Parallel Programming

Introduction to Threads and Synchronization

Structure of Next Lectures

Motivation: Parallelism is Tricky but Useful

Multiprocessing vs. Multithreading

Java Threads: Creation, Status, Join

Shared Resources, Thread Interleavings

Synchronization with `synchronize` Blocks

Coordination/Communication: Producer-Consumer with `wait` & `notify`

Parallelism: an analogy

Wake-up

Parallelism: an analogy

Wake-up

Get out of bed

Parallelism: an analogy

Wake-up

Get out of bed

Brush teeth

Parallelism: an analogy

Wake-up

Get out of bed

Brush teeth

Get dressed

Parallelism: an analogy

Wake-up

Get out of bed

Brush teeth

Get dressed

Make coffee

Make toast

Parallelism: an analogy

Wake-up

Get out of bed

Brush teeth

Get dressed

Sequential

Make coffee

Make toast

Parallel

Parallelism: an analogy (continued)

The bad news:
Parallelism is tricky!

Magic Trick (1)

Pick one card from the six cards below:

Focus on just that card!

Magic Trick (2)

I’ve shuffled the cards and removed the one which I think was your card.

Can you still remember your card?

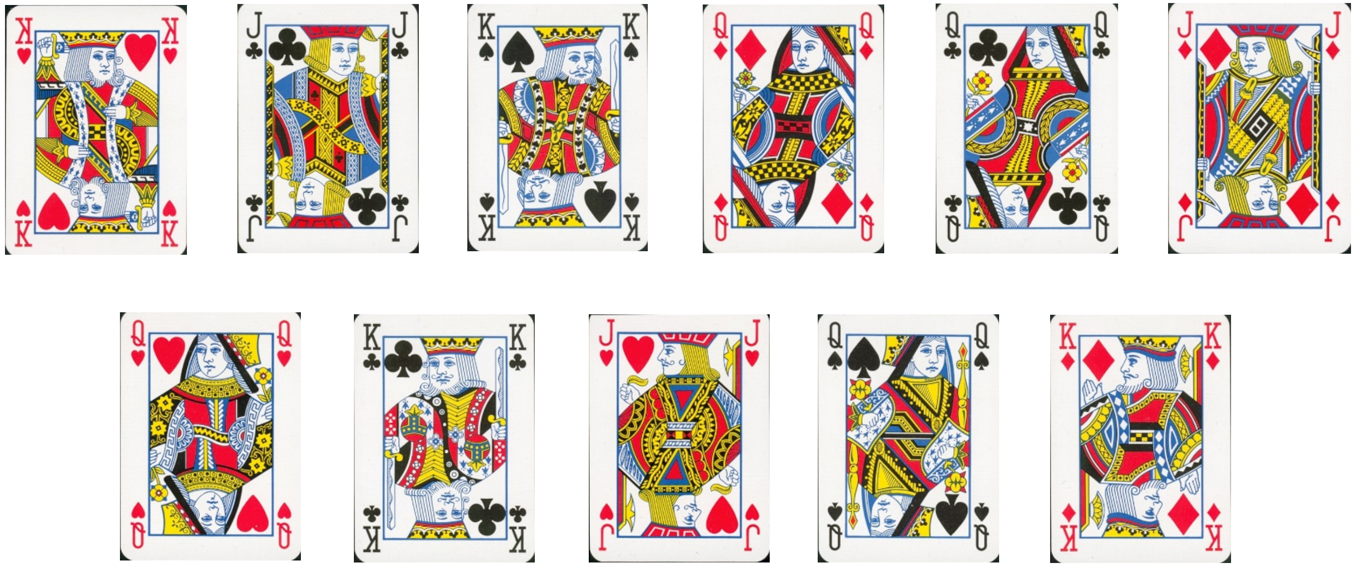
Magic Trick (3)

Here are the remaining five cards, is your card there?

Did I guess right? Or is it an illusion?

Magic Trick – The Explanation

- You just experienced ***Inattentional Blindness***
- ***None*** of the original six cards was displayed!



Take Home Message: You can’t do two things at a time

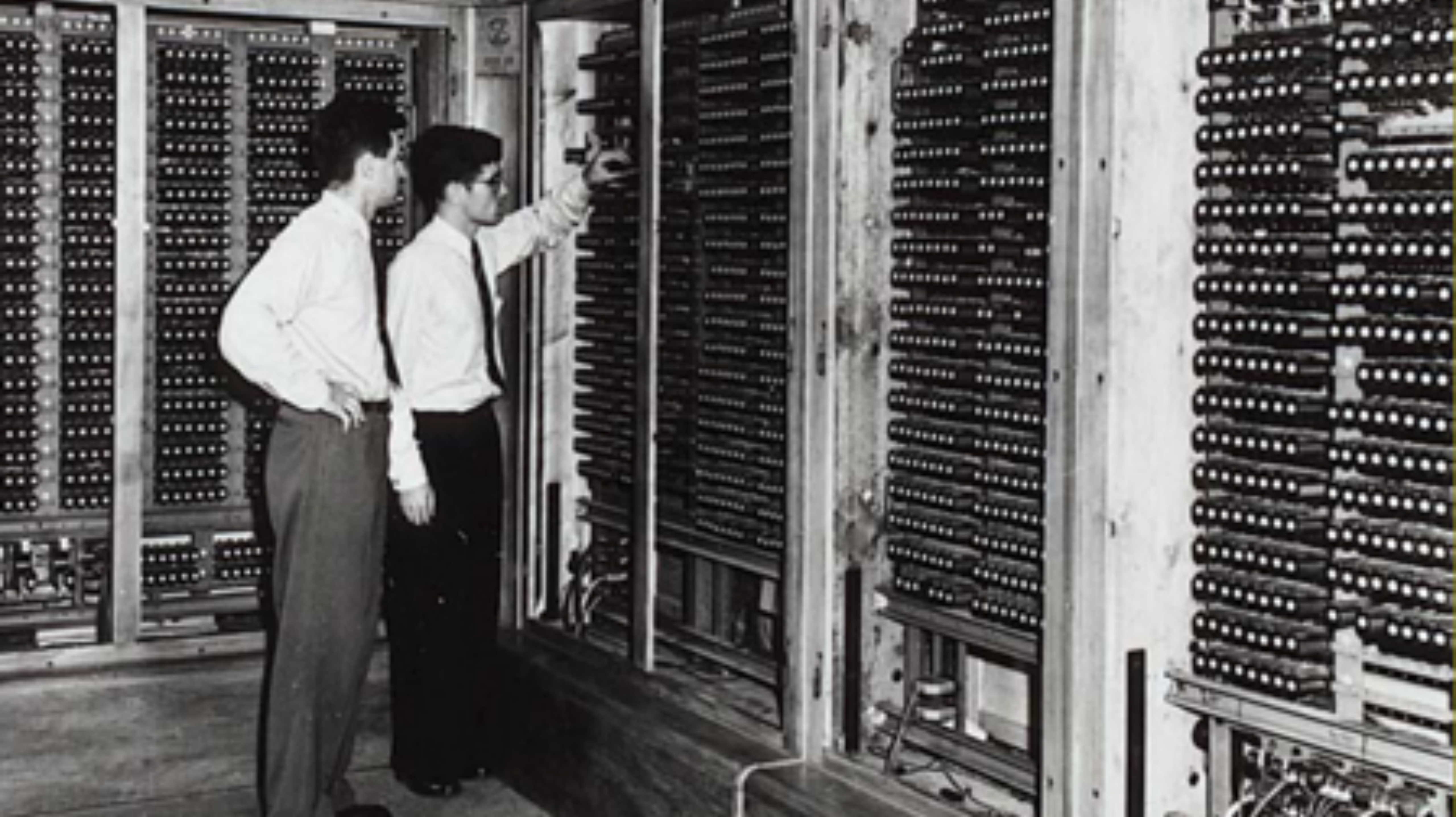
Attention involves selective processing of visual information

Our brain is not “made for” doing things in parallel (or thinking of parallelism).

If attention is elsewhere (even temporarily), changes can be missed
→ implication?

- Driving!
- Laptop in Class!

The good news:
Parallelism is useful!



