we do?!
  - One option: make add/poll acquire a lock on the linked list to check whether head/tail is null
  - If not null, acquire lock on object, then release lock on linked list.
  - Better option, use Java **Locks**, that can never be null

Java Locks

- Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time

- Benefits:
  - decide when to acquire/release lock
  - Can "give up" on trying to acquire lock

- Lots of different types of lock in `java.util.concurrent.locks`
  - Read up!

```
private Lock headLock = new
ReentrantLock();



headLock.lock();
If (head ! = null) {
     head.lock();
     headLock.unlock();
}
else {
     Node n = new Node<E>(e);
     n.lock();
     head = n;
     headLock.unlock();
}
```

With great power …

- … Comes great responsibility

- Current locking, as we’ve seen, is hard!
  - Hard to understand what/when we should lock

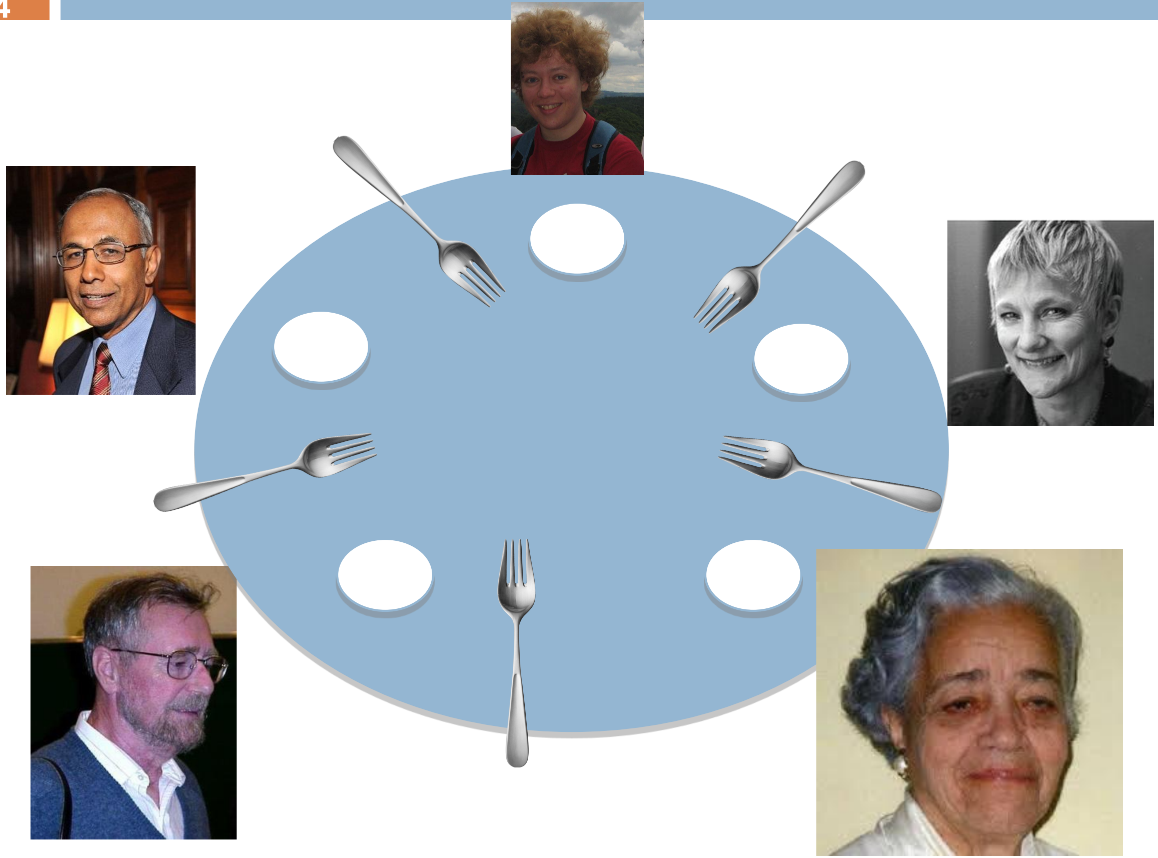
- Locking in the wrong order can lead to **deadlocks**
  - What if we synchronize/lock first on **head** then on **tail** in one method, but on **tail** then on **head** in another method?

