Parallel Programming

Introduction to Threads and Synchronization

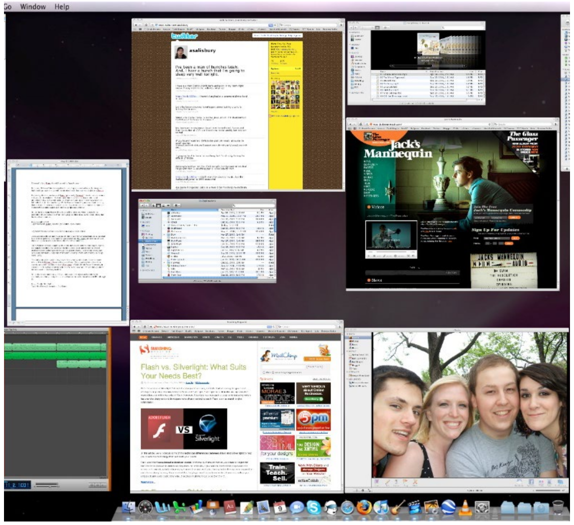
Multitasking/Multiprocessing

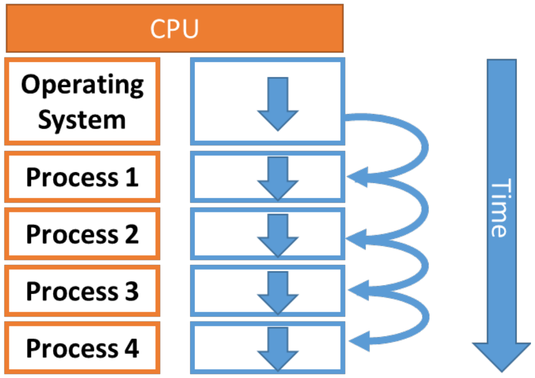
Multitasking

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

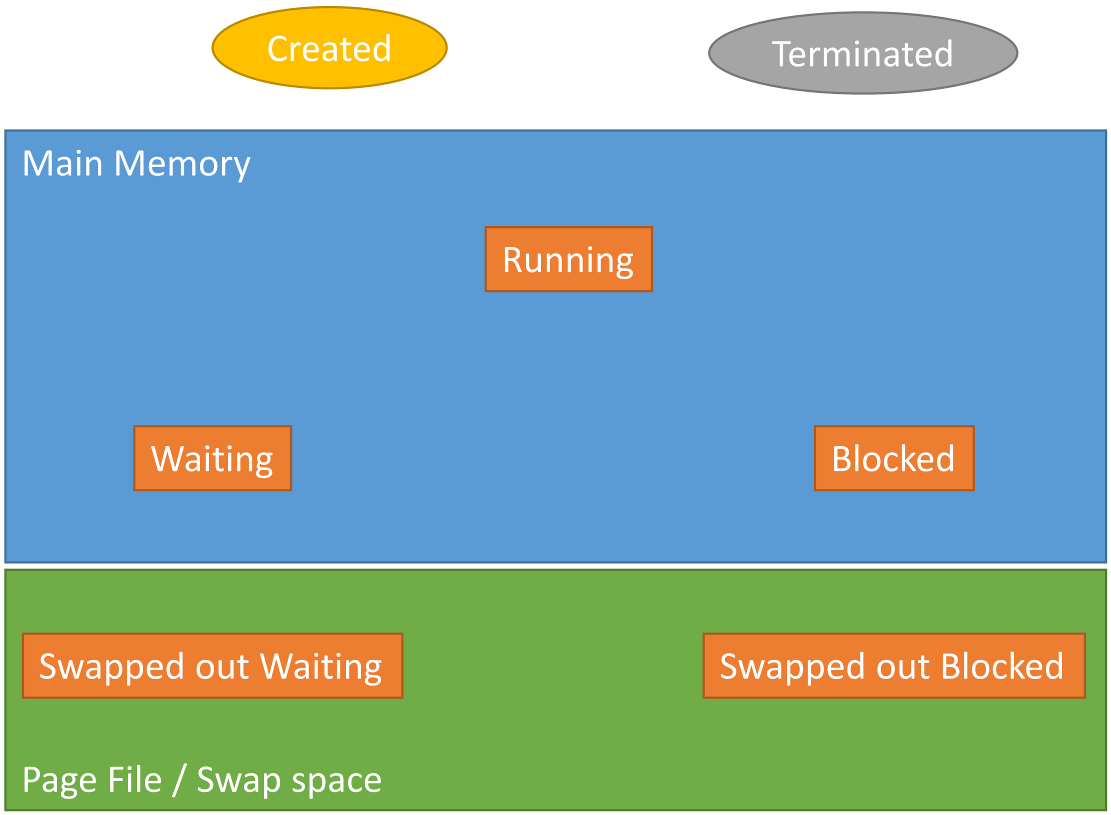
Each running instances of a program (e.g., multiple browser windows) is a separate process

