EDA095
Processes and Threads

Pierre Nugues

Lund University
http://cs.lth.se/pierre_nugues/

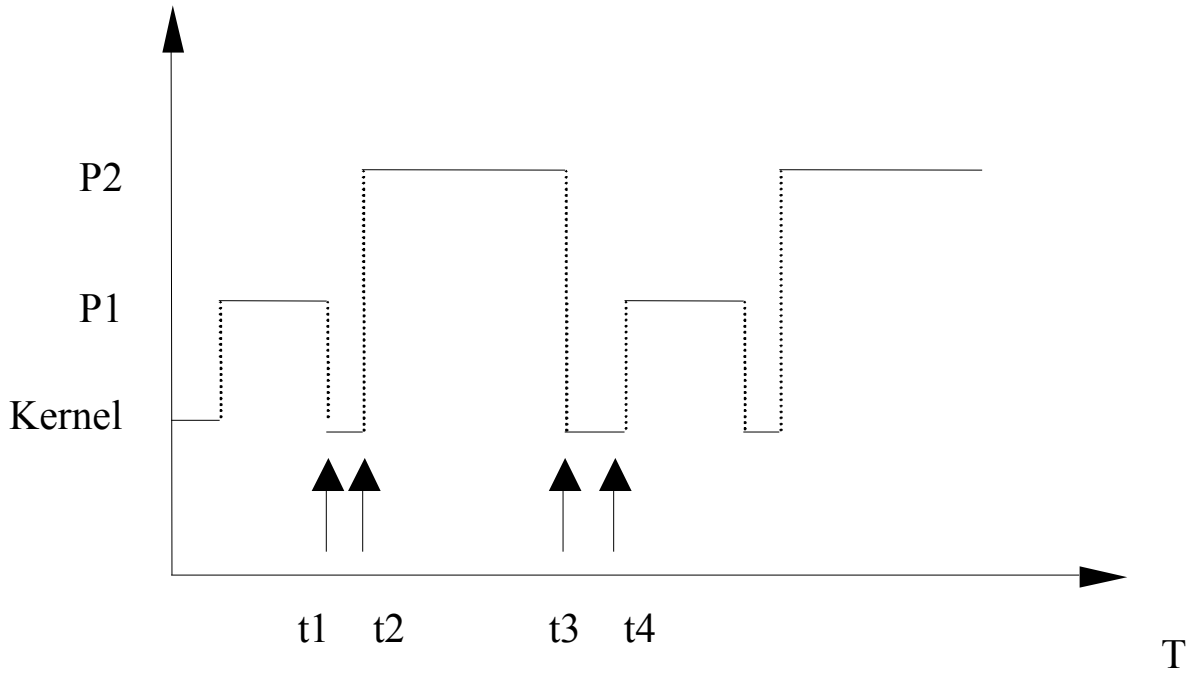
March 31, 2016

Time-Sharing Operating Systems

Processes are programs in execution

Most operating systems can run multiple processes in parallel

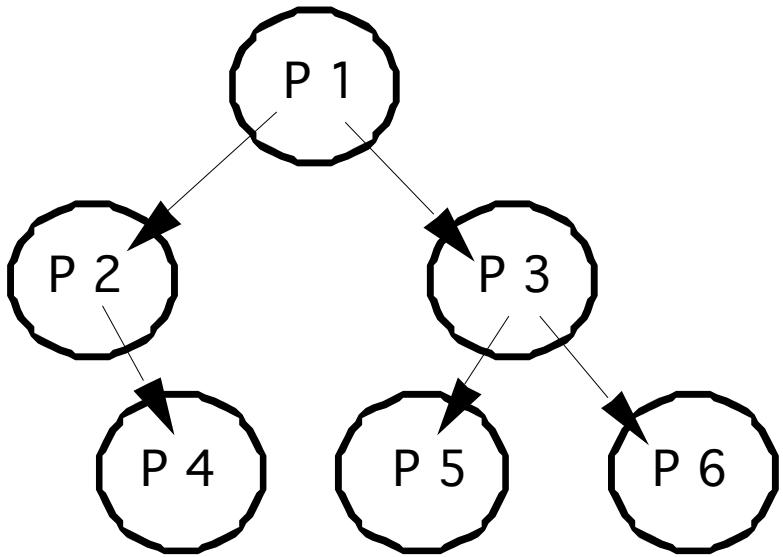
OSes allocate a small quantum of CPU time to each process



Task switching is very fast and gives the impression of simultaneous processing

Process Creation

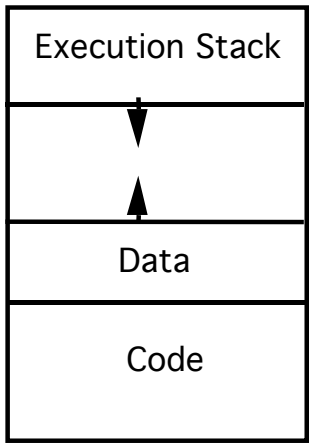
On Unix, a command interpreter – a shell – launches the user processes.
A process can then launch other processes
The creating process is the "parent" and the created processes are the "children"



An initial process started at boot time is the ancestor of all the processes: system processes and user processes

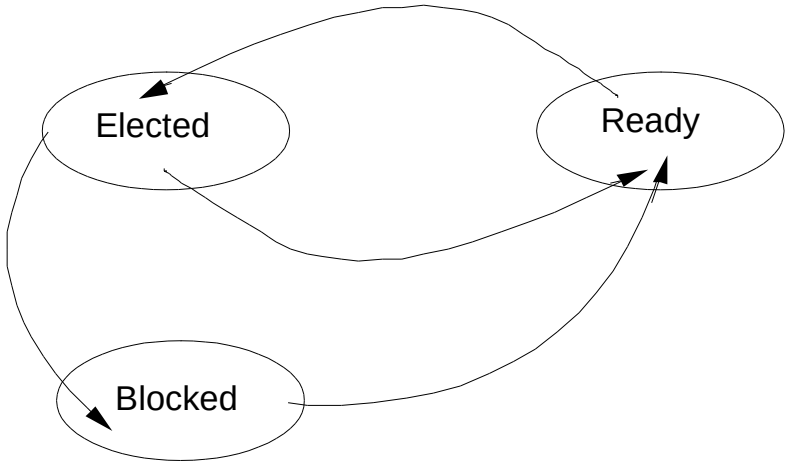
The Content of a Process

Running processes are located in the computer memory.
They contain the program code – resulting from the compiling
A data area that stores the dynamic data is allocated by the program at run-time using `new`
When the program calls functions or methods, a stack stores their parameters