Deadlock

- To prevent race conditions, one often requires a process to “acquire” resources before accessing them, and only one process can “acquire” a given resource at a time.

- But if processes have to acquire two or more resources at the same time in order to do their work, **deadlock** can occur. This is the subject of the next slides.

Dining philosopher problem



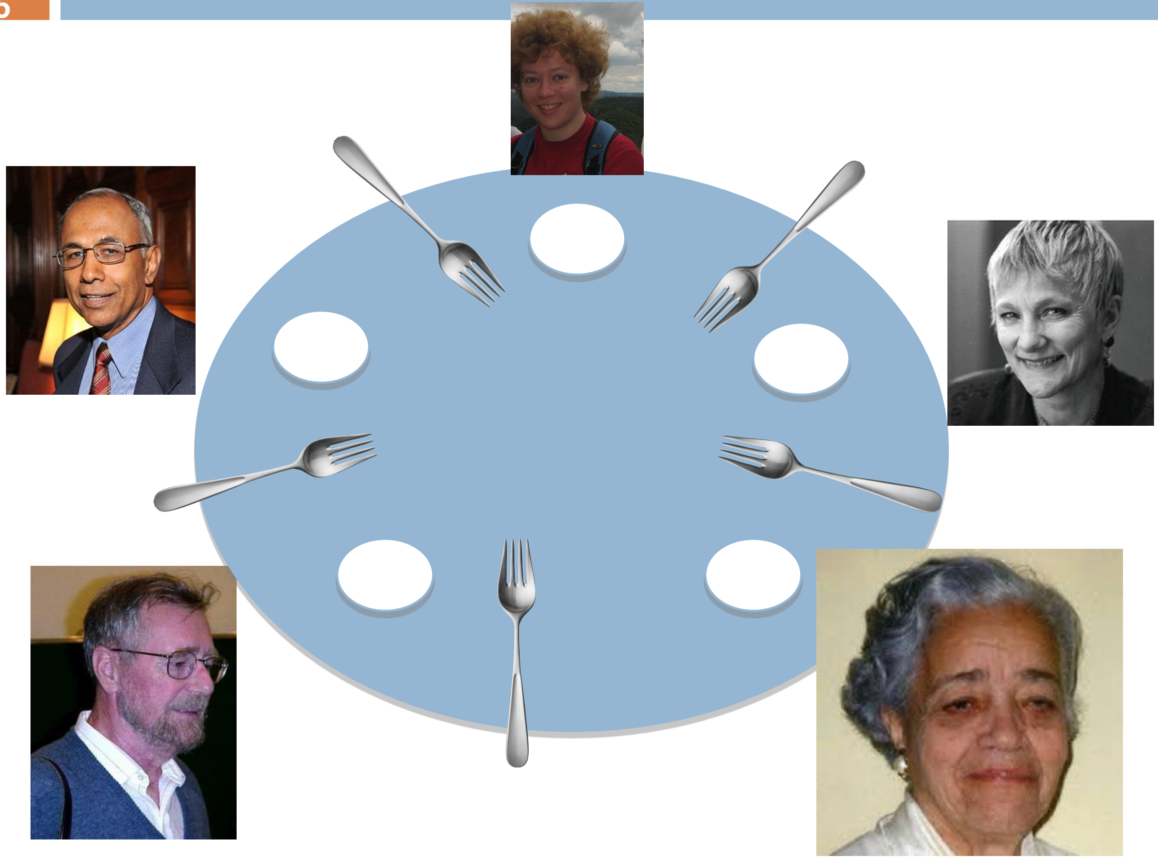
Five philosophers sitting at a table.

Each repeatedly does this:

1. think
2. eat.

Need TWO forks to eat spaghetti!

Dining philosopher problem



Five philosophers sitting at a table.

