Concurrent Programming (Part II)
Lecture 3: Java Threads

Dr Kevin Bryson

K.Bryson@cs.ucl.ac.uk

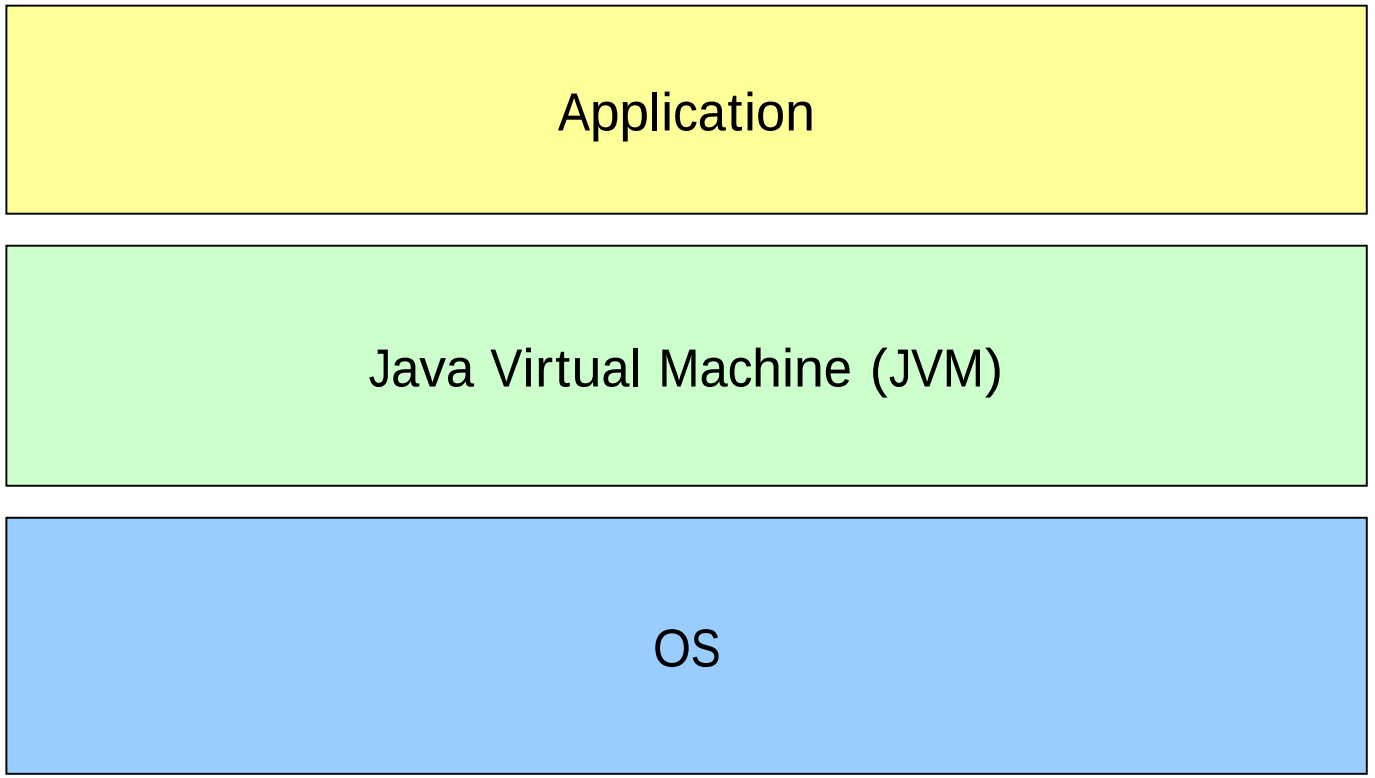
Room: 8.04

Lecture Overview

- We formally introduced the *Concurrency Abstraction* in the previous lecture. Essentially a concurrent system can be reasoned with by considering **all possible interleavings** of the **atomic actions** *that make up the processes.*
- We tried to demonstrate that this type of reasoning is valid for:
  - Multitasking within programs (i.e. Threads) or operating systems (i.e. OS processes)
  - Multiprocessors carry out multiple jobs truly in parallel.
  - Multicomputing … i.e. distributed systems working in parallel.

- In this lecture we will begin looking at how to write concurrent programs in Java.

The Java Architecture (an abstract view !)