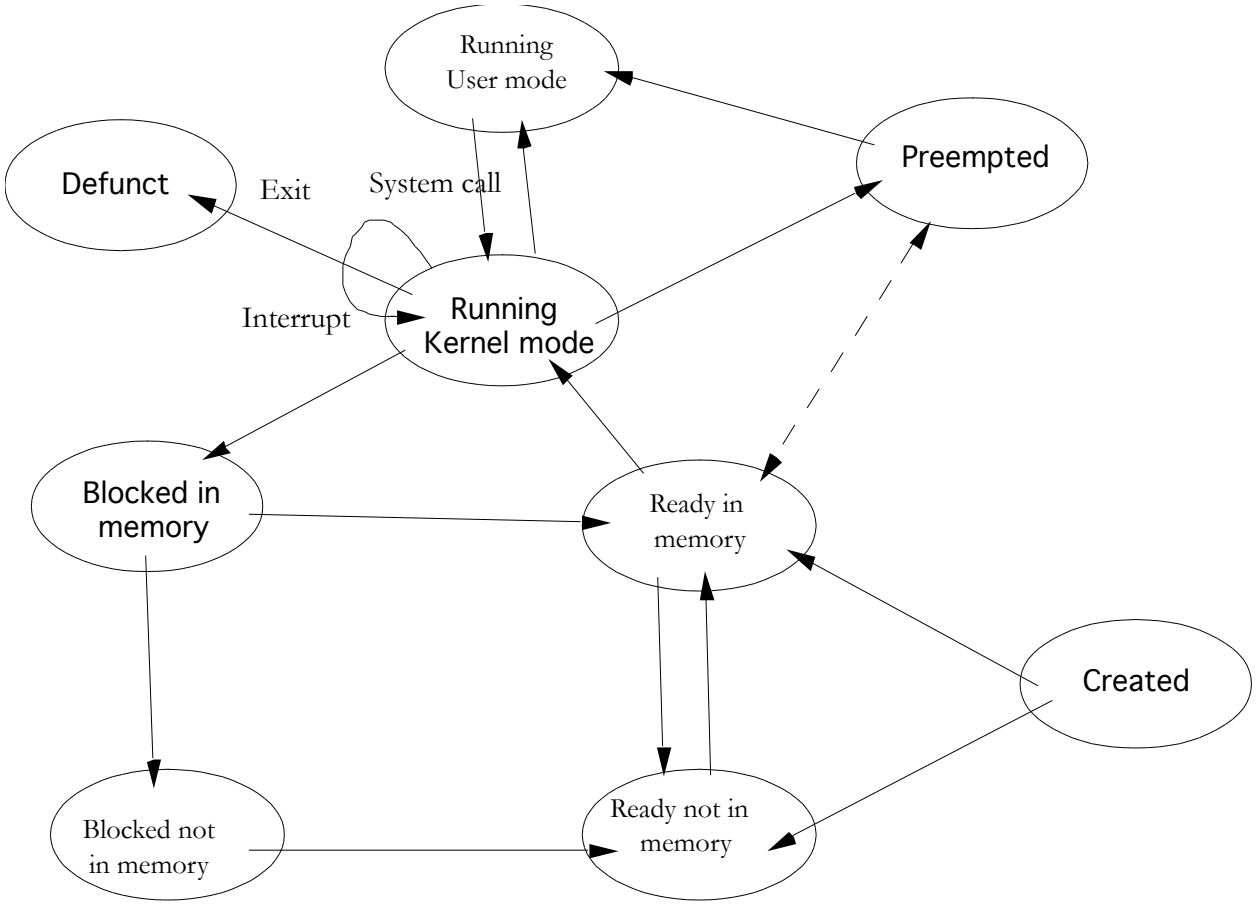


The Process States

The model of process execution is a finite-state machine
Processes waiting for the CPU are “ready”
The scheduler “elects” one process and runs it.
On an I/O, the elected process is moved to a “blocked” state until the I/O is completed.

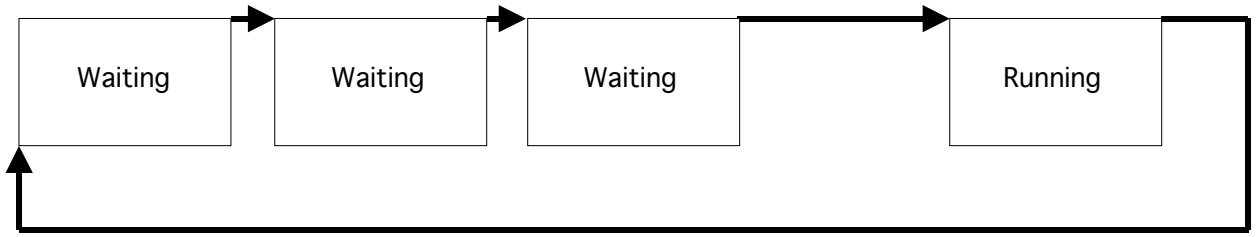


The Unix states are slightly more complex



The Scheduler

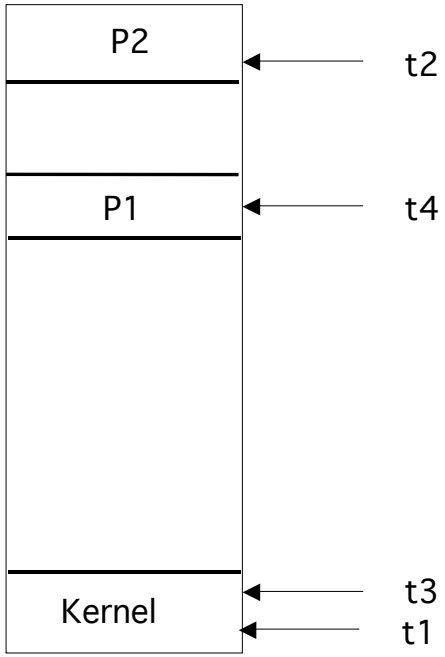
The scheduler selects one process from the queue of ready processes.
A scheduler runs every 10 ms or so and chooses a new process



There are many scheduling algorithms available.
Scheduling must be very fast and use relatively simple algorithms:
first-come, first-served, round robin, priority.

The Process Control Block

The operating system uses additional data to run a process: the process context or process control block. It contains the process state, program counter, CPU registers, etc.



When switching tasks, the OS saves the current context and restores the context of the process that it will execute.

The Operating System Operations on Processes

The operating system creates, schedules, and terminates the processes.
Processes must sometimes cooperate and share data.
The operating system offers communication means between processes, interprocess communications (IPC) and naming facilities:

- Pipes
- Shared memory
- Messages

The operating system also offers means to coordinate and synchronize processes: semaphores on Unix

Threads

Traditional processes are sequential: They have one execution path—one thread of control

Concurrent processes have multiple threads of control i.e. processes within a process.

Imagine a word processor application. The process must read the keystrokes, display the text, check spelling, and so on, at the same time

Difficult to manage with a single thread

The idea is to allocate one thread to each task.

Benefits

Easier to implement parallelism within the application
Input/output does not block the process, only one thread
More responsive programs using high-priority threads to manage user interaction.
Threads are more economical: creating a process is 10 to 100 longer than creating a thread.
But:
Threads are more difficult to coordinate and prone to nasty bugs.

Java Threads

Threads are a feature of Java: a Java program and virtual machine can run multiple threads.
Java provides constructs to create, manage, synchronize, and terminate threads through:

- the `Thread` class `http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html` or
- the `Runnable` interface `http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html`.

Threads can communicate using shared objects, pipes, or through messages (sockets)

Java Threads

Creating Threads: The `Thread` class

Threads can be created using the `Thread` class in two steps.

1. Create a new class:
   - Derive a new class from `Thread` using `extends`
   - Override the `run()` method
2. Create and run a thread object:
   - Create a thread object using `new`.
   - Start it using the `start()` method

A First Program using the Thread Class

```
public class MyThread extends Thread {
    MyThread() {}
    public void run() {
        System.out.println("My first thread");
    }
    public static void main(String args[]) {
        MyThread firstThread = new MyThread();
        firstThread.start();
    }
}
```

Creating Threads: The `Runnable` Interface

The `Runnable` interface is another option to create threads.