Multitasking/Multiprocessing

Multitasking

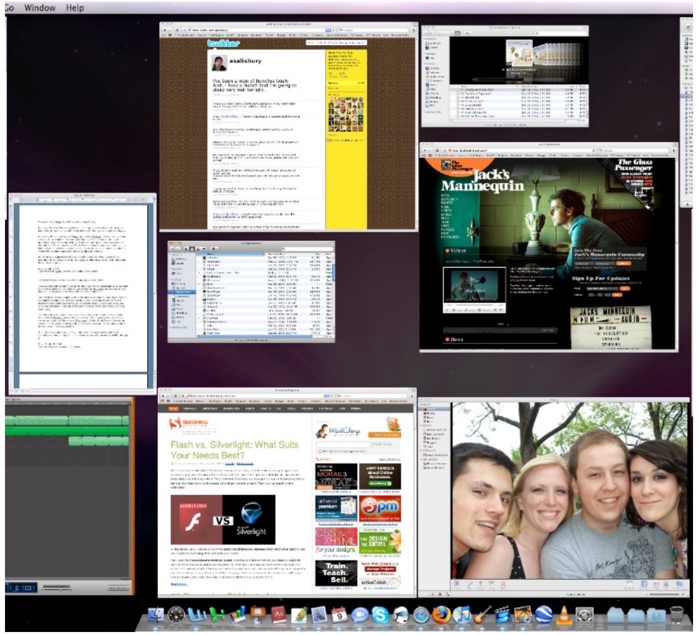
Concurrent execution of multiple tasks/processes

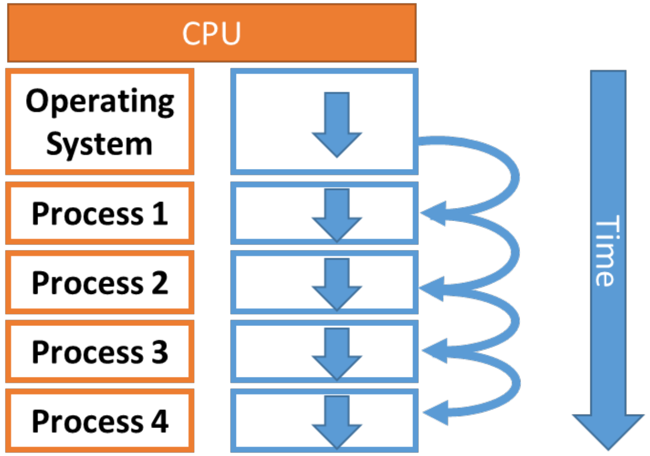
Time multiplexing of CPU

- Creates impression of parallelism
- Even on single core/CPU system

Allows for asynchronous I/O

- I/O devices and CPU are truly parallel
- 10ms waiting for HDD allows other processes to execute $>10^{10}$ instructions





Process context

A process is (essentially) a program executing inside an OS

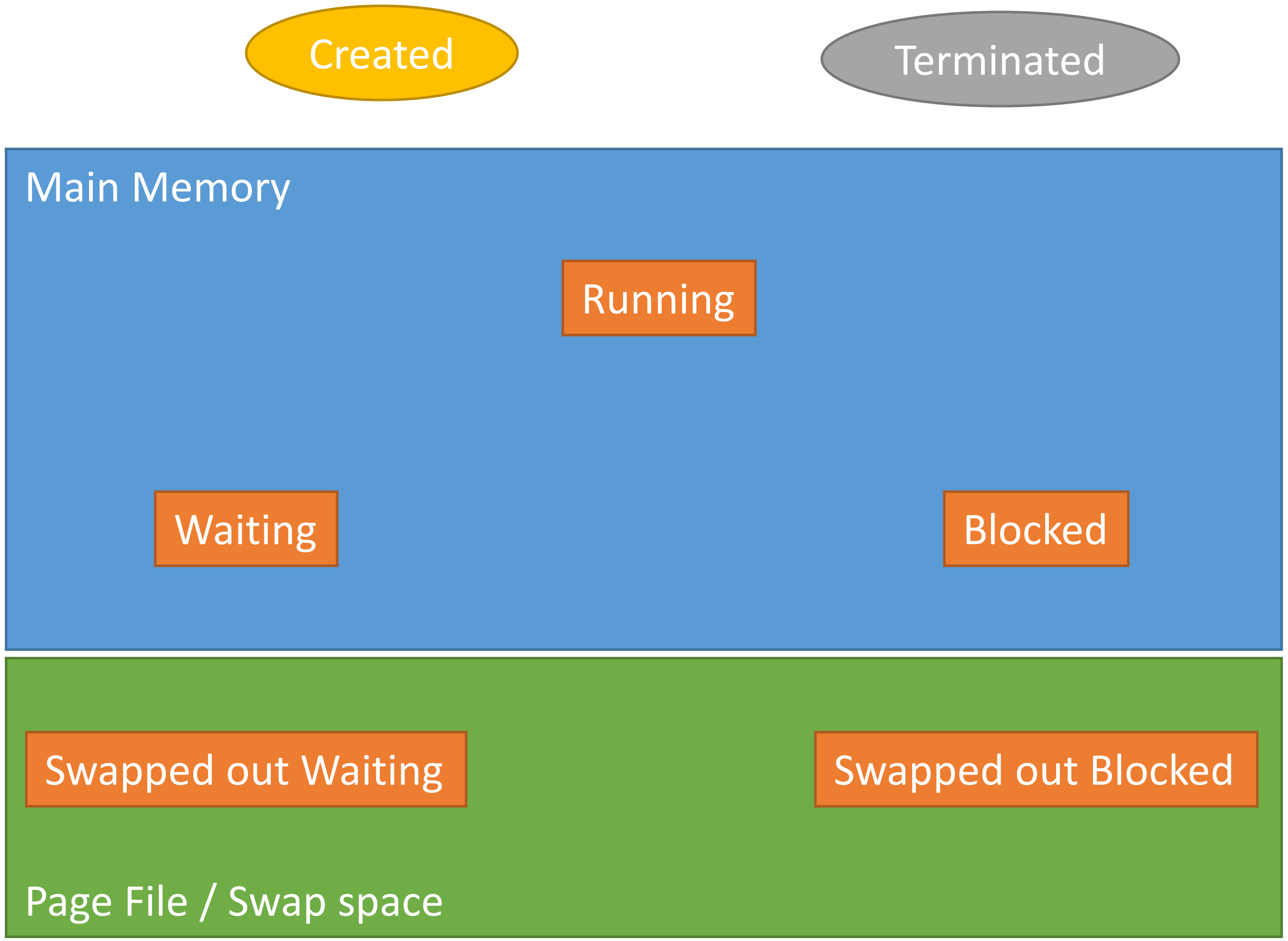
Each running instances of a program (e.g., multiple browser windows) is a separate process

Multiple applications (=processes) in parallel

Each process has a context:

- Instruction counter
- Values in registers, stack and heap
- Resource handles (device access, open files)
- …