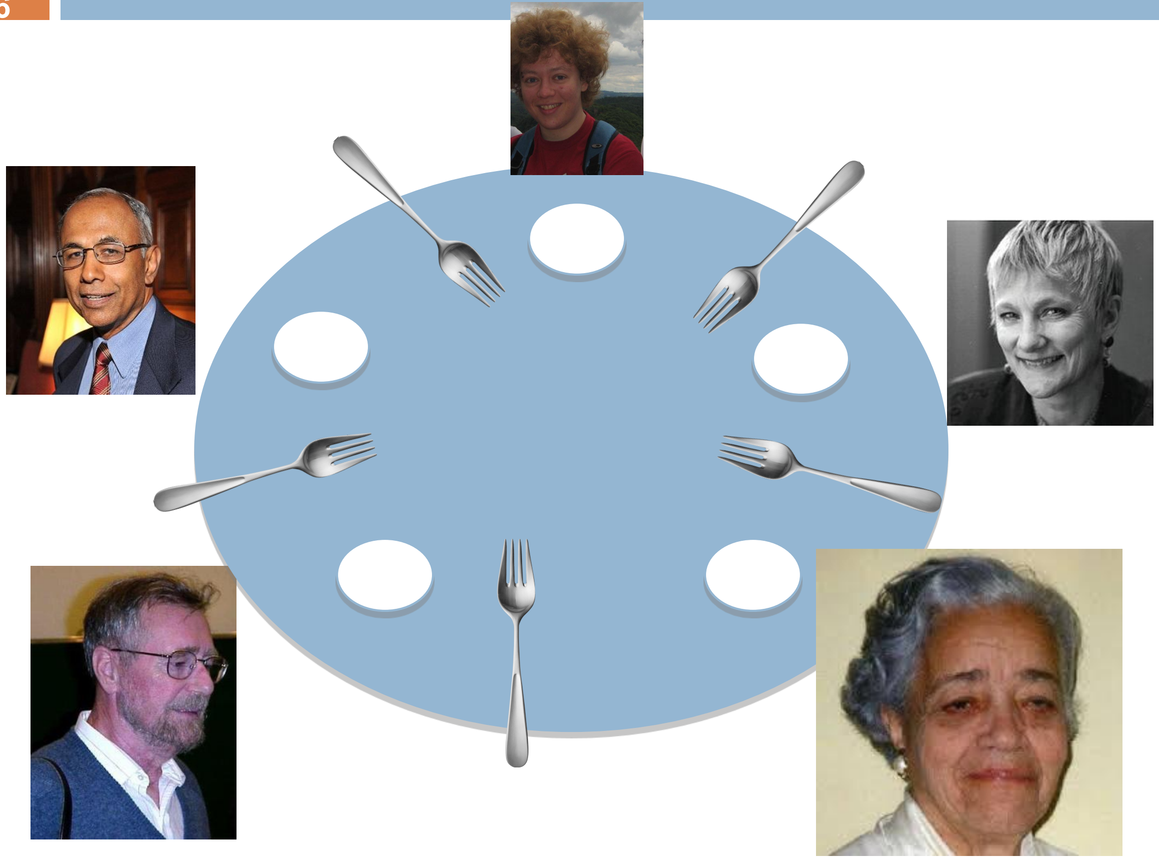
Each repeatedly does this:

1. think
2. Eat.

Only brought 5 forks!

Need TWO forks to eat spaghetti!