1. Add the `Runnable` properties to a class:
   - Implement the `Runnable` interface using `implements`
   - Add a `run()` method
2. Create and run a thread object:
   - Create a `Runnable` object using `new`.
   - Create a thread that takes the runnable object as an argument
   - Start the thread using the `start()` method

A First Program using the Runnable Interface

```
public class MyRunnable implements Runnable {
    MyRunnable() {}
    public void run() {
        System.out.println("My second thread");
    }
    public static void main(String args[]) {
        MyRunnable firstRunnable = new MyRunnable();
        Thread myThread = new Thread(firstRunnable);
        myThread.start();
    }
}
```

Passing Data to a Thread

You can pass data at creation time. `Thread(String name)` is a useful constructor.

```
public class MyThread2 extends Thread {
    private int myInt;
    MyThread2(String name, int myInt) {
        super(name);
        this.myInt = myInt;
    }
    public void run() {
        for (int i = 0; i < myInt; i++) {
            System.out.println(getName() + ": " + i);
        }
        System.out.println(getName() + " terminated");
    }
```

Passing Data to a Thread, Continued

```
    public static void main(String args[]) {
        MyThread2 secondThread = new MyThread2("Thread#2", 123);
        secondThread.start();
    }
}
```

Returning Data from a Thread

Does this work?

```
class MyThread {

  result;

  MyThread() {}

  getResult() {
    return result;
  }

  run() {

    result = produceResult();
  }
}
```

```
class Main {



  thread.start();
  thread.getResult();
}
```

Callbacks

```
class MyThread {

  result;

  MyThread() {}

  getResult() {
    return result;
  }

  run() {

    result = produceResult();
    Main.callback(result);
  }
}
```

```
class Main {
  static callback(result) {
    useResult(result);
  }

  main() {

    thread.start();
  }
}
```

(Launcher2.java and Thread2.java)

Working with Multiple Threads

The Java Virtual Machine manages the scheduling.

```
public static void main(String args[]) {
    int loopCount;
    loopCount = Integer.parseInt(args[0]);
    MyThread3 thirdThread = new MyThread3("Thread3", loopCount);
    MyThread3 fourthThread = new MyThread3("Thread4", loopCount);
    thirdThread.start();
    fourthThread.start();
}
```

The Thread API

The thread API consists of 8 constructors and ~40 methods, some of them deprecated.

Constructors
`Thread()`
`Thread(Runnable target)`
`Thread(Runnable target, String name)`
`Thread(String name)`
`Thread(ThreadGroup group, Runnable target)`
`Thread(ThreadGroup group, Runnable target, String name)`
`Thread(ThreadGroup group, Runnable target, String name, long stackSize)`
`Thread(ThreadGroup group, String name)`

Thread Methods

```
static int activeCount()
void checkAccess()
static Thread currentThread()
static void dumpStack()
static int enumerate(Thread[] tarray)
ClassLoader getContextClassLoader()
String getName()
int getPriority()
ThreadGroup getThreadGroup()
static boolean holdsLock(Object obj)
void interrupt()
static boolean interrupted()
boolean isAlive()
boolean isDaemon()
boolean isInterrupted()
```

```
void join()
void join(long millis)
void join(long millis, int nanos)
void run()
void setContextClassLoader(ClassLoader cl)
void setDaemon(boolean on)
void setName(String name)
void setPriority(int newPriority)
static void sleep(long millis)
static void sleep(long millis, int nanos)
void start()
String toString()
static void yield()
public long getId()
public Thread.State getState()
```

Deprecated Methods

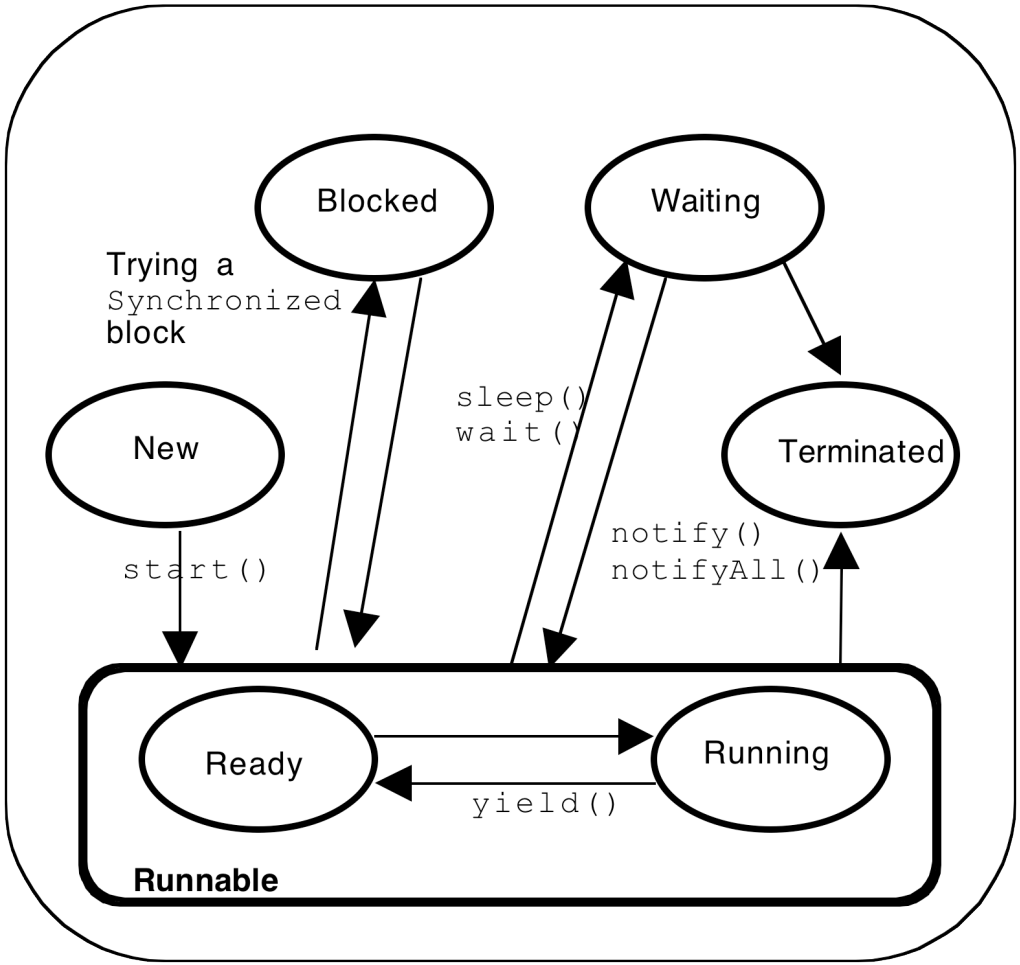
```
int countStackFrames()   void stop()
void resume()            void stop(Throwable obj)
void destroy()           void suspend()
```

A frequent question is: *Why can't I stop a thread?*

Read:

`http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`

The Thread States



From Java 1.5.0: `Thread.State` enables to know the state.
http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.State.html

Java Threads Scheduling

The JVM schedules threads using priorities.
Priorities are adjustable and range from 1 to 10:

- Thread.MAX_PRIORITY = 10
- Thread.NORM_PRIORITY = 5
- Thread.MIN_PRIORITY = 1

The Java specifications do not describe the scheduling algorithm. They are left to the implementer.

Scheduling Algorithms

Scheduling can be preemptive or cooperative:

- A cooperative scheduler selects the highest priority thread and runs it until it is completed unless the thread carries out an I/O or yields control using `yield()`
- A preemptive scheduler allocates time quanta to threads so that they all can run. High priority tasks should have more time than lower priority ones.