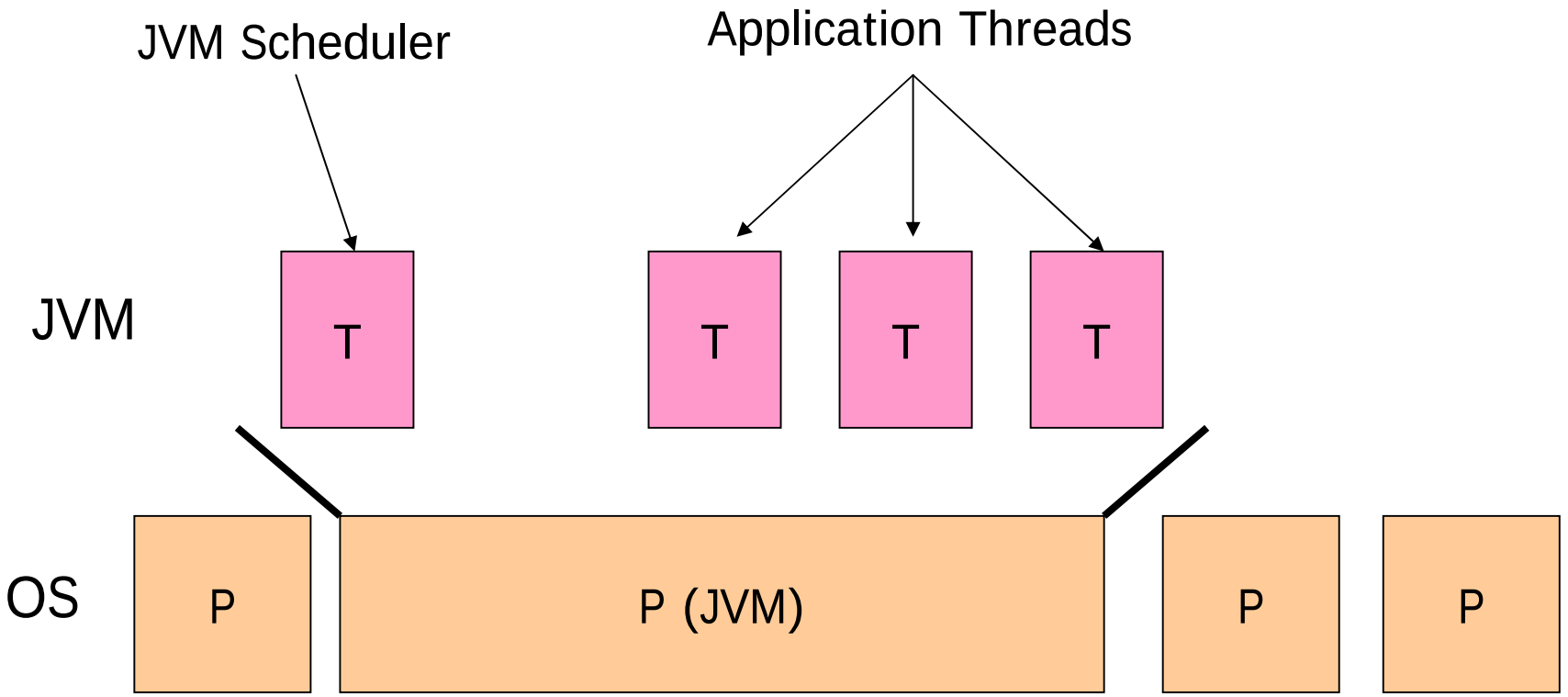
Processes and Threads

- The OS runs many processes (we are now using this term in the concrete sense rather than the abstract as in the previous lecture) which share the CPU(s).
- The processes do not share address space (i.e. they have their own code, stack and data).
- The OS scheduler controls how each of the processes is allocated time on the CPU resource(s).
- If there is only one CPU there is no real parallelism (only logical, abstract or virtual concurrency)

Processes and Threads

- The JVM runs as a single process on top of the OS
- JVM allows the spawning of **threads**
- ***Threads co-live in the same address space of the process, sharing code and data.***
- The JVM has a scheduler for the threads which sits on top of the OS scheduler
- The JVM scheduler is simply another thread in the JVM process!

Operating System Processes and JVM Threads

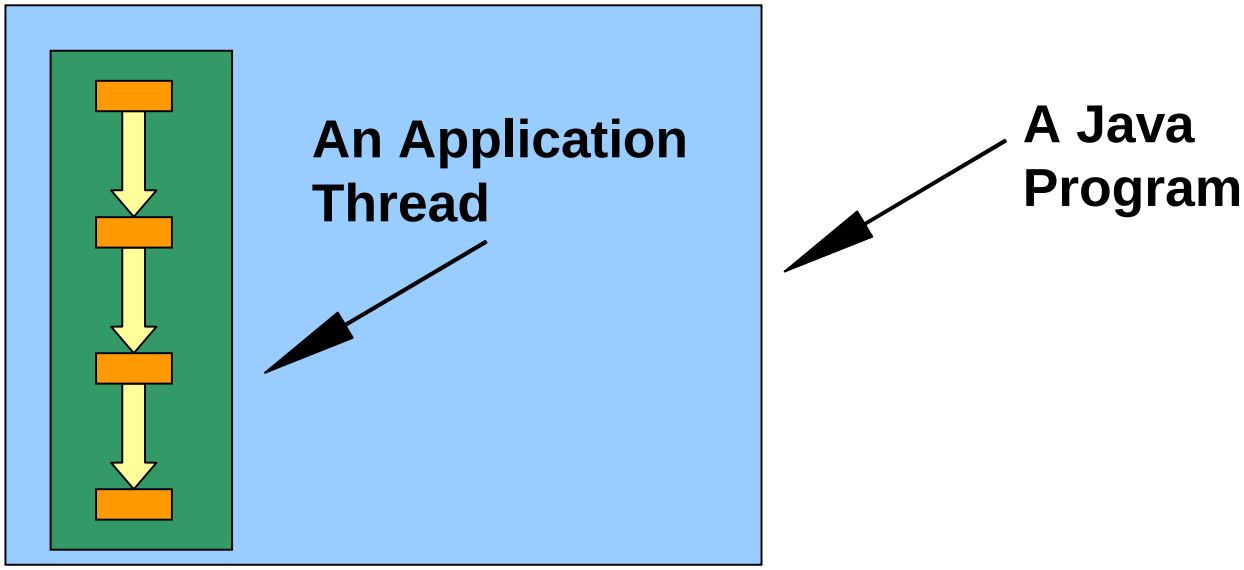


Threads

- Threads share code and heap space.
- Each thread has its own program counter, stack (for procedure calls) and ‘working memory’ (e.g. registers) that can store local versions of variables which the thread is currently working upon.
- ***Context-switching*** (ie scheduling a different thread for execution) happens while thread code is being executed. Note that when a thread/process is context-switched on a processor … then it saves all the current registers and restores all the previous registers for the newly executing thread/process.

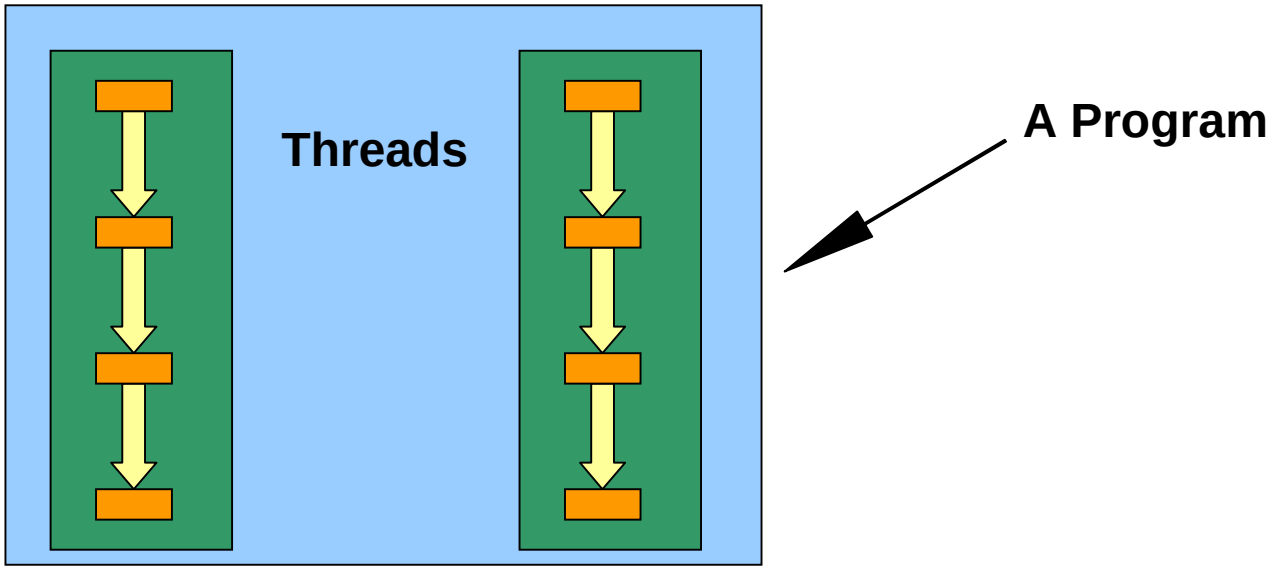
A Java Program

- Threads are actually Java objects which provide a single sequential flow of control within a program (a *Thread* of execution)