Dining philosopher problem

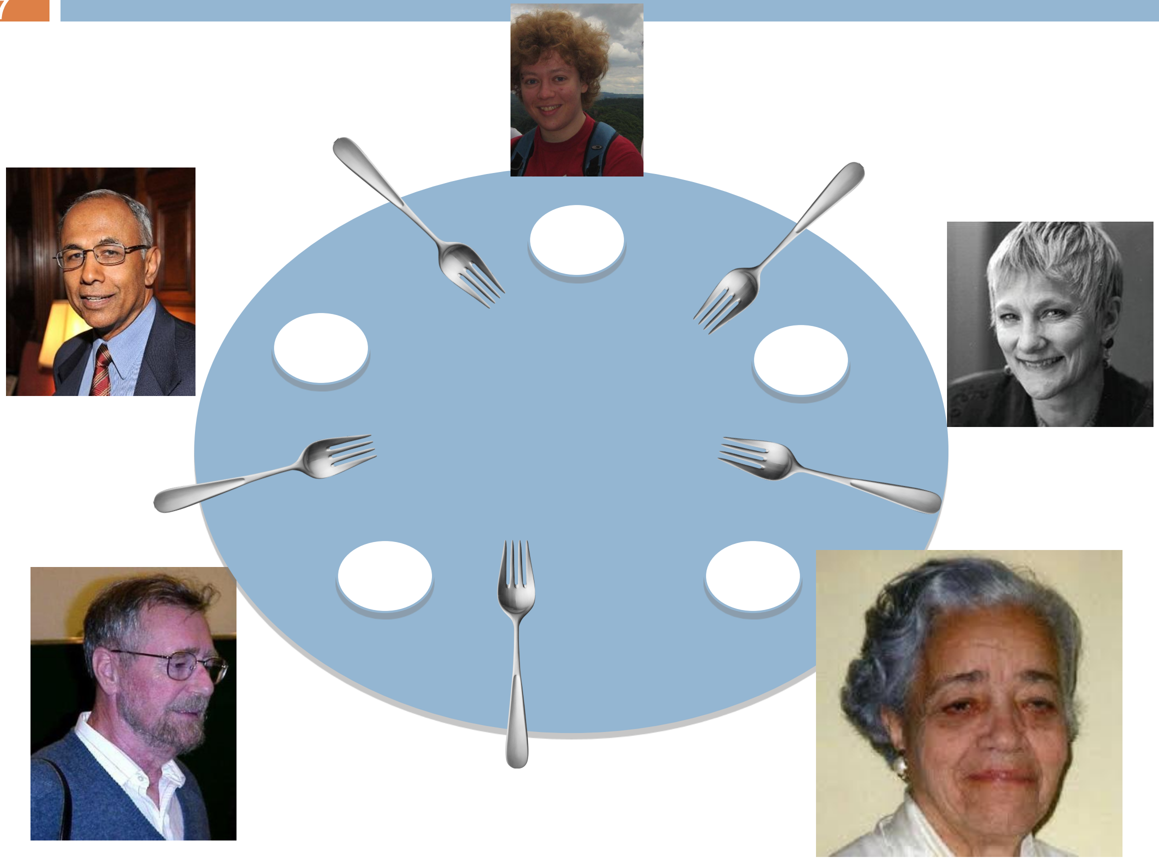


Five philosophers sitting at a table.

To eat, they first pick up the **left fork**, then the **right fork**, then **eat,** then put the **left fork** down, then put the **right fork** down.

Need TWO forks to eat spaghetti!