Be aware of the implementation differences that are not documented.
On older Java implementations, a thread cannot be taken away from the processor if it does not complete an I/O operation

Adjusting Priorities

```
public static void main(String args[]) {
    int firstPriority, secondPriority;
    int loopCount;
    firstPriority = Integer.parseInt(args[0]);
    secondPriority = Integer.parseInt(args[1]);
    loopCount = Integer.parseInt(args[2]);
    MyThread4 fifthThread = new MyThread4("Thread#5", loopCount);
    MyThread4 sixthThread = new MyThread4("Thread#6", loopCount);
    fifthThread.setPriority(firstPriority);
    sixthThread.setPriority(secondPriority);
    fifthThread.start();
    sixthThread.start();
}
```

Thread Implementation

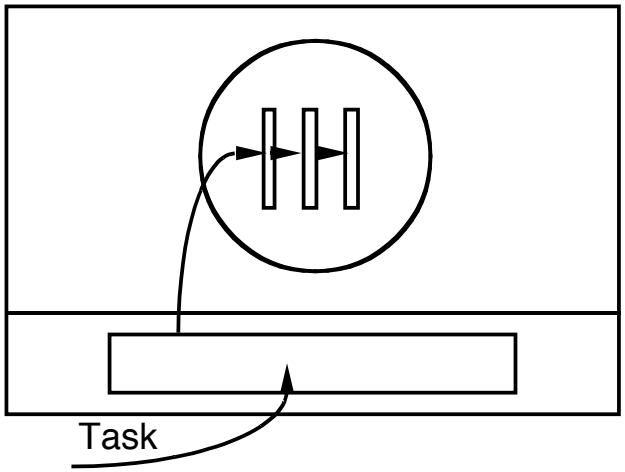
The Java Virtual Machines do not implement threads the same way.
Result of a program execution depends on the Java version, implementation, and OS variant.
Compare the execution of:

```
$ java Launcher5 1 10 1000000
```

on your Linux machines and on a Mac

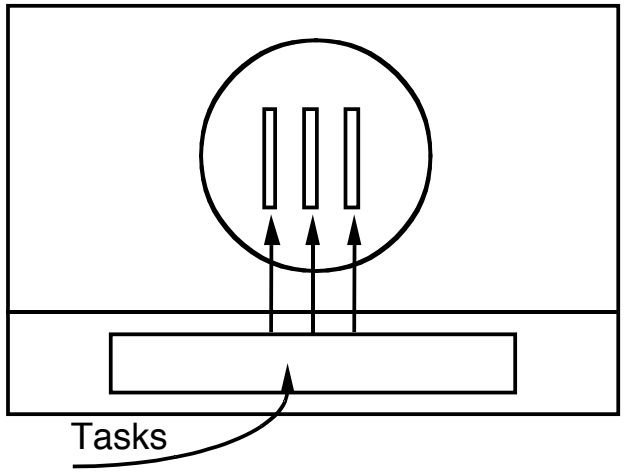
Sharing the Work Between Multiple Threads

Task pipeline



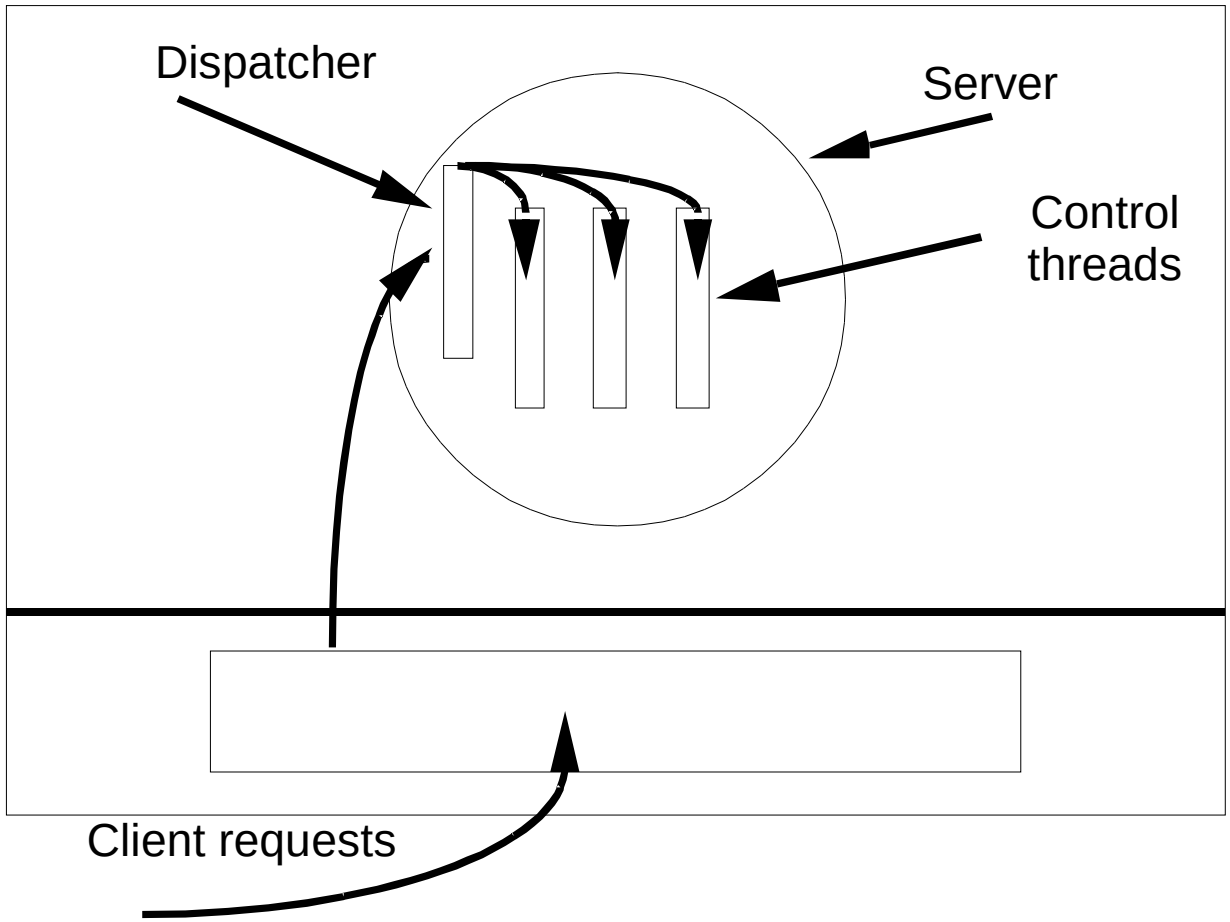
The task is split into subtasks and assigned to threads organized as a pipeline.

Multiple tasks



The tasks are marshaled and assigned to a pool of threads: Java `Executors`.

A Client-Server Organization



Java Executors

Executors are a very simple and handy way to manage threads
They are part of the `concurrent` package.
To create a thread pool for a `RunnableClass` class implementing the `Runnable` interface, the launcher:

- creates the pool with:

  ```
  ExecutorService service = Executors.newFixedThreadPool(2);
  ```

- creates the `Runnable` tasks: `Runnable task = new RunnableClass()`
- submits the tasks using `service.submit(task)` and
- shuts down the pool using `service.shutdown()`.

(MyExecutors.java)

Java Executors with Future

If the runnable object is to return a value, you have to modify:

The task: Instead of implementing `Runnable`, the class implements `Callable<ReturnValue>`

The running code: Instead of `run()`, you have a `call()` method that returns a result.

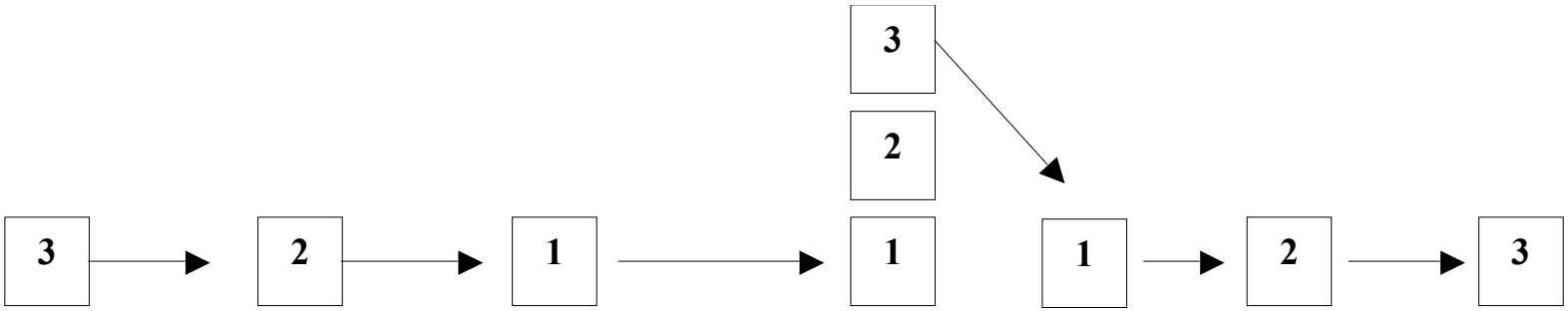
The launcher program creates a pool of threads:

```
ExecutorService service = Executors.newFixedThreadPool(2);
```

It submits the tasks using `submit()`, receives the results in the form of a `Future`, and shuts down the pool using `shutdown()`.
(MultithreadedMaxFinder.java)

Thread Coordination: A Simple Problem

Let us suppose that we want to reverse a list of numbers (countdown). A simple and inefficient algorithm would put all the items on a stack and once it is finished, remove them from the stack.