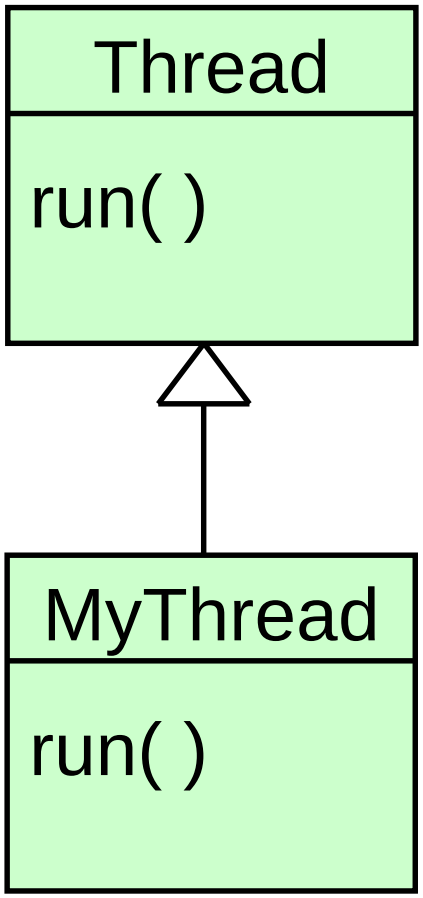


Multi-Threaded Java Program

- Concurrent activities are provided by creating multiple threads in a program (**multi-threaded**)
- *So how are Threads created ?*
  There are two ways…



1) Inheriting from the Thread class



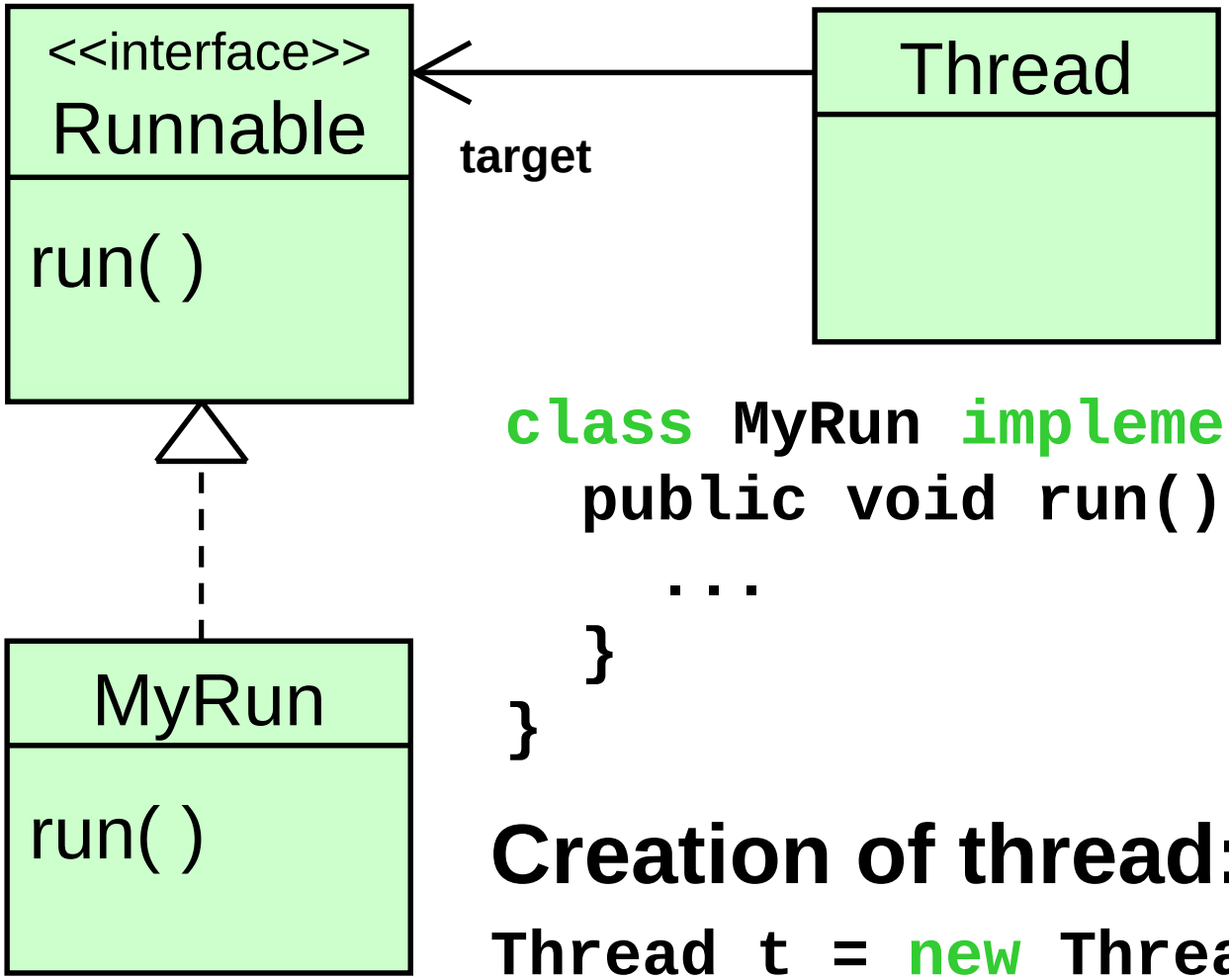

```
class MyThread extends Thread {
   public void run() {
      ...
   }
}
```

Creation of thread:

```
MyThread t = new MyThread();
```

How does this restrict MyThread ?