Dining philosopher problem

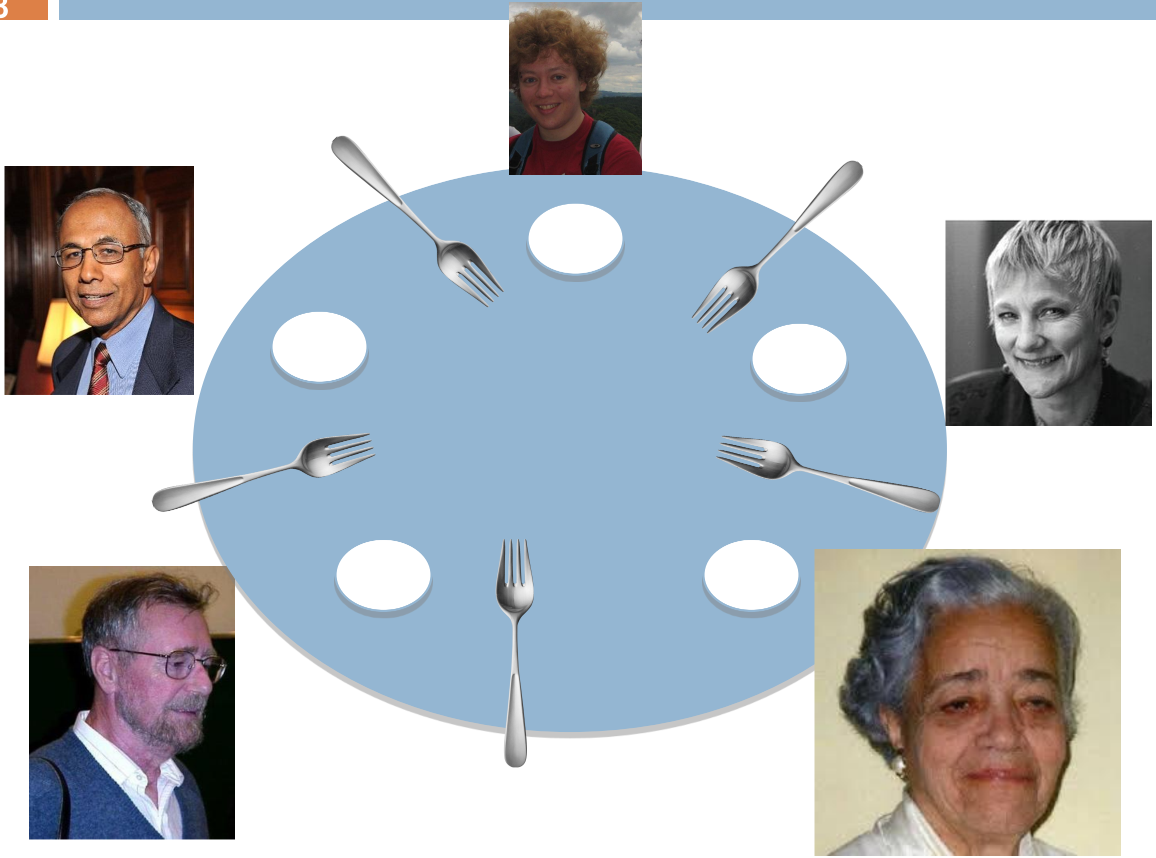


Five philosophers sitting at a table.

At one point they all pick up their **left fork**!

Need TWO forks to eat spaghetti!

Dining philosopher problem



Five philosophers sitting at a table.

At one point they all pick up their **left fork**!

We have a deadlock!

Need TWO forks to eat spaghetti!

Dining philosopher problem

Simple solution to deadlock:

Number the forks. Pick up smaller one first

1. think
2. eat (2 forks)

eat is then:

pick up smaller fork
pick up bigger fork
pick up food, eat
put down bigger fork
put down smaller fork

Correct Locking is Hard!

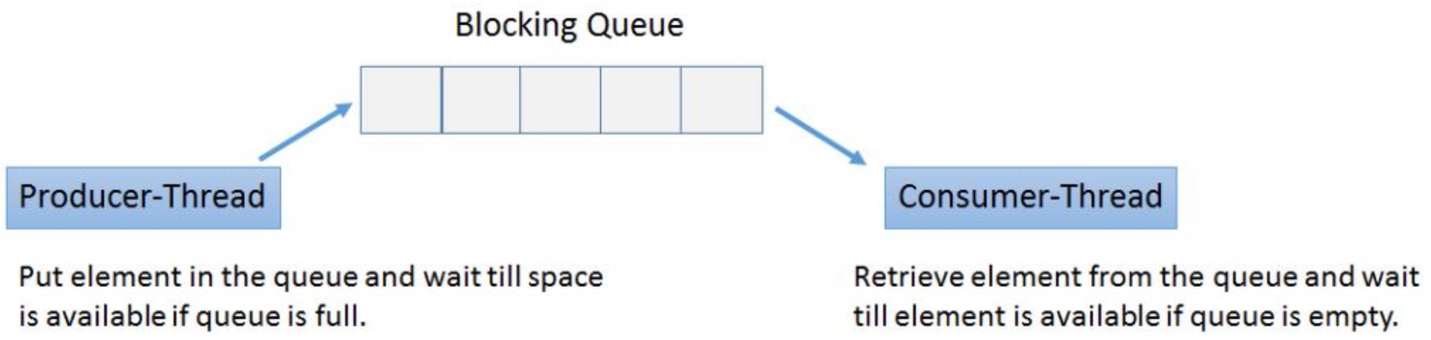
- Locking objects in different orders in different functions will cause deadlock!
- Exceptions that occur in the middle of the program may cause locks to not be released
- Insufficient locking may lead to race conditions!