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

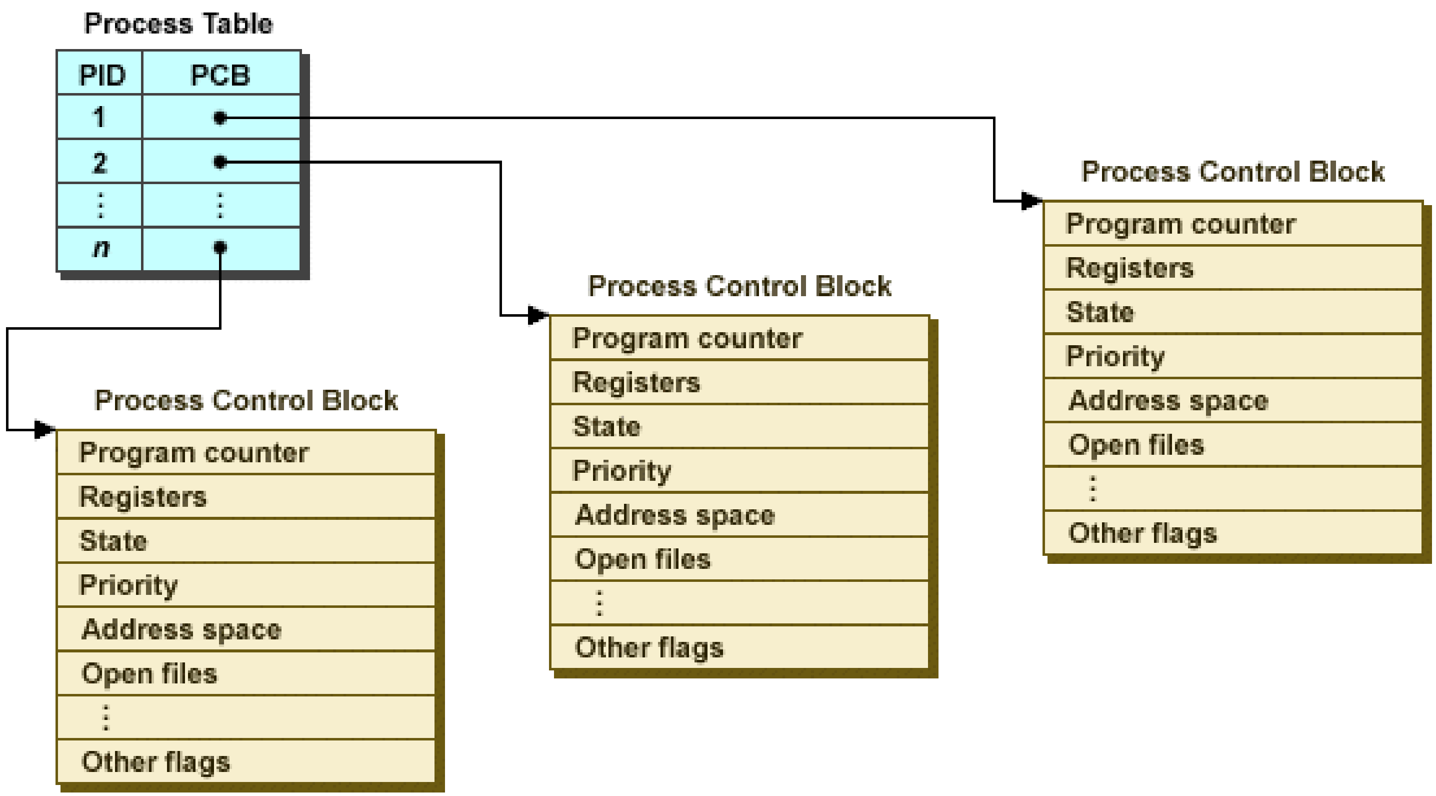