Process lifecycle states



Process management

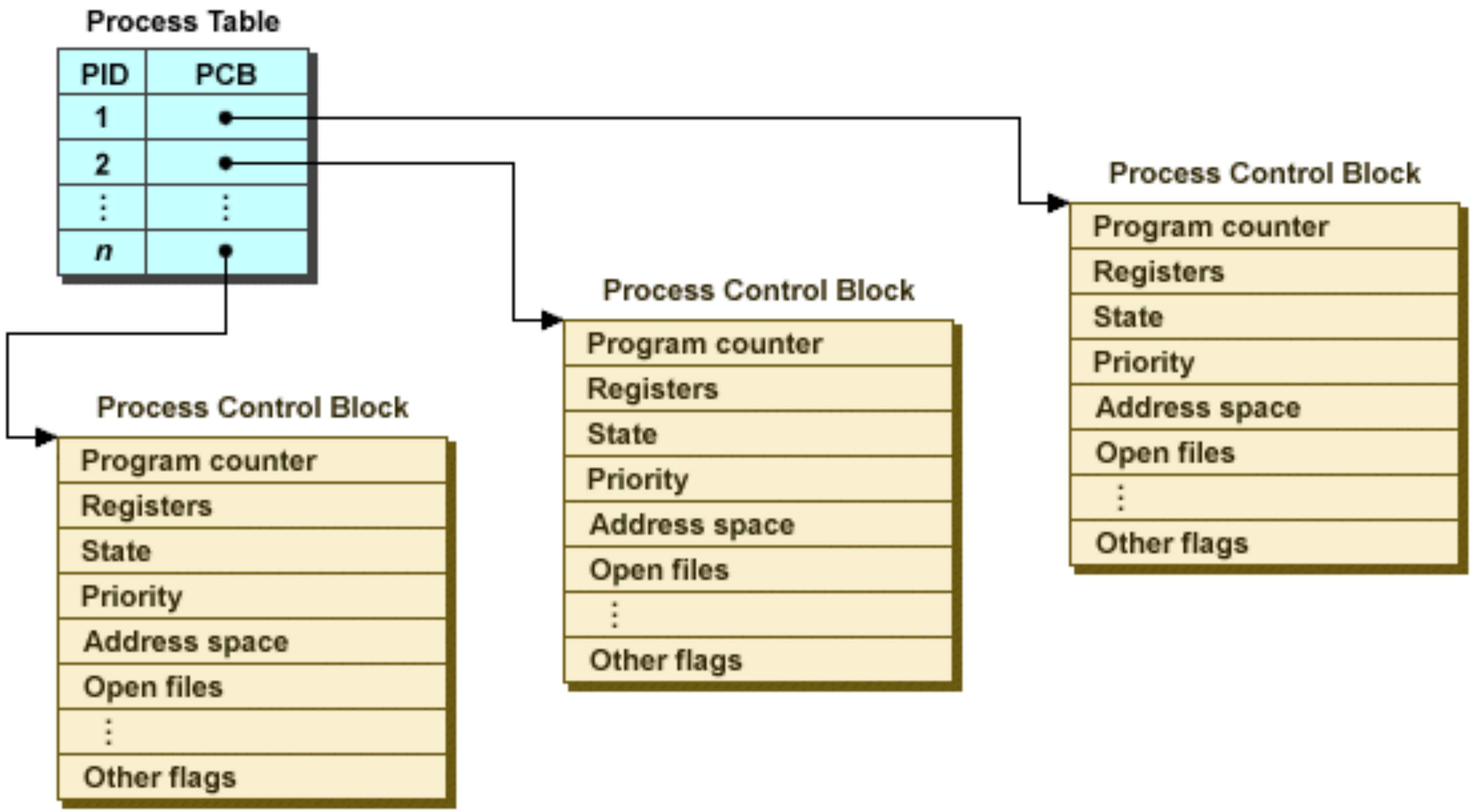
Processes need resources

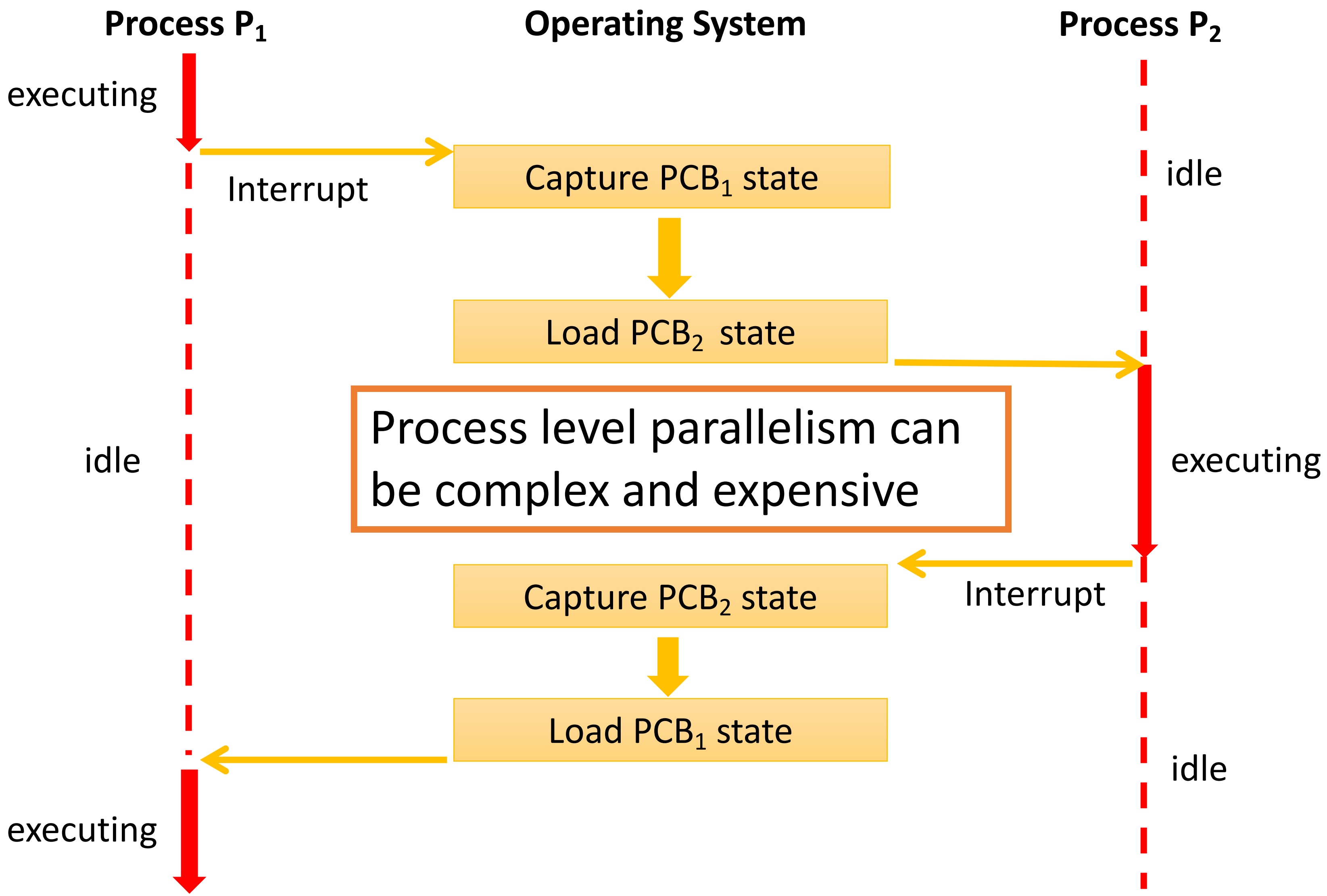
- CPU time, Memory, etc.

OS manages processes:

- Starts processes
- Terminates processes (frees resources)
- Controls resource usage (prevents monopolizing CPU time)
- Schedules CPU time
- Synchronizes processes if necessary
- Allows for inter process communication

Process control blocks (PCB)




Process P_1
Operating System
Process P_2
executing
Interrupt
Capture PCB_1 state
idle
Load PCB_2 state
Process level parallelism can be complex and expensive
idle
executing
Capture PCB_2 state
Interrupt
Load PCB_1 state
idle
executing

Multithreading

Threads

Threads (of control) are

- independent sequences of execution
- running in the same OS process

Multiple threads share the same address space.

- Threads are not shielded from each other
- Threads share resources and can communicate more easily

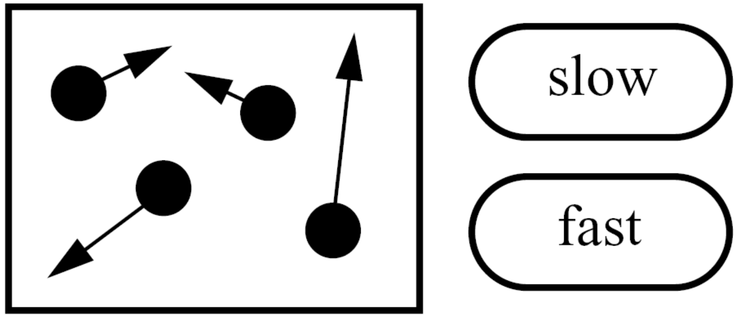
More vulnerable for programming mistakes

Context switching between threads is efficient

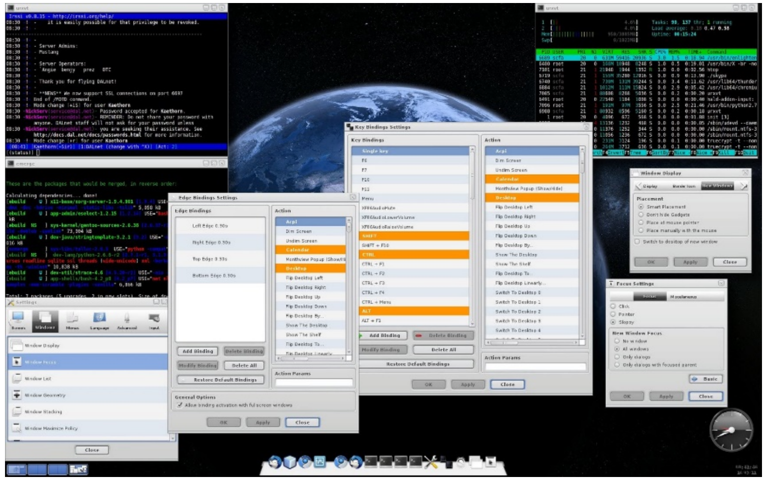
- No change of address space
- No automatic scheduling
- No saving / (re-)loading of PCB (OS process) state

Usage of Multithreading

Reactive systems – constantly monitoring



More responsive to user input – GUI application can interrupt a time-consuming task

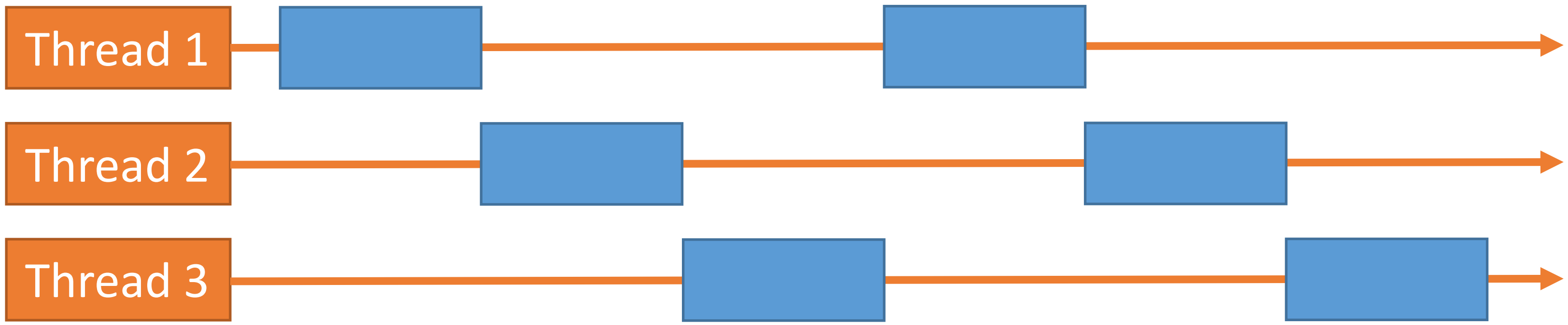


Server can handle multiple clients simultaneously

Take advantage of multiple CPUs/cores

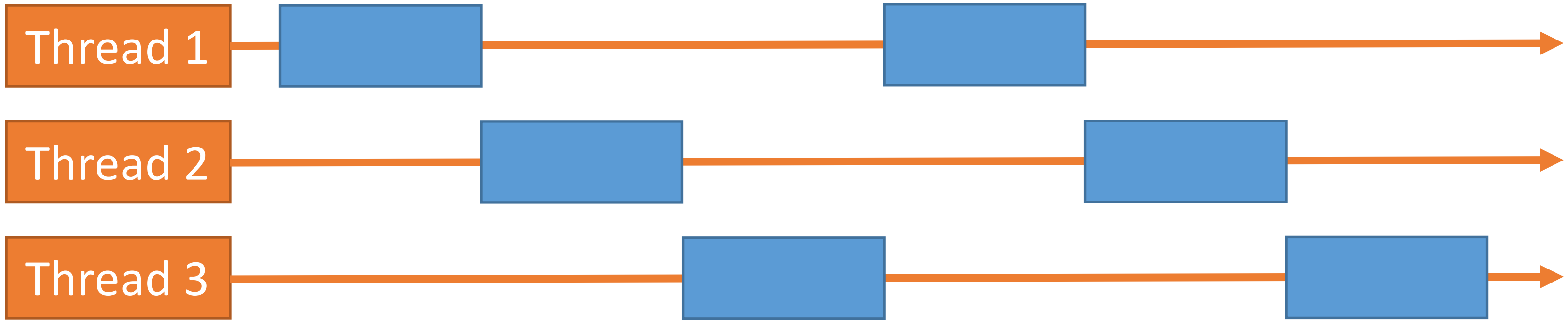
Multithreading: 1 vs. many CPUs

Multiple threads sharing a single CPU



Multithreading: 1 vs. many CPUs

Multiple threads sharing a single CPU



Multiple threads on multiple CPUs

Java Threads

Java Threads

Thread

- A set of instructions to be executed one at a time, in a specified order
- A special Thread class is part of the core language