2) Implementing the Runnable Interface




```
class MyRun implements Runnable {
   public void run() {
      ...
   }
}
```

Creation of thread:

```
Thread t = new Thread(new MyRun());
```

How do these approaches differ?

- One is through subclasses and one uses interfaces

- Java does not allow multiple inheritance
- With the first method the class MyThread cannot inherit from other classes (as well as from Thread)
- With the second method this is possible

Available Methods of the Thread Class

```
package java.lang;

public class Thread implements Runnable {

   public void    start();
   public void    run();
   public void    stop();    // Deprecated - DO NOT USE.
   public void    resume();  // Deprecated - DO NOT USE.
   public void    suspend(); // Deprecated - DO NOT USE.
   public boolean isAlive();

   public Thread.State getState();

   public static void sleep(long millis);
   public static void sleep(long millis, int nanos);

   public void           interrupt();
   public boolean        isInterrupted();
   public static boolean interrupted();

   public void join() throws InterruptedException;
}
```

The Runnable Interface

THIS SIMPLE INTERFACE IS THE KEY !

You write a class which ‘implements’ this interface and provides a concrete ‘run’ method which defines what the Thread must do when it is running.

```
package java.lang;

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

Thread States - returned by *getState()*

NEW

A thread that has not yet started is in this state.

RUNNABLE

A thread executing in the Java virtual machine is in this state.

BLOCKED

A thread that is blocked waiting for a monitor lock is in this state.

WAITING

A thread that is waiting indefinitely for another thread to perform a particular action is in this state.

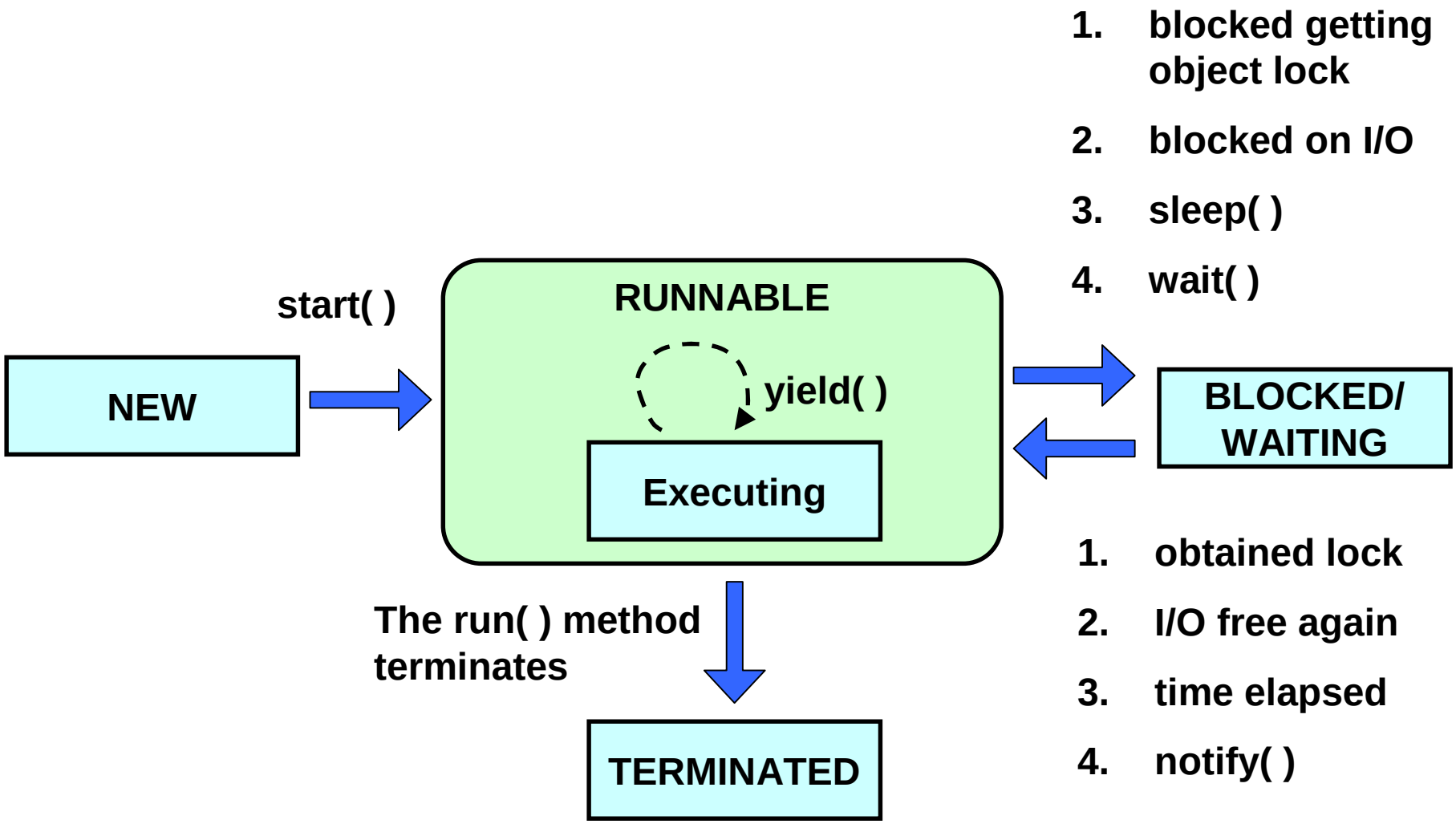
TIMED_WAITING

A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.

TERMINATED

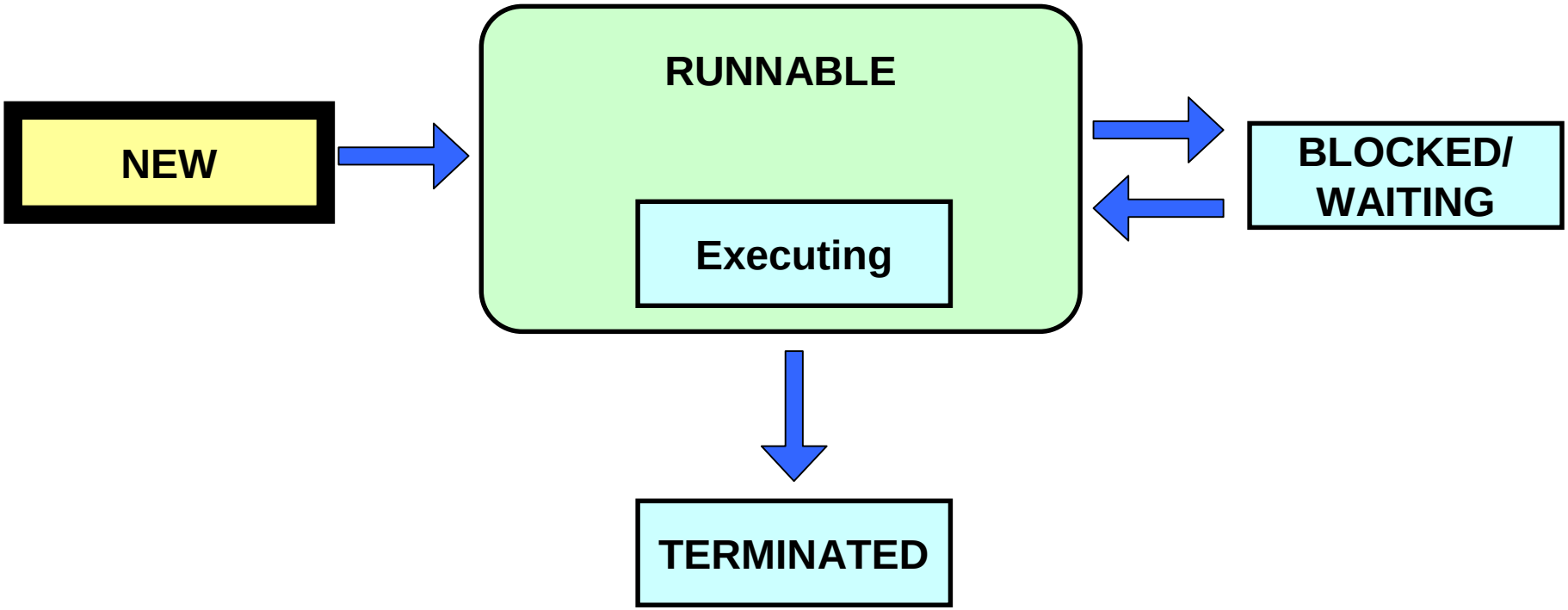
A thread that has exited is in this state.

Thread Lifecycle



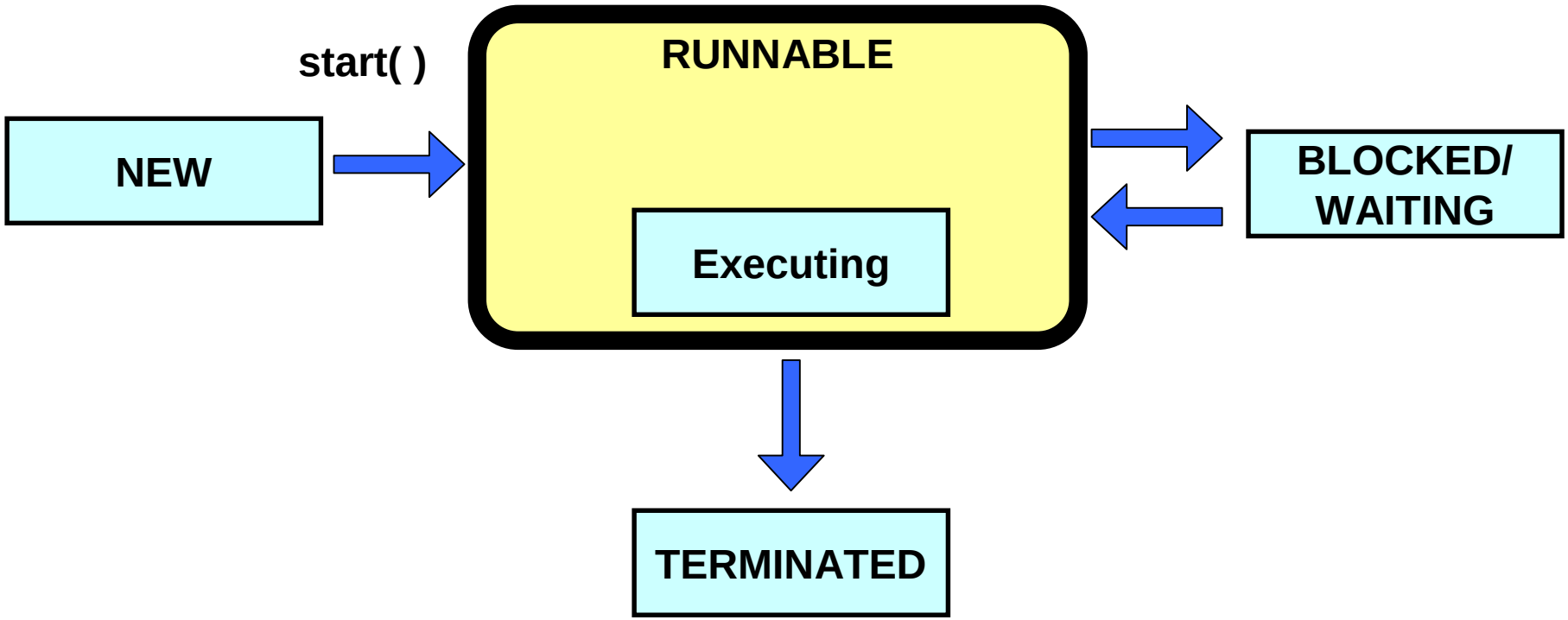
A newly created Thread is largely an empty object with no resources allocated