Processes need resources

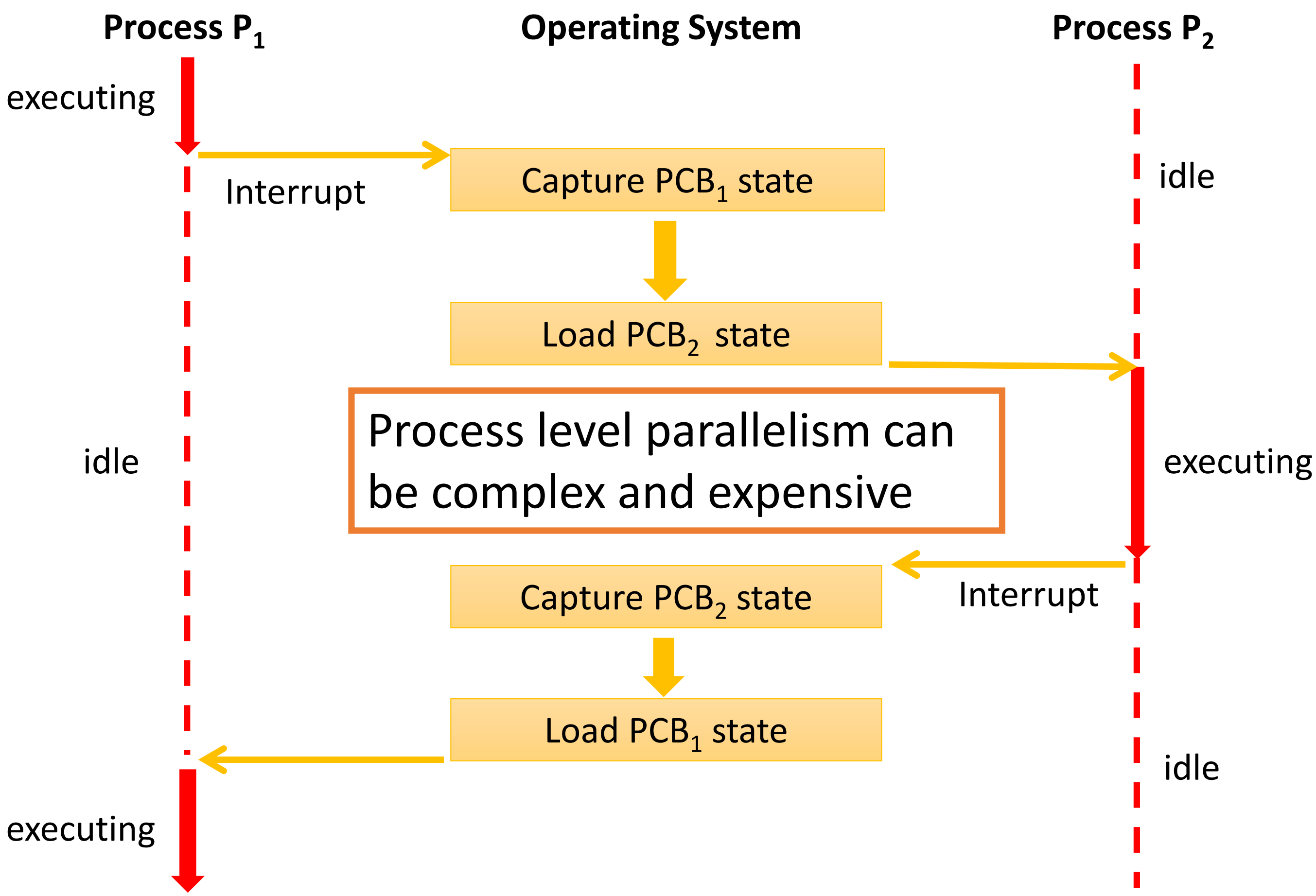
- CPU time, Memory, etc.