Good practices

- Prefer the use of `Lock` locks over synchronized.
- When in doubt, use a lock for the whole method!
  - Only optimise when you need to
  - Correct code is always faster than incorrect code :-)
- Always acquire locks at the beginning of the method unless a good reason not to
- Always try to release locks in a finally clause

Thread Coordination

- Threads often have to **coordinate** their actions
  - Example 1: Thread 1 (thread that monitors user input) must **notify** thread 2 (the GUI thread) that there are new characters to draw. Thread 2 is **waiting** for Thread 1's notification.
  - Example 2: producer/consumer pattern



Option 1: Busy waiting


```
while (dll.isEmpty()) {
     // Do nothing
}
// If exited the loop, means
// element was in thread
V v  = dll.poll();
```

Loop until condition is satisfied.
Only then do you exit the loop

Option 1: Busy waiting


```
while (dll.isEmpty()) {
    // Do nothing
}
// If exited the loop, means
// element was in thread
V v  = dll.poll();
```

Loop until condition is satisfied. Only then do you exit the loop

Inefficient. Not necessary to constantly re-check condition. Keeping thread busy for no reason

Option 2: Guarded Blocks on Objects


```
synchronized(this) {
    while (dll.isEmpty()) {
    // Do nothing
        this.wait();
    }
    V v = dll.poll();
}
```

Suspend the current thread until condition is satisfied by calling **Object.wait()**

The invocation of **wait** does not return until another thread has issued a **notification** that some special event may have occurred — though not necessarily the event this thread is waiting for:

Option 2: Guarded Blocks on Objects


```
synchronized(dll) {
    while (dll.isEmpty()) {
    // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}
```

Calling **wait()** blocks the current thread. Thread releases the **monitor** lock of the **dll** object. It will re-acquire it when receiving the notification

Option 2: Guarded Blocks on Objects


```
synchronized(dll) {
    while (dll.isEmpty()) {
    // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}
```

Calling **wait()** blocks/suspends the current thread. Thread releases the **monitor** lock of the **dll** object. It will re-acquire it when receiving the notification

Option 2: Guarded Blocks on Objects


```
synchronized(dll) {
    while (dll.isEmpty()) {
    // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}
```

One should always invoke **wait** **inside a loop** that tests for the condition being waited for. Threads can be woken up for a number of reasons. Notification may not be for the particular condition that current thread was waiting for.

Option 2: Guarded Blocks on Objects


```
synchronized(dll) {
    while (dll.isEmpty()) {
    // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}
```

```
synchronized(dll) {
    dll.insert(v);
    dll.notifyAll();
}
```

Notifications are sent using the **notify** or **notifyAll** keywords

notifyAll wakes up **all threads** waiting on that lock that something important happened.

notify() wakes up a **single thread**.

Option 3: Guarded Blocks with Locks


```
  Lock lock = new
               ReentrantLock();
  Condition hasElement =
lock.newCondition();
```

```
     dllLock.lock();
     while (dll.isEmpty()){
          hasElement.await();
     }
     V v = dll.poll();
     dllLock.unlock();
```

```
     dllLock.lock();
     dll.insert(v);
     hasElement.notifyAll();
     dllLock.unlock();
```

Locks are associated with **Conditions** that support **await** and **notify/notifyAll** methods.

Can associate as many conditions per locks as desired

Main benefit: more flexibility. Make it explicit what condition you are awaiting on

Java Concurrent Collections

- BlockingDeque<E>
  - A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.

- BlockingQueue<E>
  - A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

- ConcurrentMap<K,V>
  - A Map providing thread safety and atomicity guarantees.

Concurrency

- Brief overview of concurrency in Java!
- More formal treatment in higher level courses.
- Remember: **monitor locks**, **synchronized**, **wait**, **notify/All**, **conditions**, **race conditions**, **deadlocks**