```
MyThread t = new MyThread();
```

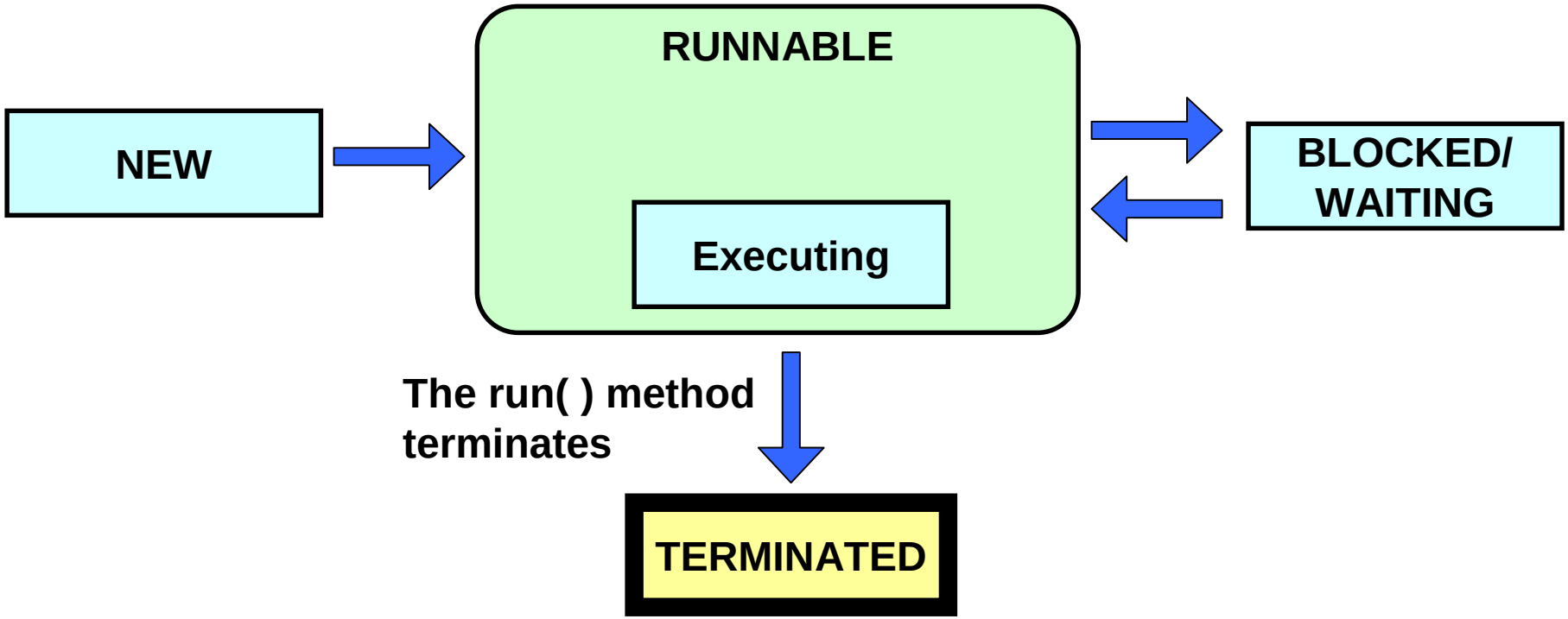


Started with start(), which invokes run() *within a new thread of execution* – resources are allocated for running the Thread