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 level parallelism can be complex and expensive

Multithreading

Threads

Threads (of control) are
- independent sequences of execution
- running in the same OS process

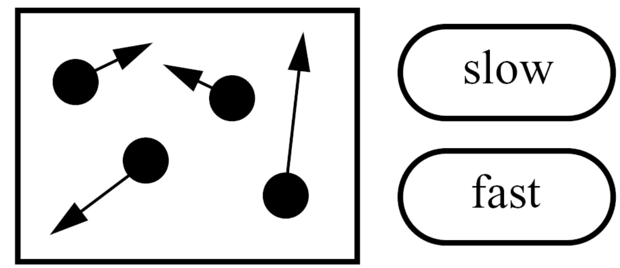
Multiple threads share the same address space.
- Threads are not shielded from each other
- Threads share resources and can communicate more easily

More vulnerable for programming mistakes

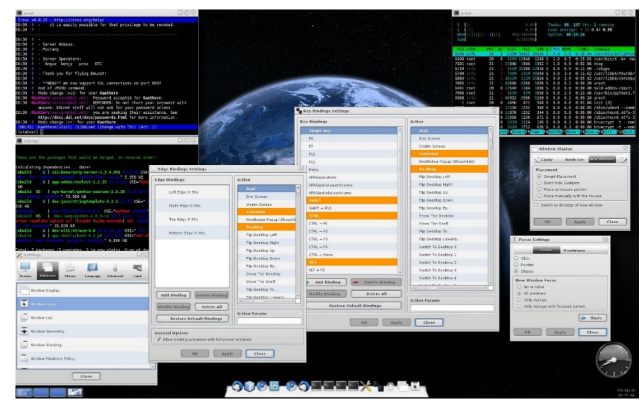
Context switching between threads is efficient
- No change of address space
- No automatic scheduling
- No saving / (re-)loading of PCB (OS process) state

Usage of Multithreading

Reactive systems – constantly monitoring



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

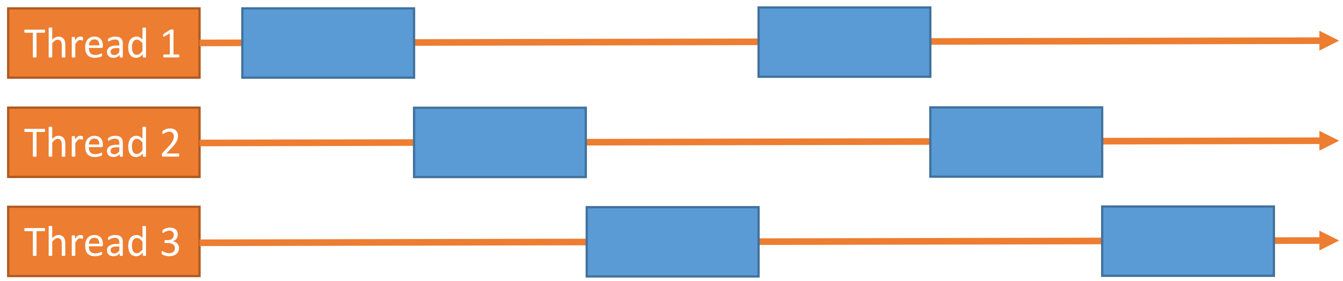


Server can handle multiple clients simultaneously

Take advantage of multiple CPUs/cores

Multithreading: 1 vs. many CPUs

Multiple threads sharing a single CPU



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)