In the real world, we would divide this task into two subtasks: pile the boxes and then take them from the stack.
Let us try to implement it with two threads.

The Stack Class

Stacks are a very common LIFO data structure.
Java has a built-in `Stack` class.
`Stack` has two main methods:

- `Object push(Object item)` puts one item onto the top of the stack and
- `Object pop()` removes one item at the top of the stack and returns it.

The `empty()` method is a Boolean to test the stack state.

Wrapper Types

The `Stack` class as `List`, `Vector`, and other collections manipulates `Objects`

It cannot store primitive types like `boolean`, `int`, `float`, or `double` that are not objects

To store an integer variable, the program must associate it to an object – a wrapper

Each primitive type has an object counterpart: `char` and `Char`, `int` and `Integer`, etc.

From Java 1.5, moving an `int` to an `Integer` and the reverse are automatic and are called "boxing" and "autoboxing."

A Class to Create and Read a Stack

```
class MyStack extends Stack<Integer> {
    int stackSize;
    MyStack(int stackSize) { this.stackSize = stackSize; }
    void buildStack() {
        for (int count = 0; count < stackSize; count++) {
            this.push(count);
        }
        System.out.println("Stack complete");
    }
    void printStack() {
        while (!this.empty()) {
            System.out.println(this.pop());
        }
        System.out.println("Stack printed");
    }
}
```

A Single Threaded Program

```
public class Launcher9 {
    public static void main(String args[]) {
        int loopCount = 0;

        loopCount = Integer.parseInt(args[0]);
        MyStack myStack = new MyStack(loopCount);

        myStack.buildStack();
        myStack.printStack();
    }
}
```

A Multi-Threaded Program Sharing a Stack

Now let us create two threads to share the work:

```
BuildingThread buildingThread = new BuildingThread(myStack);
PrintingThread printingThread = new PrintingThread(myStack);
buildingThread.start();
printingThread.start();

class BuildingThread extends Thread {
    MyStack myStack;
    BuildingThread(MyStack myStack) {
        this.myStack = myStack;
    }
    public void run() {
        myStack.buildStack();
    }
}
```

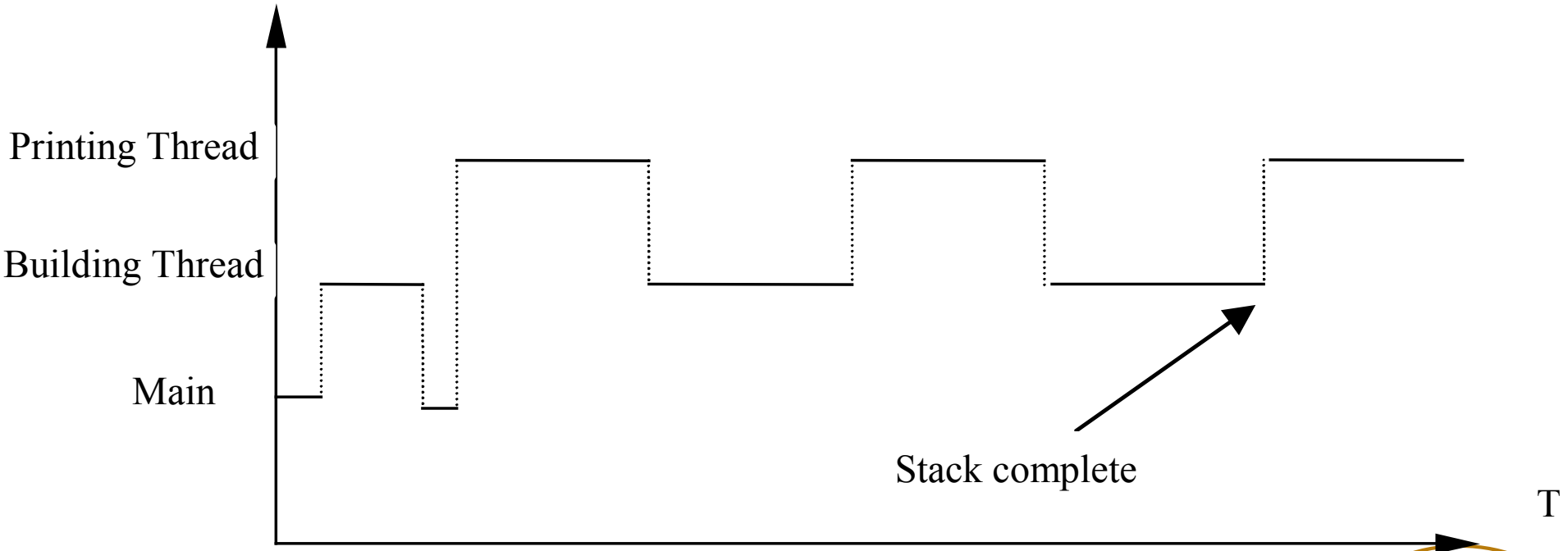
A Multi-Threaded Program (Continued)

```
class PrintingThread extends Thread {
    MyStack myStack;
    PrintingThread(MyStack myStack) {
        this.myStack = myStack;
    }
    public void run() {
        myStack.printStack();
    }
}
```

(Launcher10.java)
Better design? What do you think?

The Execution Flow

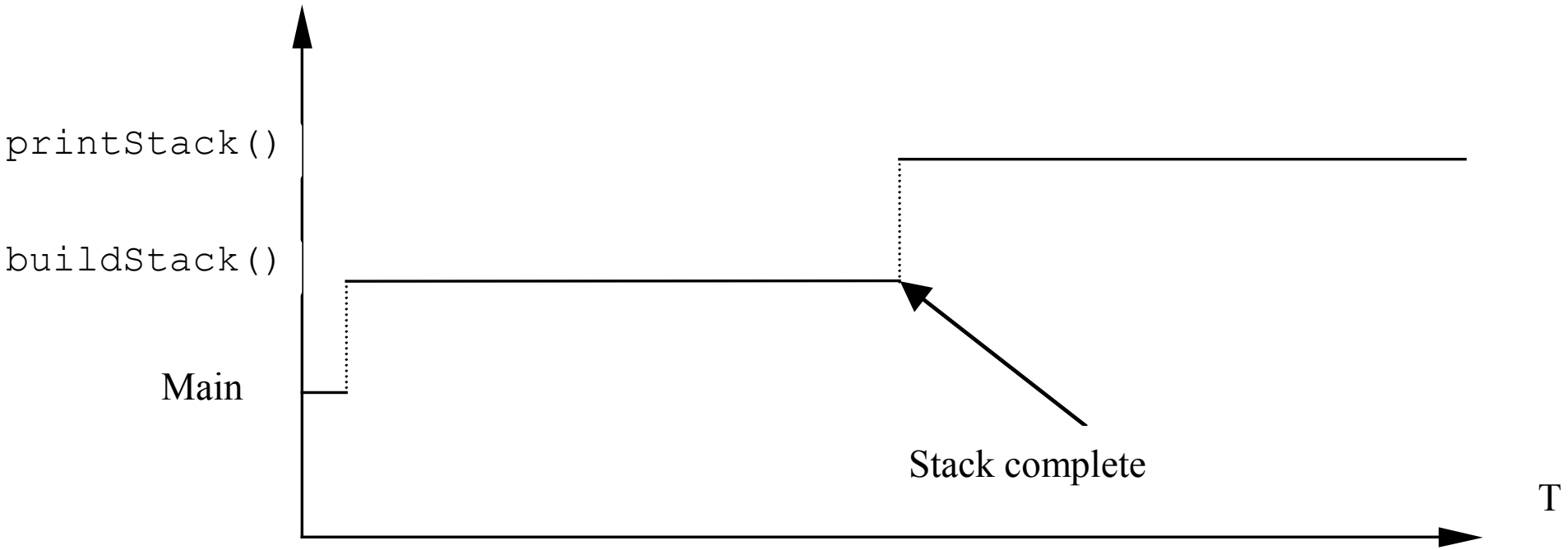
The scheduler shares the time between the two threads