```
t.start()
```

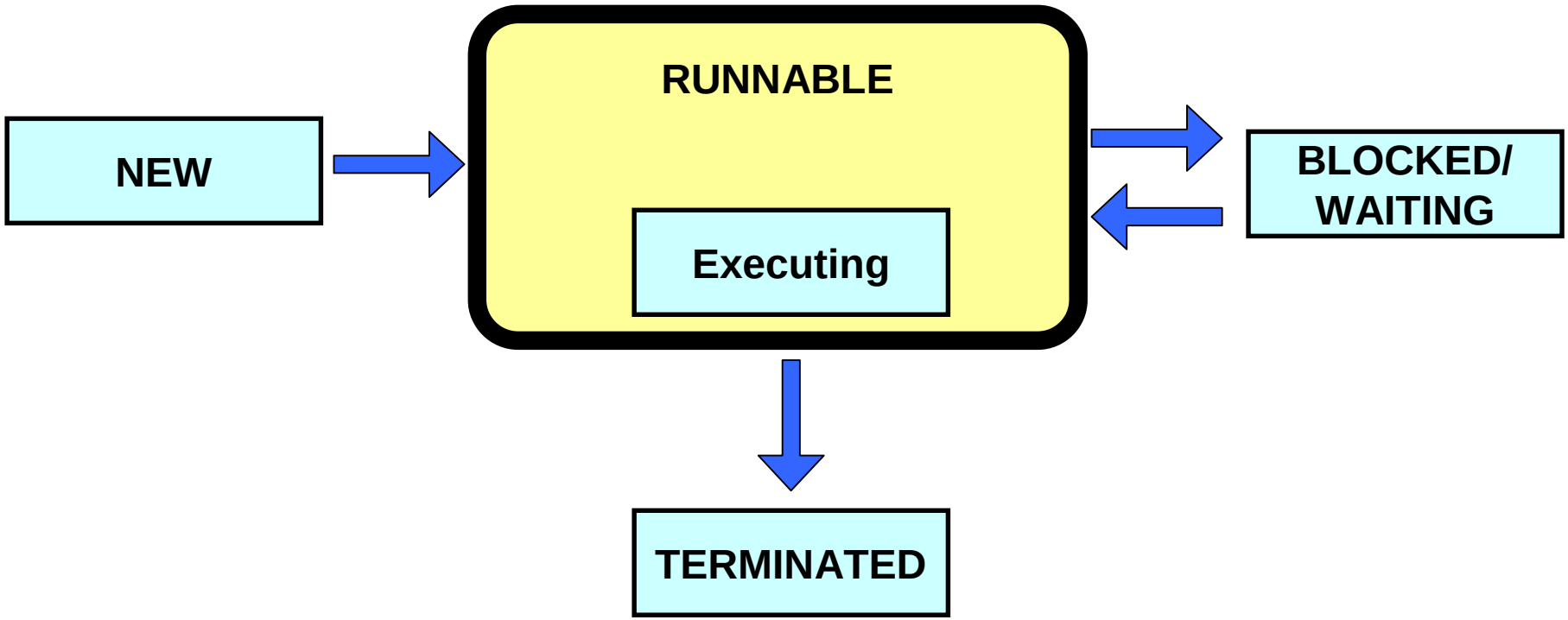


Terminated when run() returns – resources are released



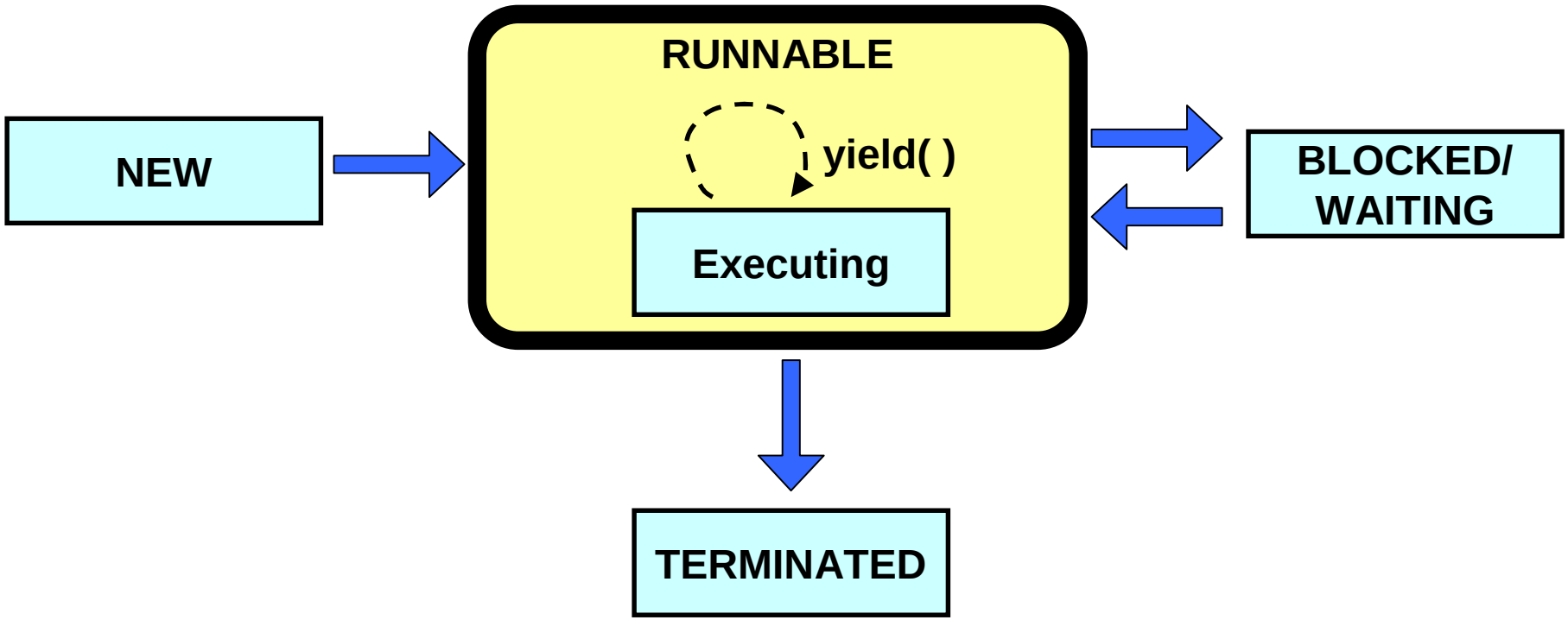
When a Thread is ‘RUNNABLE’ … it does not need to be Running (Executing on the Processor) !

The Thread may be waiting for other resources from the operating system such as a processor to actually ‘run’ on. CPU is shared between Runnable Threads (Scheduling)



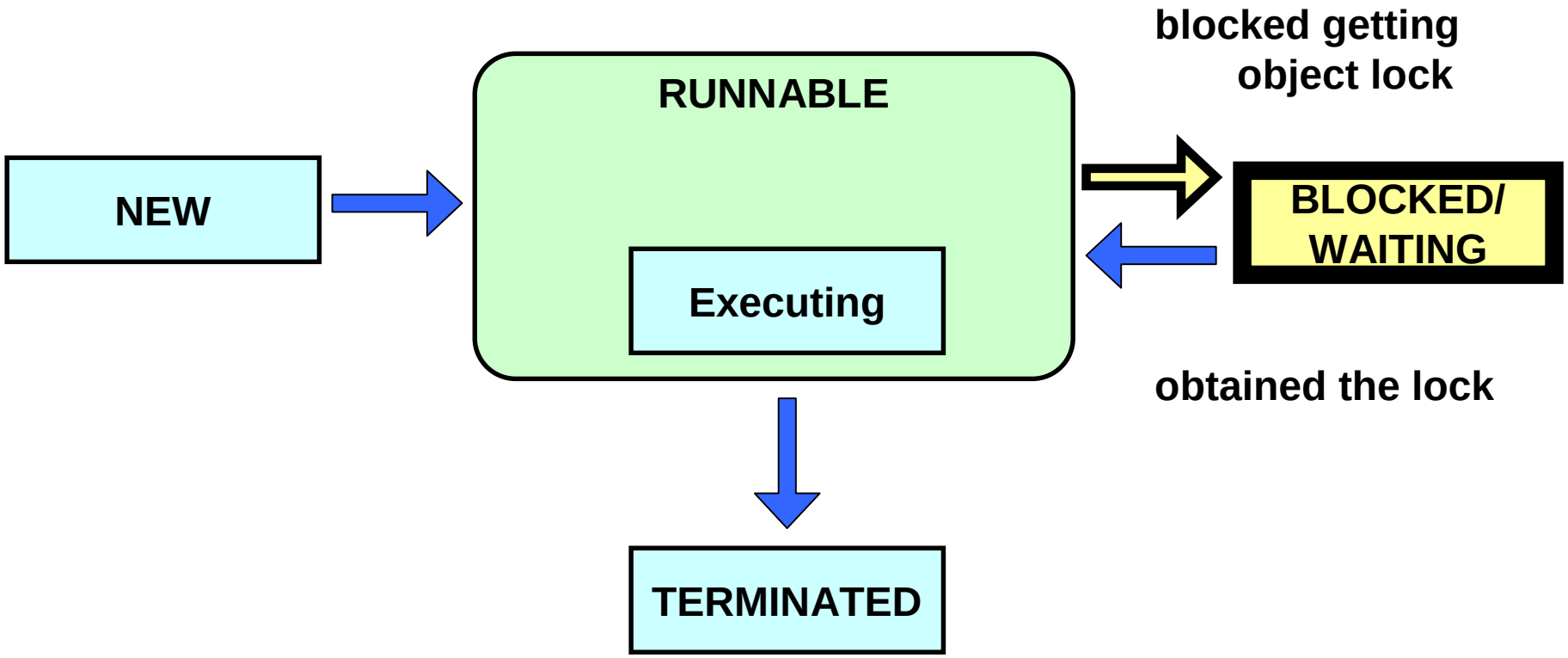
A Thread which is allocated to a processor can yield(), giving the JVM scheduler a chance to select another Thread for execution (more about this later)