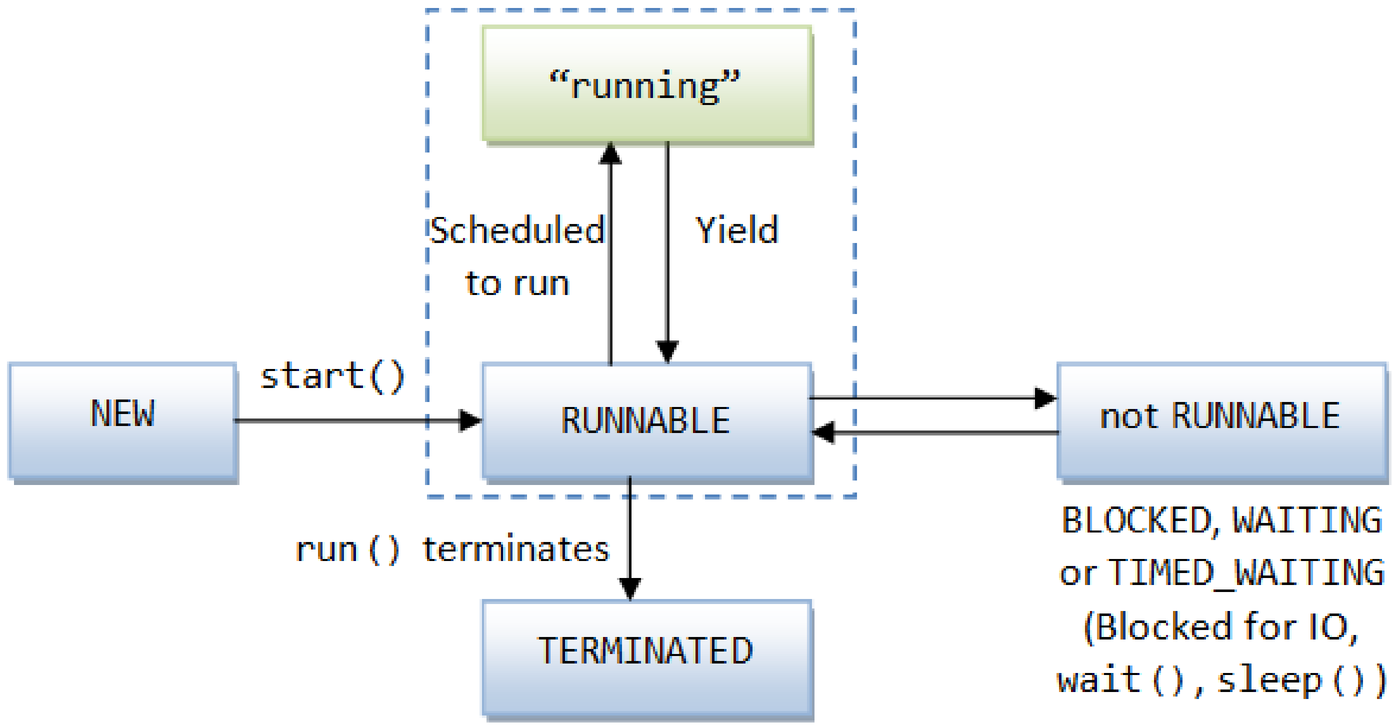
Implement `java.lang.Runnable`

- 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)

```
// 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/nati
ve/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

```
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();
}}
```

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
```

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

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

Previously:

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

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?

Shared Resources

`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 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.