(Some) methods of class java.lang.Thread

- start() : method called to spawn a new thread
    - Causes JVM to call run() method on object
- interrupt() : freeze and throw exception to thread

Create Java Threads: Option 1 (oldest)

Instantiate a subclass of `java.lang.Thread` class

- Override run method (must be overridden)
- run() is called when execution of that thread begins
- A thread terminates when run() returns
- start() method invokes run()
- Calling run() does not create a new thread

```
class ConcurrWriter extends Thread { …
    public void run() {
        // code here executes concurrently with caller
    }
}
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start();   // calls ConcurrWriter.run()
```

Creating the Thread object does not start the thread!

Need to actually call start() to start it.

Create Java Threads: Option 2 (better)

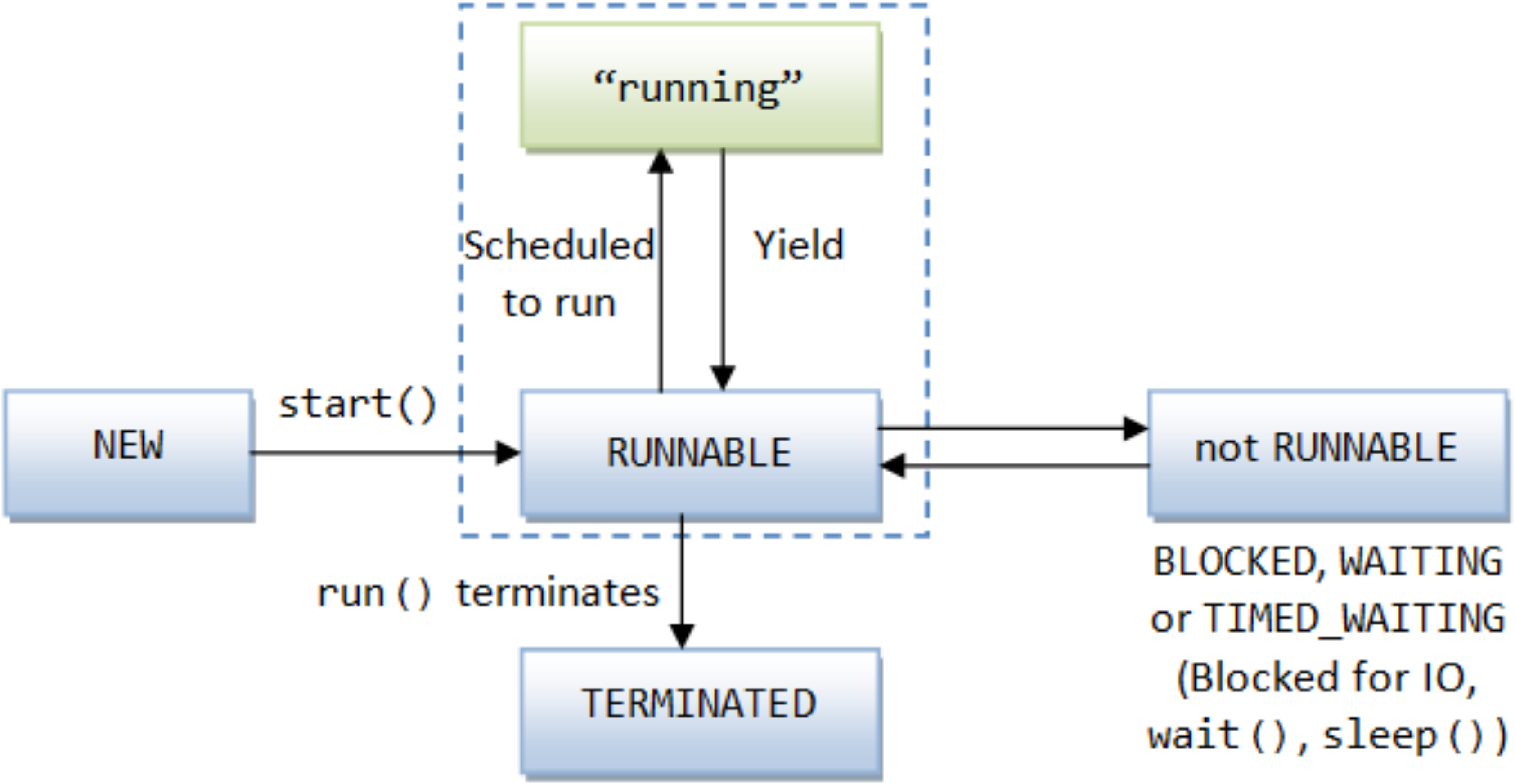
Implement `java.lang.Runnable`

- Single method: public void run()
- Class implements Runnable

```
public class ConcurrWriter implements Runnable {
     …
     public void run() { …
        // code here executes concurrently with caller
      }
}

ConcurrReader readerThread = new ConcurrReader();
Thread t = new Thread(readerThread);
t.start();   // calls ConcurrWriter.run()
```

Thread state model in Java



http://pervasive2.morselli.unimo.it/~nicola/courses/IngegneriaDelSoftware/java/J5e_multithreading.html

java.lang.Thread (under the hood)

```java
// Thread.java from OpenJDK:
// https://hg.openjdk.java.net/jdk/jdk/file/tip/src/java.base/share/classes/java/lang/Thread.java
public class Thread implements Runnable {
    static { registerNatives(); }

    private volatile String name;
    private int priority;

    private boolean daemon = false;

    ...

    public static native void yield();
    public static native void sleep(long millis) throws InterruptedException;

    private Thread(...) { ... }

    public synchronized void start() { ... }

    private native void start0();

    ...
```

A Thread is Runnable

Creates execution environment for the thread
(sets up a separate run-time stack, etc.)

java.lang.Thread (under the hood)

```java
// Thread.java from OpenJDK:
// https://hg.openjdk.java.net/jdk/jdk/fi
public class Thread implements Runnable {
    static { registerNatives(); }

    private volatile String name;
    private int priority;

    private boolean daemon = false;

    ...

    public static native void yield();
    public static native void sleep(long

    private Thread(...) { ... }

    public synchronized void start() { ..

    private native void start0();

    ...
```

```c
// Thread.c from OpenJDK:
// https://hg.openjdk.java.net/jdk/jdk/file/tip/src/java.base/share/native/libjava/Thread.c

#include "jni.h"
#include "jvm.h"

#include "java_lang_Thread.h"

#define THD "Ljava/lang/Thread;"
#define OBJ "Ljava/lang/Object;"
#define STE "Ljava/lang/StackTraceElement;"
#define STR "Ljava/lang/String;"

#define ARRAY_LENGTH(a) (sizeof(a)/sizeof(a[0]))

static JNINativeMethod methods[] = {
    {"start0",          "()V",        (void *)&JVM_StartThread},
    ...
    {"yield",           "()V",        (void *)&JVM_Yield},
    {"sleep",           "(J)V",       (void *)&JVM_Sleep},
    ...
```

Native C implementation of Java's `native` thread methods

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1 -10

```
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
        this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1 -10

```java
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
        this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

```java
public static void main(String[] args) {
//Launch 10 threads that make the operation
with a different number
  for (int i=1; i <= 10; i++){
    Calculator calculator = new Calculator(i);
    Thread thread = new Thread(calculator);
    thread.start();
}}
```

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1 -10

```
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
        this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

```
public static void main(String[] args) {
//Launch 10 threads that make the operation
with a different number
  for (int i=1; i <= 10; i++){
    Calculator calculator = new Calculator(i);
    Thread thread = new Thread(calculator);
    thread.start();
}}
```

Sample output:

```
....
Thread-9: 10 * 10 = 100
Thread-4: 5 * 8 = 40
Thread-4: 5 * 9 = 45
Thread-4: 5 * 10 = 50
Thread-5: 6 * 7 = 42
Thread-2: 3 * 4 = 12
Thread-5: 6 * 8 = 48
Thread-0: 1 * 5 = 5
....
```

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1 -10

```
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
        this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

```
public static void main(String[] args) {
//Launch 10 threads that make the operation
with a different number
  for (int i=1; i <= 10; i++){
    Calculator calculator = new Calculator(i);
    Thread thread = new Thread(calculator);
    thread.start();
}}
```

Sample output:

....

```
Thread-9: 10 * 10 = 100
Thread-4: 5 * 8 = 40
Thread-4: 5 * 9 = 45
Thread-4: 5 * 10 = 50
Thread-5: 6 * 7 = 42
Thread-2: 3 * 4 = 12
Thread-5: 6 * 8 = 48
Thread-0: 1 * 5 = 5
```

....