The lack of coordination produces a garbled output

What Should the Execution be

The scheduler must run the `printStack()` method after the `buildStack()` method is complete



The code sections where the stack is being accessed – built and read – are **critical sections**

Their access must be **exclusive**: one thread at a time

Busy Waiting

A first solution is to test continuously a condition before entering the critical section

The condition is set when the task is complete

```
class BuildingThread extends Thread {
    MyStack myStack;
    volatile boolean complete = false;
    BuildingThread(MyStack myStack) {
        this.myStack = myStack;
    }
    public void run() {
        myStack.buildStack();
        complete = true;
    }
    boolean getStatus() {
        return complete;
    }
}
```

Busy Waiting (Continued)

The condition is tested before starting the 2nd thread (Launcher11.java)
It is called **busy waiting**

```
while (buildingThread.getStatus() == false) {
    ;
}
```

Busy waiting requires an atomic access to the condition variable
This is implemented using the `volatile` keyword
Busy waiting is generally **not** a good solution
An improved program would test the condition in the 2nd thread and use `yield()` if it is not met
`yield()` moves the executing thread to `runnable` and allows the scheduler to select and run another thread
It is a **poor design** too.

Monitors

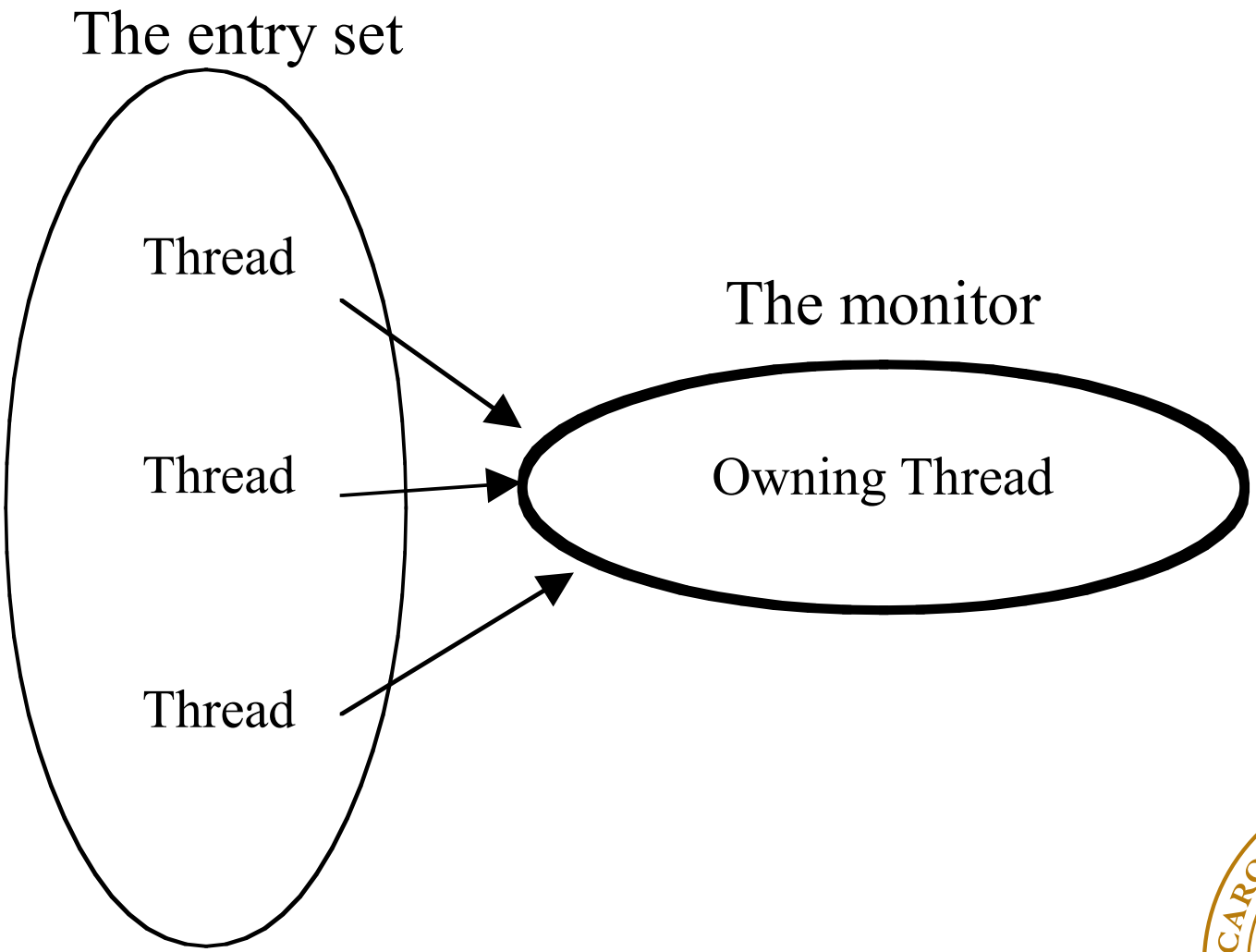
Monitors are constructs that guarantee the mutual exclusion of methods
Per Brinch Hansen developed this concept of monitor in 1973
Any Java object is a potential monitor
The `synchronized` keyword declares the object methods that are part of a monitor

```
class MyClass {
    synchronized void m1() {}
    void m2() {}
    synchronized void m3() {}
}
```

Monitors (Continued)

The methods `m1()` and `m3()` are part of the monitor: `myObject.m1()` and `myObject.m3()` won't run concurrently
The first method started must be finished before another one is started
Similarly, two threads can't run `myObject.m1()` concurrently
The method `myObject.m2()` is not part of the monitor. It can be run at any time
The set of threads competing to acquire a monitor is called the **entry set**
The Boolean method `holdsLock(Object)` returns true if the thread holds the monitor lock

The Entry Set



The New Class

```
class MyStack extends Stack {
    int stackSize;
    MyStack(int stackSize) { this.stackSize = stackSize; }
    synchronized void buildStack() {
        for (int count = 0; count < stackSize; count++) {
            this.push(count);
        }
        System.out.println("Stack complete");
    }
    synchronized void printStack() {
        while (!this.empty()) {
            System.out.println(this.pop());
        }
        System.out.println("Stack printed");
    }
} //Launcher12.java
```