Note that threads do not appear in the order they were created…

Java Threads: some key points

Every Java program has at least one execution thread

- First execution thread calls `main()`

Each call to **`start()`** method of a `Thread` object creates an actual execution thread

Program ends when all threads (non-daemon threads) finish.

Threads can continue to run even if `main()` returns

Creating a `Thread` object does not start a thread

Calling `run()` doesn't start thread either (need to call start()!)

(Some) Useful Thread attributes and methods

ID: this attribute denotes the unique identifier for each Thread.

```
Thread t =  Thread.currentThread();  // get the current thread
System.out.println("Thread ID" + t.getId()); // prints the current ID.
```

Name: this attribute denotes the name of Thread.

```
t.setName("PP" + 2019);   // can be modified like this
```

Priority: denotes the priority of the thread. Threads can have a priority between 1 and 10:

JVM uses the priority of threads to select the one that uses the CPU at each moment

```
t.setPriority(Thread.MAX_PRIORITY);   // updates the thread's priority
```

Status: denotes the status the thread is in: one of new, runnable, blocked, waiting, time waiting, or terminated (we will discuss the different statuses in more detail later):

```
if (t.getState() == State.TERMINATED)  //check if thread's status is terminated
```

Using Thread states and priorities

```
public static void main(String[] args) {
// Launch 10 threads to do the operation, 5 with the max
// priority, 5 with the min
Thread threads[] = new Thread[10];
Thread.State status[] = new Thread.State[10];

for (int i=0; i<10; i++){
   threads[i]=new Thread(new Calculator(i));
   if ((i%2)==0){
      threads[i].setPriority(Thread.MAX_PRIORITY);
   } else {
      threads[i].setPriority(Thread.MIN_PRIORITY);
   }
   threads[i].setName("Thread "+i);
} ...
```

Cont'd on next slide

```
try (FileWriter file = new FileWriter(".\\data\\log.txt");PrintWriter pw = new PrintWriter(file);){

for (int i=0; i<10; i++){
   pw.println("Main : Status of Thread "+i+" : "+threads[i].getState());
   status[i]=threads[i].getState();
   threads[i].start();
}

boolean finish=false;
while (!finish) {
   for (int i=0; i<10; i++){
     if (threads[i].getState()!=status[i]) {
       writeThreadInfo(pw, threads[i],status[i]);
       status[i]=threads[i].getState();
     }
   }
...
```

Cont'd on next slide

Using Thread states and priorities

```
   ...
   finish=true;
   for (int i=0; i<10; i++){
      finish=finish &&(threads[i].getState()==State.TERMINATED);
   }
}//end while

} catch (IOException e) {
e.printStackTrace();
}
```

Thread priorities: Output

Problems @ Javadoc Declaration Console

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 6, 2014,
Minimum Priority: 1
Normal Priority: 5
Maximun Priority: 10
Thread 0: 0 * 1 = 0
Thread 8: 8 * 1 = 8
Thread 6: 6 * 1 = 6
Thread 4: 4 * 1 = 4
Thread 2: 2 * 1 = 2
Thread 2: 2 * 2 = 4
Thread 2: 2 * 3 = 6
Thread 2: 2 * 4 = 8
Thread 2: 2 * 5 = 10
Thread 2: 2 * 6 = 12
Thread 2: 2 * 7 = 14
Thread 2: 2 * 8 = 16
Thread 2: 2 * 9 = 18
Thread 2: 2 * 10 = 20
Thread 3: 3 * 1 = 3
Thread 4: 4 * 2 = 8
Thread 4: 4 * 3 = 12
Thread 4: 4 * 4 = 16
Thread 4: 4 * 5 = 20
Thread 4: 4 * 6 = 24
Thread 4: 4 * 7 = 28
Thread 4: 4 * 8 = 32
Thread 4: 4 * 9 = 36
Thread 4: 4 * 10 = 40
Thread 5: 5 * 1 = 5
Thread 6: 6 * 2 = 12
Thread 8: 8 * 2 = 16
Thread 0: 0 * 2 = 0
Thread 8: 8 * 3 = 24
Thread 6: 6 * 3 = 18
Thread 6: 6 * 4 = 24
```

log.txt

```
Main : ************************************
Main : Id 15 - Thread 6
Main : Priority: 10
Main : Old State: BLOCKED
Main : New State: BLOCKED
Main : ************************************
Main : Id 15 - Thread 6
Main : Priority: 10
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : ************************************
Main : Id 14 - Thread 5
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : ************************************
Main : Id 14 - Thread 5
Main : Priority: 1
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : ************************************
Main : Id 12 - Thread 3
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : ************************************
Main : Id 12 - Thread 3
Main : Priority: 1
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : ************************************
Main : Id 18 - Thread 9
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : ************************************
Main : Id 18 - Thread 9
```

Thread priorities: Observations

Parallel calculators perform I/O (println)
→ most threads typically blocked

High-priority threads typically finish before low-priority thread

log.txt

```
Main : ************************************
Main : Id 15 - Thread 6
Main : Priority: 10
Main : Old State: BLOCKED
Main : New State: BLOCKED
Main : ************************************
Main : Id 15 - Thread 6
Main : Priority: 10
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : ************************************
Main : Id 14 - Thread 5
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : ************************************
Main : Id 14 - Thread 5
Main : Priority: 1
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : ************************************
Main : Id 12 - Thread 3
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : ************************************
Main : Id 12 - Thread 3
Main : Priority: 1
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : ************************************
Main : Id 18 - Thread 9
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : ************************************
Main : Id 18 - Thread 9
```

Joining Threads

Results, please!

Common scenario:

- Main thread starts (*forks, spawns*) several *worker threads*...
- ... then needs to wait for the worker's results to be available

Previously:

- Busy waiting by *spinning* (looping) until each worker's state is `TERMINATED`
- Boilerplate code
- Inefficient! Main thread spinning uses up CPU time

```
...
finish = false;
While (!finish) {
  ...
  finish = true;
  for (int i=0; i<10; i++){
    finish = finish && (threads[i].getState() == State.TERMINATED);
  }
}
```

Wake me up when work is done

From main thread's perspective:

- Instead of busily waiting for the results (ready? now ready? now?) ...
- ... go to sleep and be woken up once the results are ready

```
...
for (int i=0; i<10; i++) {
  threads[i].join(); // May throw InterruptedException
}
```

Performance trade-off:

- Join (sleep, wakeup) typically incurs context switch overhead
- If worker threads are short-lived, busy waiting may perform better
- Later in the course: `SpinLock`

Question: Is joining `threads[0]`, ..., `threads[9]`optimal?

Exceptions

Exceptions in a single-threaded (i.e. sequential) program terminate the program, if not caught

What if a worker thread throws an exception?

- Exception is (usually) shown on console
- Behaviour of thread.join() is unaffected
- → Main thread may not be aware of an exception inside a worker thread

```
public class Worker extends Thread {
  Data result;
  ...

  @Override
  public void run() {
    ...
    // someObject could be null → NPE
    result = calculate(someObject.getData());
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Worker worker = new Worker(...);
    worker.start();

    worker.join(); // Unaffected
    println(worker.result); // Another NPE
  }
}
```

Setting UncaughtExceptionHandlers

Implementing `UncaughtExceptionHandler` interface allows us to handle unchecked exceptions

Three options:

- Register exception handler with `Thread` object
- Register exception handler with `ThreadGroup` object
- Use `setDefaultUncaughtExceptionHandler()` to register handler for all threads

Handler can then record which threads terminated exceptionally, or restart them, or ...

UncaughtExceptionHandlers: Example

```java
public class ExceptionHandler
    implements UncaughtExceptionHandler {

  public Set<Thread> threads = new HashSet<>();

  @Override
  public void uncaughtException(Thread thread,
                                Throwable throwable) {

    println("An exception has been captured");
    println(thread.getName());
    println(throwable.getMessage());
    ...
    threads.add(thread);
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    ...

    ExceptionHandler handler = new ExceptionHandler();

    thread.setUncaughtExceptionHandler(handler);

    ...

    thread.join();