Race Conditions

What happens if threads are started the other way around?

```
printingThread.start();
buildingThread.start(); // Launcher13.java
```

instead of

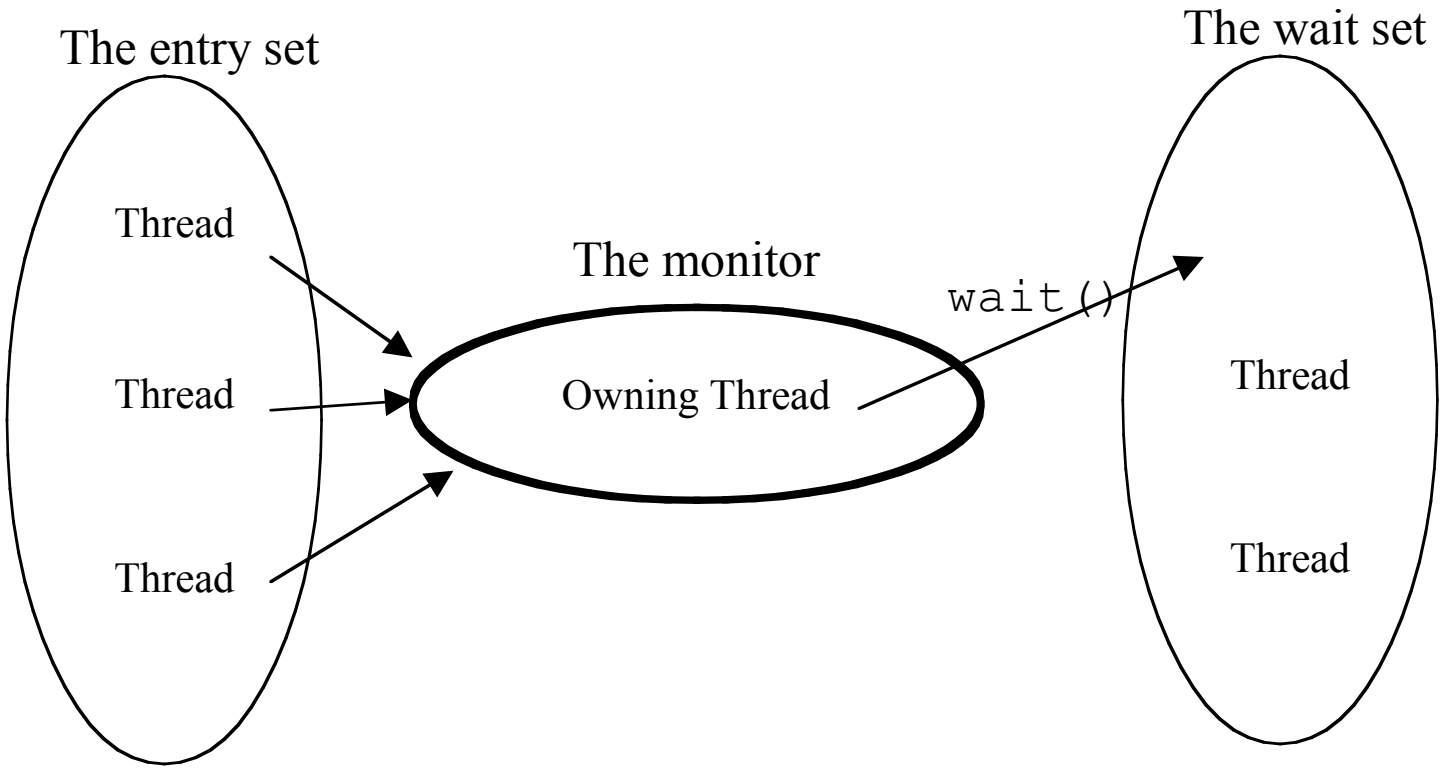
```
buildingThread.start();
printingThread.start(); // Launcher12.java
```

The result depends on the particular order of the instructions
This is called a race condition
Can we improve the monitor to avoid it?

Introducing the `wait()` Method

When a thread runs a synchronized method, it owns the object exclusively. The others are in the `blocked` state.
Sometimes the object is not ready as when the stack is empty. The thread is unable to start or continue.
It must wait then...
The `wait()` method moves the thread from `running` to the `waiting` state and places it in a waiting list – the **wait set**.
All objects inherits the `wait()` method as potential monitors (`this.wait()`)

The Wait Set


The entry set
Thread
Thread
Thread
The monitor
Owning Thread
wait()
The wait set
Thread
Thread

The wait() Method (Continued)

The new code is:

```
synchronized void printStack() {
    while (this.empty()) { // do not use if!
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    while (!this.empty()) {
        System.out.println(this.pop());
    }
    System.out.println("Stack printed");
} // Launcher14.java
```

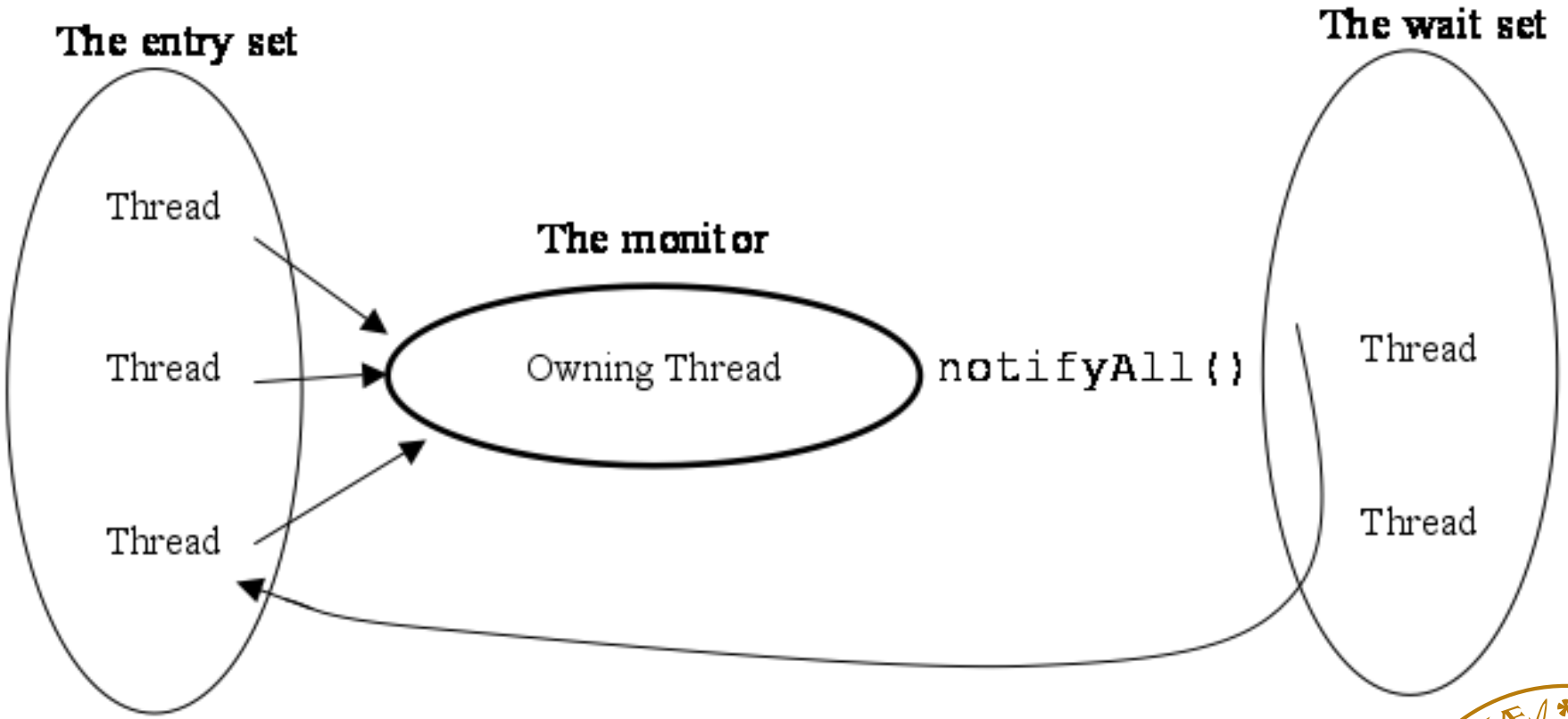
The stack is not printed! Why?