    if (handler.threads.contains(thread)) {
      // bad
    } else {
      // good
    }
  }
}
```

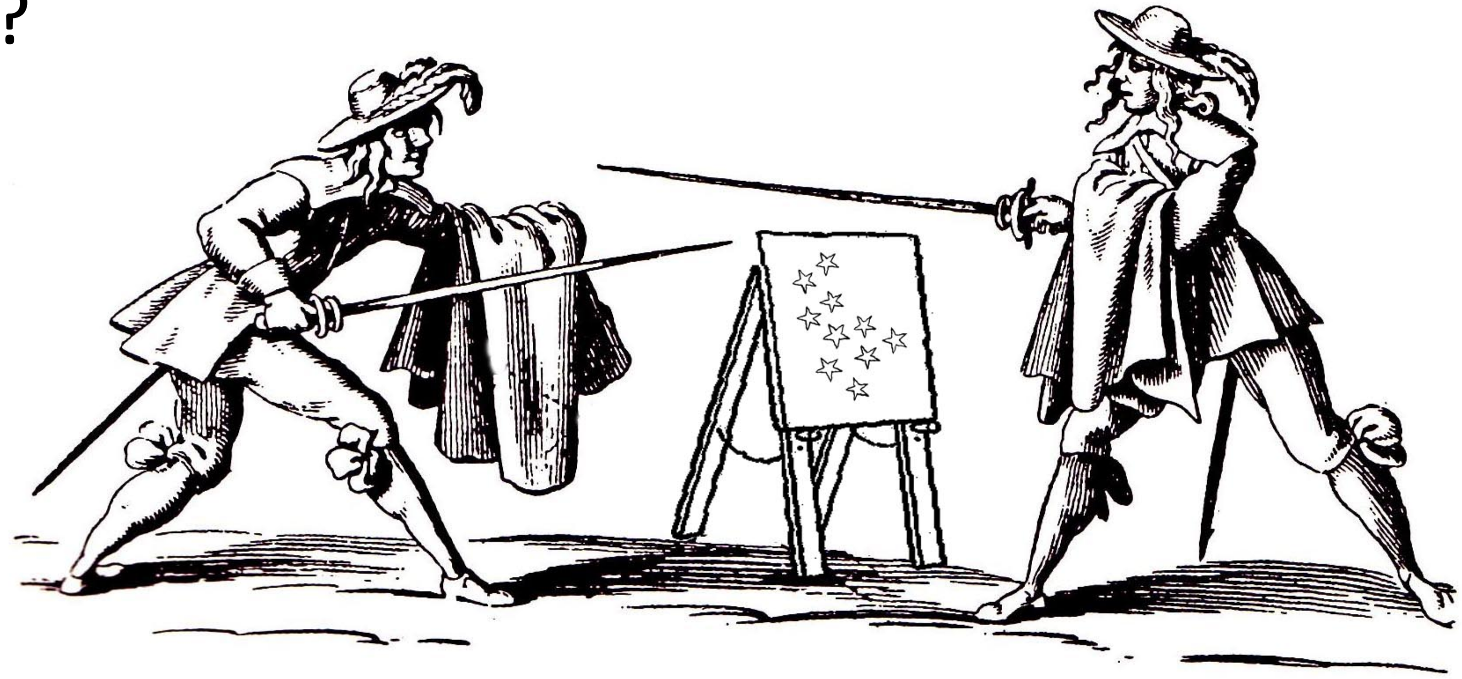
Shared Resources

Battle of the Threads

Two threads "fighting" over console

One writes stars; the other deletes stars (in parallel)

Who will win?



Two thread “fighting” over the console

```
public class BackAndForth {
   public static void main(String args[]){
    System.out.print("********************");
    System.out.flush();
    new Forth().start();
    new Back().start();
}}
```

```
class Back extends Forth{
   @Override
   public void printStars(){
      System.out.print("\b\b\b\b\b\b\b\b\b\b");
   }
}
```

```
public class Forth extends Thread {
   public void run(){
      while(true){
         try {
            sleep((int)(Math.random()*1000));
         } catch (InterruptedException e) { return; }
         printStars();
         System.out.flush();
      }
   }
   public void printStars(){
     System.out.print("*****");
   }
}
```

Synchronized incrementing and decrementing

```
public class Counter implements Runnable {
  public int ticks = -1;

  private Cell cell;
  private int delta;
  private int maxTicks;

  Counter(Cell cell, int delta, int maxTicks) {
    this.cell = cell;
    this.delta = delta;
    this.maxTicks = maxTicks;
  }

  @Override
  public void run() {
    ticks = 0;

    while (ticks < maxTicks) {
      cell.inc(delta);
      ++ticks;
    }
  }
}
```

```
Cell value: -799
Cell value: 667088
Cell value: -281765
Cell value: 147854
...
```

```
public class Main {
  public static void main(String[] args) {
    ...

    Counter up = new Counter(cell, 1, MAX_TICKS);
    Counter down = new Counter(cell, -1, MAX_TICKS);

    Thread upWorker = new Thread(up);
    Thread downWorker = new Thread(down);

    upWorker.start(); downWorker.start();
    upWorker.join();  downWorker.join();

    System.out.printf("Cell value:   %d\n", cell.get());
  }
}
```

```
public class Cell {
  private long value;

  ...

  public void inc(long delta) {
    this.value += delta;
  }
}
```

Updating shared state in parallel

Single statement in LongCell.inc

```
this.value += delta;
```

is executed in several small steps

```
// relevant bytecode
ALOAD 0
DUP
GETFIELD LongCell.value
LLOAD 1
LADD
PUTFIELD LongCell.value
```

Many different interleavings possible

Including bad interleavings in which state data is used

Preview: Threads Safety Hazard

Thread safety

- implies program safety
- typically refers to “nothing bad ever happens”, in any possible interleaving (a safety property)

This is often hard to achieve and requires careful design with parallel execution in mind from the beginning

Preview: Threads Liveness Hazard

Thread safety means: “nothing bad happens”

Liveness means: “eventually something good happens”

Endless loops are an example of liveness hazards in sequential programs

Threads makes liveness hazards more frequent:

- If `ThreadA` holds a resource (e.g. a file handle) exclusively …
- ... then `ThreadB` might be waiting for that resource forever

(What does “holds exclusively” mean in Java? → soon)

Preview: Threads Performance Hazard

Liveness means that progress *will* be made (at some point)

But in (parallel) programming, we're interested in *fast* progress

Multithreaded applications introduce potential performance bottlenecks:

- Frequent context switches: CPU time spend scheduling versus running threads
- Loss of locality (→ next week)
- With synchronization (enforcing mutual exclusion) there is an additional overhead

(What does "synchronization" mean in Java? → soon)

Correctness of Parallel Programs

Examples of safety properties we will encounter in this course include:

- absence of data races
- mutual exclusion
- linearizability
- atomicity
- schedule-deterministic
- absence of deadlock
- custom invariants (e.g., age > 15)

To ensure the parallel program satisfies such properties, we need to correctly synchronize the interaction of parallel threads so to make sure they do not step on each other toes.

synchronized

Shared memory interaction between threads

Two or more threads may read/write the same data (shared objects, global data). Programmer responsible for avoiding bad interleaving by explicit synchronization!

How do we synchronize? Via synchronization primitives.

In Java, all objects have an internal lock, called intrinsic lock or monitor lock

Synchronized operations (see next) lock the object: while locked, no other thread can successfully lock the object

Generally, if you access shared memory, make sure it is done under a lock (Java memory model is complicated!).

(can also use `volatile` keyword, more for experts writing concurrent collections)

Synchronized Methods

```
// synchronized method: locks on "this" object
public synchronized type name(parameters) { ... }

// synchronized static method: locks on the given class
public static synchronized type name(parameters) { ... }
```

A synchronized method grabs the object or class's lock at the start, runs to completion, then releases the lock

Useful for methods whose *entire* bodies are critical sections (recall Alice and Bob’s farm), and thus should not be entered by multiple threads at the same time.

I.e. a synchronized method is a critical section with guaranteed mutual exclusion.

Synchronized incrementing and decrementing

```java
public class Counter implements Runnable {
  public int ticks = -1;

  private Cell cell;
  private int delta;
  private int maxTicks;

  Counter(Cell cell, int delta, int maxTicks) {
    this.cell = cell;
    this.delta = delta;
    this.maxTicks = maxTicks;
  }

  @Override
  public void run() {
    ticks = 0;

    while (ticks < maxTicks) {
      cell.inc(delta);
      ++ticks;
    }
  }
}
```

```
Cell value: 0
Cell value: 0
Cell value: 0
Cell value: 0
...
```

```java
public class Main {
  public static void main(String[] args) {
    ...

    Counter up = new Counter(cell, 1, MAX_TICKS);
    Counter down = new Counter(cell, -1, MAX_TICKS);

    Thread upWorker = new Thread(up);
    Thread downWorker = new Thread(down);

    upWorker.start(); downWorker.start();
    upWorker.join();  downWorker.join();

    System.out.printf("Cell value:   %d\n", cell.get());
  }
}
```

```java
public class Cell {
  private long value;

  ...

  public synchronized void inc(long delta) {
    this.value += delta;
  }
}
```

Synchronized Blocks

```
// synchronized block: uses the given object as a lock
synchronized (object) {
   statement(s); // critical sections
}
```

A synchronized method, e.g.

```
public synchronized void inc(long delta) {
  this.value += delta;
}
```

is syntactic sugar for

```
public void inc(long delta) {
  synchronized (this) {
    this.value += delta;
  }
}
```

Synchronized Blocks

```
// synchronized block: uses the given object as a lock
synchronized (object) {
    statement(s); // critical sections
}
```

Enforces mutual exclusion w.r.t to some object

Every Java object can *act* as a lock for concurrency:

A thread T_1 can ask to run a block of code, synchronized on a given object *O*.

- If no other thread has locked *O*, then T_1 locks the object and proceeds.
- If another thread T_2 has already locked *O*, then T_1 becomes blocked and must wait until T_1 is finished with *O* (that is, unlocks *O*). Then, T_1 is woken up, and can proceed.

Preview: Locks

In Java, <u>all</u> objects have an *internal* lock, called intrinsic lock or monitor lock, which are used to implement `synchronized`

Java also offers external locks (e.g. in package `java.util.concurrent.locks`)

- Less easy to use
- But support more sophisticated locking idioms, e.g. for reader-writer scenarios

Locks Are Recursive (Reentrant)

A thread can request to lock an object it has already locked

```
public class Foo {
     public void synchronized f() { … }
     public void synchronized g() { … f(); … }
}

Foo foo = new Foo();
synchronized(foo) { … synchronized(foo) { … } … }
```

Examples: Synchronization granularity

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

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

The advantage of not synchronizing the entire method is efficiency but need to be careful with correctness

Examples: Synchronization with different locks

```
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++;
      }
   }
}
```

The locks are disjoint – allows for more concurrency.

Examples: Synchronization with static methods

```
public class Screen {
  private static Screen theScreen;

  private Screen(){…}

  public static synchronized getScreen() {
    if (theScreen == null) {
      theScreen = new Screen();
    }

    return theScreen;
  }
}
```

Which object does synchronized lock here?
What if `Screen` instances call `getScreen()`?

Interleavings: Examples

Suppose we have 2 threads, T1 and T2, both incrementing a shared counter. If we use synchronized (say on 'this' object), we will get the desired result of 2 by the time both threads have finished executing their code below.

```
                    c = 0

T1:                               T2:

synchronized (this)               synchronized (this)
{                                 {
 1:   t1 = c;                       4: t2 = c;
 2:   t1++                          5: t2++
 3:   c = t1;                       6: c = t2;
}                                 }
```

For convenience, we use labels 1-6 to refer to the instructions. The possible interleavings / executions of this program are that either T1 runs before T2 or vice versa. So we will have:

Interleaving 1: 123456　　　　Interleaving 2: 456123

Interleavings: Another example

Suppose the programmer forgot to use synchronized in thread T2. What is an example of an undesirable interleaving that we can see?

```
                 c = 0

T1:                         T2:

synchronized (this)
{
 1:   t1 = c;                  4: t2 = c;
 2:   t1++                     5: t2++
 3:   c = t1;                  6: c = t2;
}
```

A possibly bad interleaving is: 4 1 2 3 5 6

This interleaving will result in the counter ‘c’ being set to 1 at the end of the interleaving.

Interleavings: Another example

Suppose the programmer now uses synchronized in thread T2 but not on ‘this’, but say another object ‘p’. Does this prevent the bad interleaving we just saw?

```
                  c = 0

T1:                           T2:

synchronized (this)           synchronized (p)
{                             {
 1:  t1 = c;                    4: t2 = c;
 2:  t1++                       5: t2++
 3:  c = t1;                    6: c = t2;
}                             }
```

No, the bad interleaving: 4 1 2 3 5 6   can still happen because ‘p’ and ‘this’ are different objects.

Synchronized and Exceptions

What happens if in the middle of a synchronized block, an exception triggers?

```
public void foo() {
 synchronized (this) {
   longComputation();  // say this takes a while…
   divisionbyZero();   // this throws an exception..
   someOtherCode();    // something else
  }
}
```

In this case, after `longComputation()` completes, an exception is thrown. What happens then is as follows. First, the synchronized on the 'this' object will be released -- as if the synchronized scope ends right at the point where the exception is thrown. Second, the exception is caught, then the exception handler is executed. If there is no exception handler, as in our example, then the exception is propagated back down to the caller of `foo()` as usual.

Note that the code `someOtherCode()` will NOT be executed in this case. Also note that any side effects of `longComputation()` are **NOT reverted**, they do take effect, even if exceptions are thrown.

Synchronized and Exceptions (optional)

(not exam relevent)

If you want to know more on exactly how synchronized/exceptions interact in the bytecode, you can compile the following code:

```
class test {
  public void foo() {
    int pp;
    synchronized (this) { pp = 1; }
  }
}
```

Then you can call the command: **javap –c test (our old friend)**

This will show you the 14 bytecodes for this method foo(). You can then see exactly how synchronized is handled (e.g., via monitorenter/monitorexit) and see the 2 exception tables generated in the case of exceptions inside synchronized. This is not something that will be examined, it is for your own information when you need to debug the code sometimes.

You should know what happens with synchronized/exceptions though as outlined on the previous slide.

How is synchronized actually implemented? (preview)

Recall the native layer we briefly discussed in Lecture 2.

Internally, the JVM implements synchronized by using native, operating system primitives (and low level architecture instructions, say Intel's x86 e.g. compare-and-swap, or IBM Power's LL/SC). This means the implementation of synchronized will look different on different OS/architecture combinations.

If you remember our informal mutual exclusion, we essentially provided an implementation of **synchronized** that works for 2 threads (Alice and Bob) that relies only shared reads and writes to 3 variables (flag1, flag2, and turn)

In later lectures we will see the instructions that are used to implement synchronized.

Few Historic Notes: Objects/Monitors

(not exam relevent)

1960's - Simula 67 introduces the concept of objects - by Ole-Johan Dahl and Kristen Nygaard

1971 – Ideas around monitor concept discussed by Per Brinch Hansen/Tony Hoare/Edsger Dijkstra

1972 - Proposes first monitor notation, influenced by Simula 67's classes – by Per Brinch Hansen, later refined by Tony Hoare

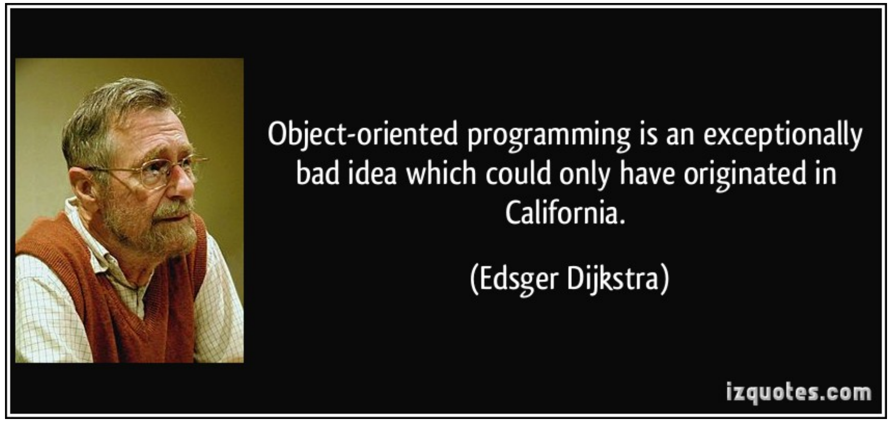
1970's - Smalltalk introduces object oriented programming (OOP) – by Alan Kay and Xerox PARC

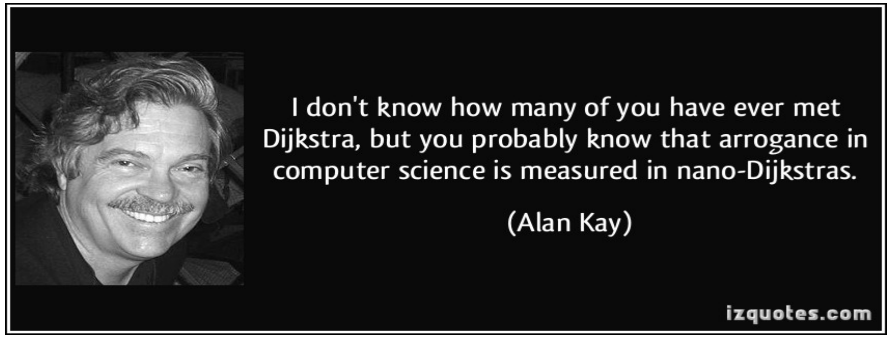
1985 – Eiffel: OOP + Design-by-contract – by Bertrand Meyer

1985 – C++ : by Bjarne Stroustrup

1995 – Java: by James Gosling and Sun Microsystems … also borrowed the concept of monitors.

Many more follow: JavaScript, Scala, Kotlin, etc…





Few Historic Notes: Memory Models

(not exam relevent)

Java semantics =
- \+ statement semantics (under single-threaded execution)
- \+ memory model (how threads interact through memory)

Java’s 1995 memory model is seen as
- first serious attempt for a popular language (C/C++ simply had none)
- too vague
  - unclear if code was ever correct
  - reduced potential for compile/runtime optimizations
  - unexpected behavior in practice, e.g. final fields changing their values

2004: New, improved Java memory model takes effect

2011: C11/C++11 memory model takes effect
- more complex (and powerful) than Java’s
- different levels of how “weak” memory may be: fewer guarantees for developers means more optimization potential for compilers and hardware

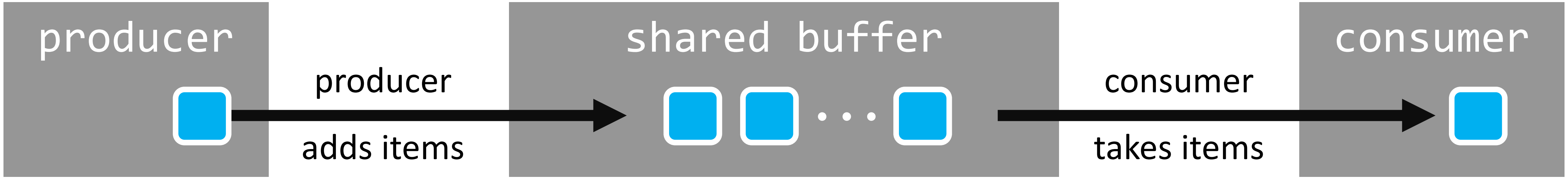
C/C++’s memory model is currently under revision: soundness and performance issues in specific situations

Wait, Notify, NotifyAll

Producer-Consumer

Producer and consumer run indefinitely

Producer puts items into a shared buffer, consumer takes them out



For simplicity, buffer is unbounded (has no capacity limit); producing is always possible

But consumption only possible if buffer isn't empty

Producer-Consumer: v1

```java
public class UnboundedBuffer {
  // Internal implementation could be a standard collection,
  // or a manually-maintained array or linked-list

  public boolean isEmpty() { ... }
  public void add(long value) { ... }
  public long remove() { ... }
}
```

```java
public class Consumer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    while (true) {
      while (buffer.isEmpty()); // Spin until item available
      performLongRunningComputation(buffer.remove());
    }
  }
}
```

Can you see any problems?