The notify() Method

After a wait() call, the thread is stuck in the wait set
The notify() method selects arbitrarily one thread from the wait set and moves it to the entry set and the runnable state
The notifyAll() method moves all the threads in the wait set to the entry set and to the runnable state

```
synchronized void buildStack() {
    for (int count = 0; count < stackSize; count++) {
        this.push(count);
    }
    System.out.println("Stack complete");
    notifyAll();
} // Launcher15.java
```

The notifyAll() Method



Exiting the Wait Set

A thread exits the wait set when it is "notified"
It is also possible to set a time limit to `wait()` using
`public final void wait (long milliseconds)`
or
`public final void wait (long milliseconds, int nanos)`
The `nanos` value is not reliable however
This moves the thread in the `timed_waiting` state, similar to `waiting`.
Finally, the `interrupt()` method of the `Thread` class enables a thread to exit the wait set

The interrupt() Method

Under normal running conditions, interrupt() sets the interrupt status and has not other effects
When the thread is in the waiting state because of wait(), sleep(), or join(), it receives an InterruptedException
Input/output blocks a running thread until the I/O is completed. With the nio package, interrupt() wakes up a thread in an I/O method.
The Boolean method isInterrupted()returns the status value and interrupted() returns and clears it

```
printingThread.start();
printingThread.interrupt();
buildingThread.start(); //(Launcher16.java)
```

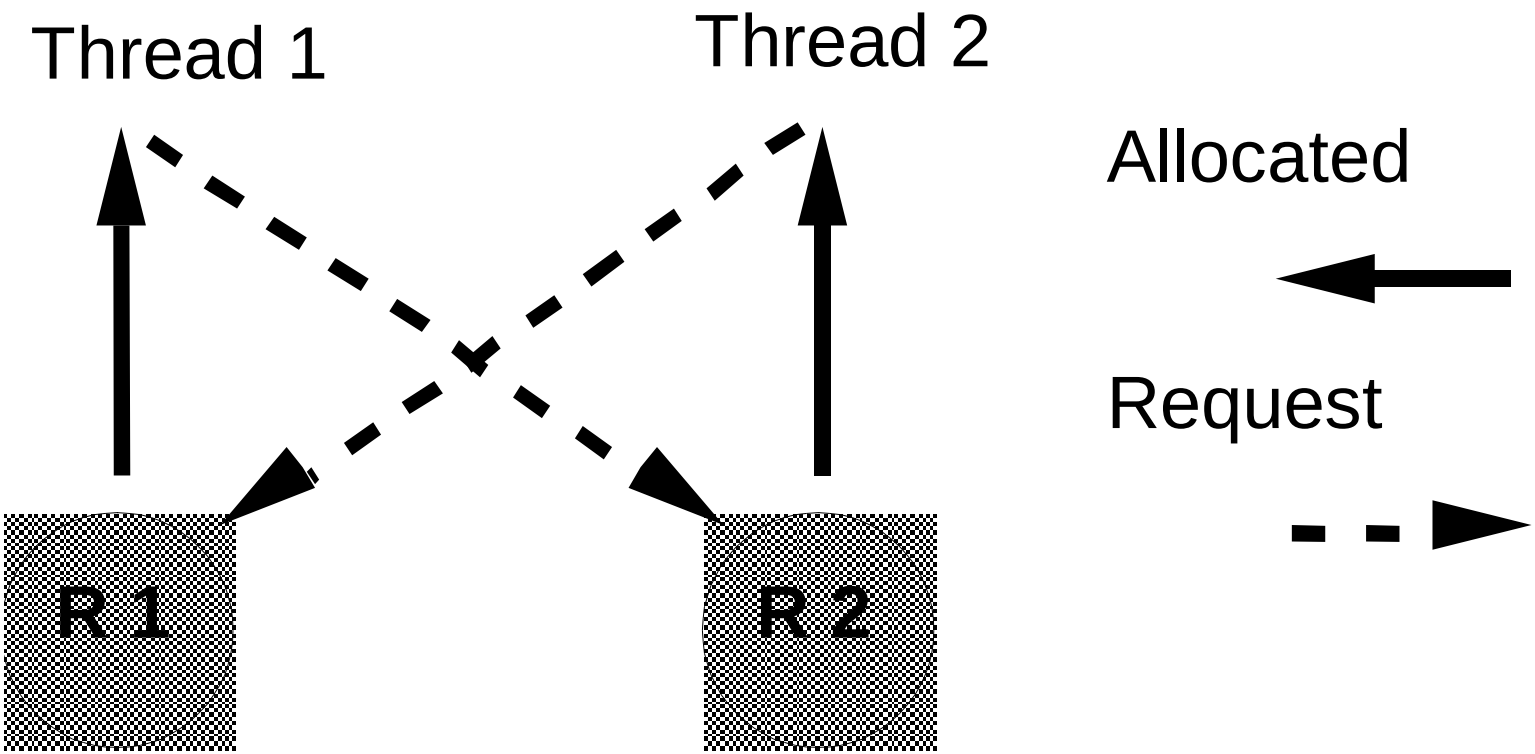
Deadlocks

Threads programming is difficult to master well
Deadlocks are a major source of bugs
A deadlock occurs when these conditions are met:

1. A thread has an exclusive resource that another thread is waiting for and
2. The other thread has a resource that the first thread is waiting for

It is a hopeless circular wait

Deadlocks (Continued)



A Deadlock Example

In addition to methods, blocks of code can be synchronized as:

```
synchronized (Object) {
...
}
```

Objects can wait and notify using `Object.wait()` and `Object.notify()`
Let us program a deadlock: A first thread acquires two synchronized objects, `lock1` and `lock2` and a second thread acquires the same objects the other way around

A Deadlock Example

```
class Stuck1 extends Thread {
    Integer lock1, lock2;
    Stuck1(String name, Integer lock1, Integer lock2) {
        super(name);
        this.lock1 = lock1;
        this.lock2 = lock2;
    }
    public void run() {
        synchronized (lock1) {
            System.out.println(getName() + " acquired lock1");
            synchronized (lock2) {
                System.out.println(getName() + " acquired lock2");
            }
        }
    }
}
```

A Deadlock Example

```
class Stuck2 extends Thread {
    Integer lock1, lock2;
    Stuck2(String name, Integer lock1, Integer lock2) {
        super(name);
        this.lock1 = lock1;
        this.lock2 = lock2;
    }
    public void run() {
        synchronized (lock2) {
            System.out.println(getName() + " acquired lock2");
            synchronized (lock1) {
                System.out.println(getName() + " acquired lock1");
            }
        }
    }
}
```

A Deadlock Example

```
public class Launcher17 {
    public static void main(String args[]) {
        Integer lock1 = new Integer(1), lock2 = new Integer(2);
        Stuck1 stuck1 = new Stuck1("Stuck1", lock1, lock2);
        Stuck2 stuck2 = new Stuck2("Stuck2", lock1, lock2);
        stuck1.start();
        stuck2.start();
    }
}
```

The deadlock is not systematic. It depends on the completion speed of `stuck1`

Reentrance

A single thread can't deadlock itself however because Java monitors are reentrant

```
public class Reentrant {
    public synchronized void a() {
        b();
        System.out.println("Running a()");
    }
    public synchronized void b() {
        System.out.println("Running b()");
    }
} // Launcher18.java
```

Thread Death

A thread terminates when it returns from the `run()` method.
Do not use `stop()`
Instead of using synchronized methods, we could simply have waited the end the building thread.
This is possible using `join()` that waits for a thread to finish.
`isAlive()` tests if a thread is alive. It returns false if it is dead or not started.
(Launcher19.java)

Semaphores

Semaphores are another type of coordination device
They are widely used although more difficult than monitors
They are available on Unix in the IPC library and from version 1.5.0 of Java in the `java.util.concurrent` package
A semaphore is a positive integer that is decremented and incremented atomically using the P and V operations
A mutex is a semaphore initialized to one
It enables to protect a critical section as in

```
mutex = 1
P(mutex)
criticalSection()
V(mutex)
```