```java
public static void main(String[] args) {
  UnboundedBuffer buffer = new UnboundedBuffer();

  Producer producer = new Producer(buffer);
  producer.start();

  Consumer[] consumers = new Consumer[3];

  for (int i = 0; i < consumers.length; ++i) {
    consumers[i] = new Consumer(i, buffer);
    consumers[i].start();
  }
}
```

```java
public class Producer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      buffer.add(prime);
    }
  }
}
```

Producer-Consumer: v1 – Bad Interleavings

```
public class Consumer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    while (true) {
      while (buffer.isEmpty()); // Spin until item available
      performLongRunningComputation(buffer.remove());
    }
  }
}
```

Problem: buffer could be emptied between isEmpty() and remove()

Problem: buffer operations (add(), remove()) might be interleaved on bytecode level → buffer's internal state might get corrupted

```
public class Producer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      buffer.add(prime);
    }
  }
}
```

Producer-Consumer: v2

```
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty());
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

```
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
      }
    }
  }
}
```

Added `synchronize(buffer)` blocks around operations on buffer to enforce *mutual exclusion* in the *critical sections*

Can you see any new problems?

Producer-Consumer: v2 – Deadlock

```
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty());
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

```
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
      }
    }
  }
}
```

Problem:

1. Consumer locks buffer (`synchronize (buffer)`)
2. Consumer spins on `isEmpty()`, i.e. waits for producer to add item
3. Producer waits for lock to become available (`synchronize (buffer)`)
4. → Deadlock! Consumer and producer wait for each other; no progress

Producer-Consumer: v3

```
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty())
          buffer.wait();
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

`buffer.wait()`:

1. Consumer thread goes to sleep (status NOT RUNNABLE) …
2. … and gives up buffer's lock

```
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
        buffer.notifyAll();
      }
    }
  }
}
```

`buffer.notifyAll()`:

1. All threads waiting for buffer's lock are woken up (status RUNNABLE)

Beyond synchronization: Wait, Notify, NotifyAll

```
public class Object {
      ...
      public final native void notify();
      public final native void notifyAll();

      public final native void wait(long timeout) throws InterruptedException;
      public final void wait() throws InterruptedException { wait(0); }
      public final void wait(long timeout, int nanos)
         throws InterruptedException { ... }
}
```

wait() releases object lock, thread waits on internal queue

notify() wakes the highest-priority thread closest to front of object's internal queue

notifyAll() wakes up all waiting threads

- Threads non-deterministically compete for access to object
- May not be fair (low-priority threads may never get access)

May only be called when object is locked (e.g. inside `synchronize`)

Why do we need loop and synchronized when we use wait/notify?

CASE I: Lets consider the case where we do NOT have a loop (we use an ‘if’ instead) and do NOT have synchronized: see code below.

```
public void consume() {
    if (!consumable()) {
        wait();
     }    // release lock and wait for resource
    …   // have exclusive access to resource, can consume
}

public void produce() {
    … // do something to make consumable() return true
    notifyAll();    // tell waiting threads to try consuming
    //  can also call notify() to notify one thread at a time
}
```

For a moment, let’s assume that bad interleavings *on the bytecode level* aren’t already a problem.

A remaining problem is that we can have a situation where the consumer checks if it can proceed and consumable() returns false. Right before calling wait(), produce() now completes successfully, and consume resumes and goes to wait(). If produce never runs again, consume will be blocked forever even though there is something to consume (i.e. consumable() would return true).

Note that in Java, if wait() is called without synchronized on that object, an exception will be thrown. However, even if it was not thrown somehow, the above bad scenario can happen.

Why do we need loop and synchronized when we use wait/notify?

CASE II: Let us now consider the case where we have synchronized but still no loop, we have an `if`.

```
public synchronized void consume() {
    if (!consumable()) {
        wait();
     }    // release lock and wait for resource
    …   // have exclusive access to resource, can consume
}

public synchronized void produce() {
    … // do something to make consumable() return true
    notifyAll();    // tell waiting threads to try consuming
    //  can also call notify() to notify one thread at a time
}
```

The problem here is that the consumer can return from a wait() call for reasons other than being notified (e.g. due to a thread interrupt), or because different consumer's have different conditions.

If we do not recheck the consumable() condition upon return from wait, we do not know why the thread returned from wait().

This is the reason why it is *strongly recommended* to use a while loop around the condition, instead of just an if statement.

Nested Lockout Problem

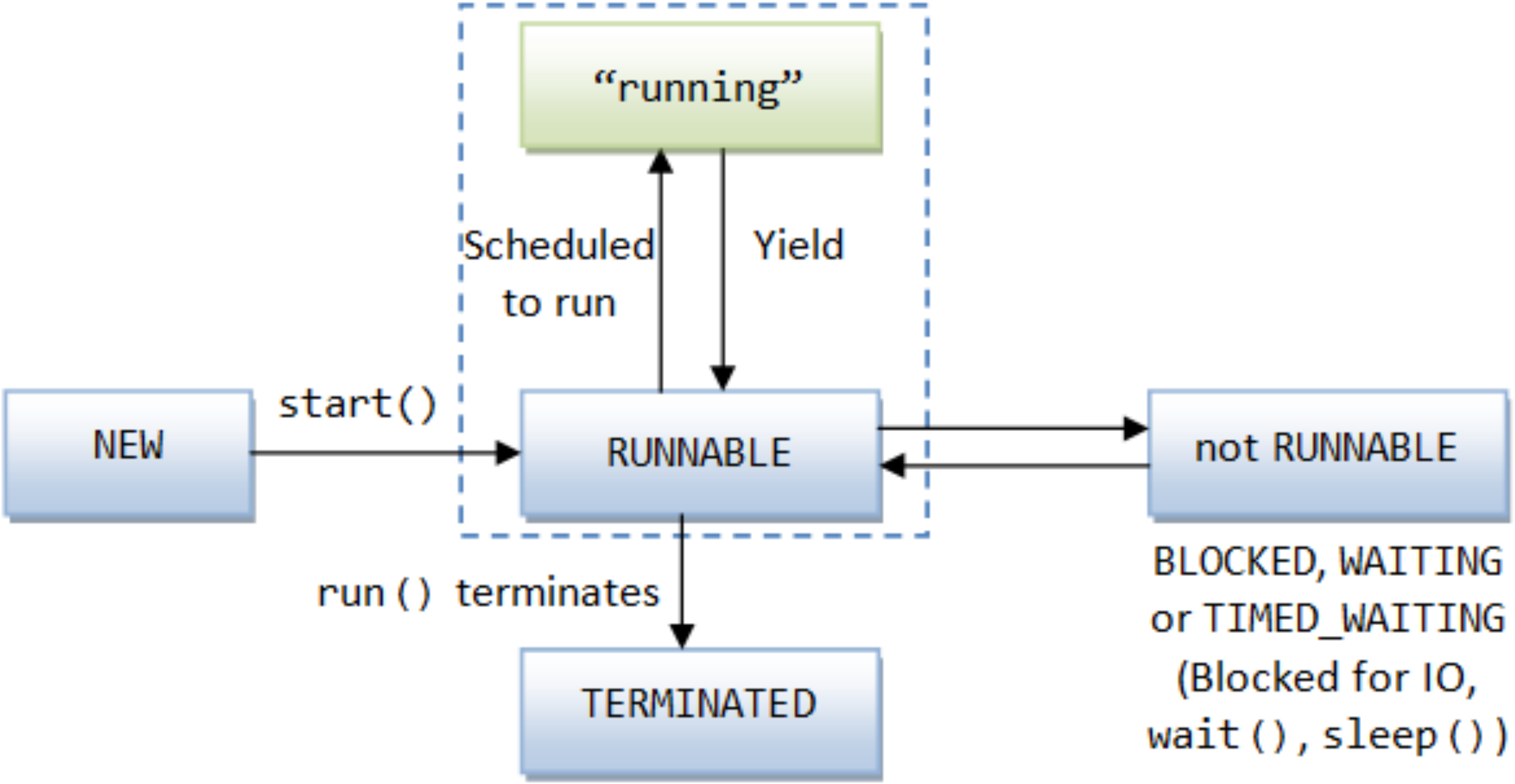
Potentially blocking code within a synchronized method can lead to deadlock

```
class Stack {
    LinkedList list = new LinkedList();
    public synchronized void push(Object x) {
      synchronized(list) {
         list.addLast( x ); notify();
    } }
    public synchronized Object pop() {
      synchronized(list) {
         if( list.size() <= 0 ) wait();
         return list.removeLast();
    } }
}
```

Releases lock on `this` object but not lock on `list`; a push from another thread will deadlock

Preventing the problem: No blocking code/calls in synchronized methods, or provide some non-synchronized method of the blocking object. No simple solution that works for all programming situations.

Thread state model in Java (repetition)



http://pervasive2.morselli.unimo.it/~nicola/courses/IngegneriaDelSoftware/java/J5e_multithreading.html

Thread States: Summary

Thread is created when an object derived from the Thread class is created. At this point, the thread is not executable, it is in a new state.

Once the `start` method is called, the thread becomes eligible for execution by the scheduler.

If the thread calls the `wait` method in an Object, or calls the `join` method in another thread object, the thread becomes not runnable and no longer eligible for execution.

It becomes executable as a result of an associated `notify` method being called by another thread, or if the thread with which it has requested a join, becomes terminated.

A thread enters the terminated state, either as a result of the run method exiting (normally, or as a result of an unhandled exception) or because its destroy method has been called.

In the latter case, the thread is abruptly moved to the terminated state and does not have the opportunity to execute any finally clauses associated with its execution; it may leave other objects locked.