Such a Thread is well-behaved and not selfish !



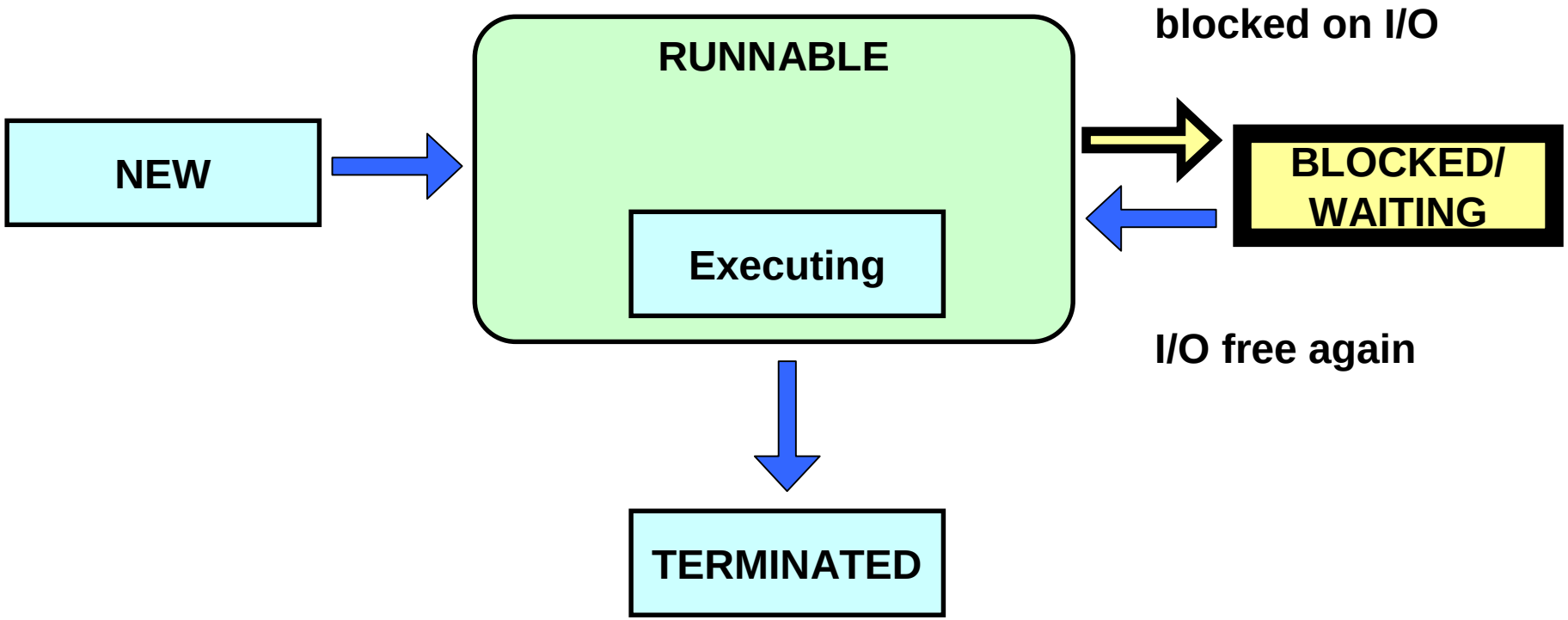
Blocked acquiring an object lock ('monitor')

Like the fridge scenario, a thread may be blocked while waiting to gain 'access' to a locked object. It will only become RUNNABLE again once it gains the object lock.



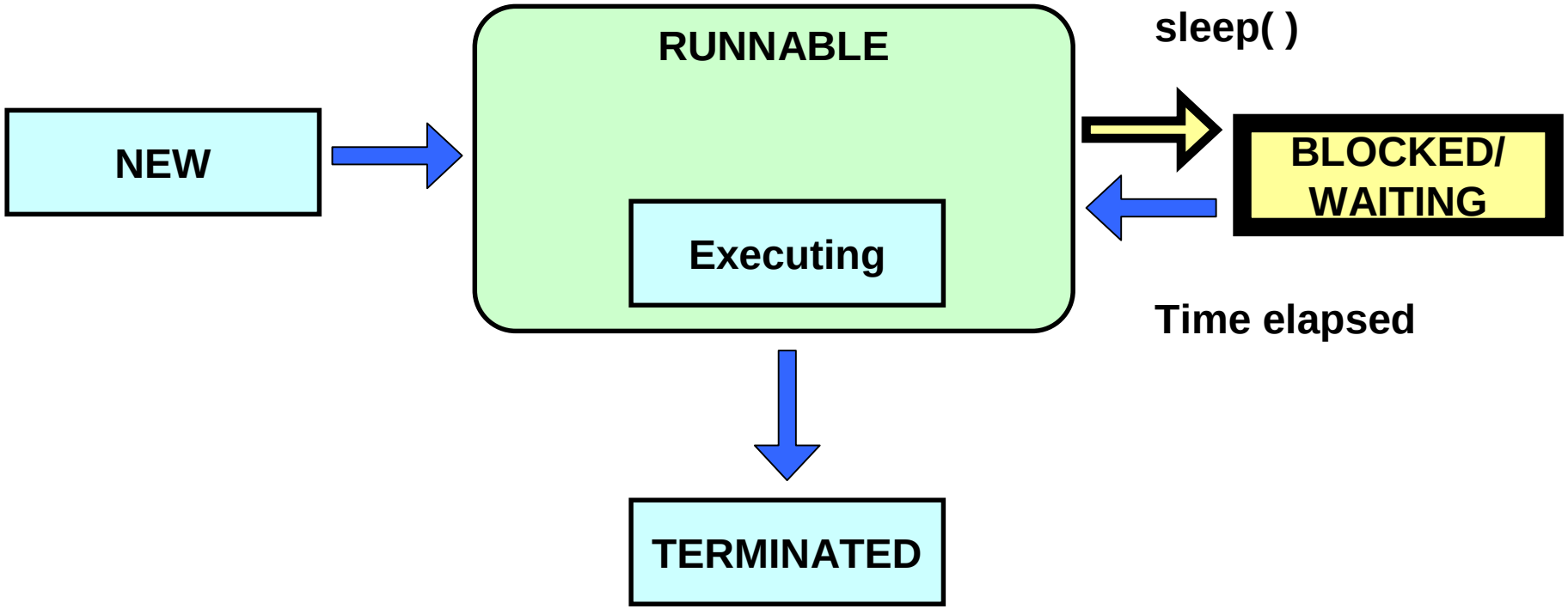
Blocked on I/O …

A Thread may become ‘Not Runnable’
if it is blocked carrying out Input/Output.
It becomes Runnable again once free.



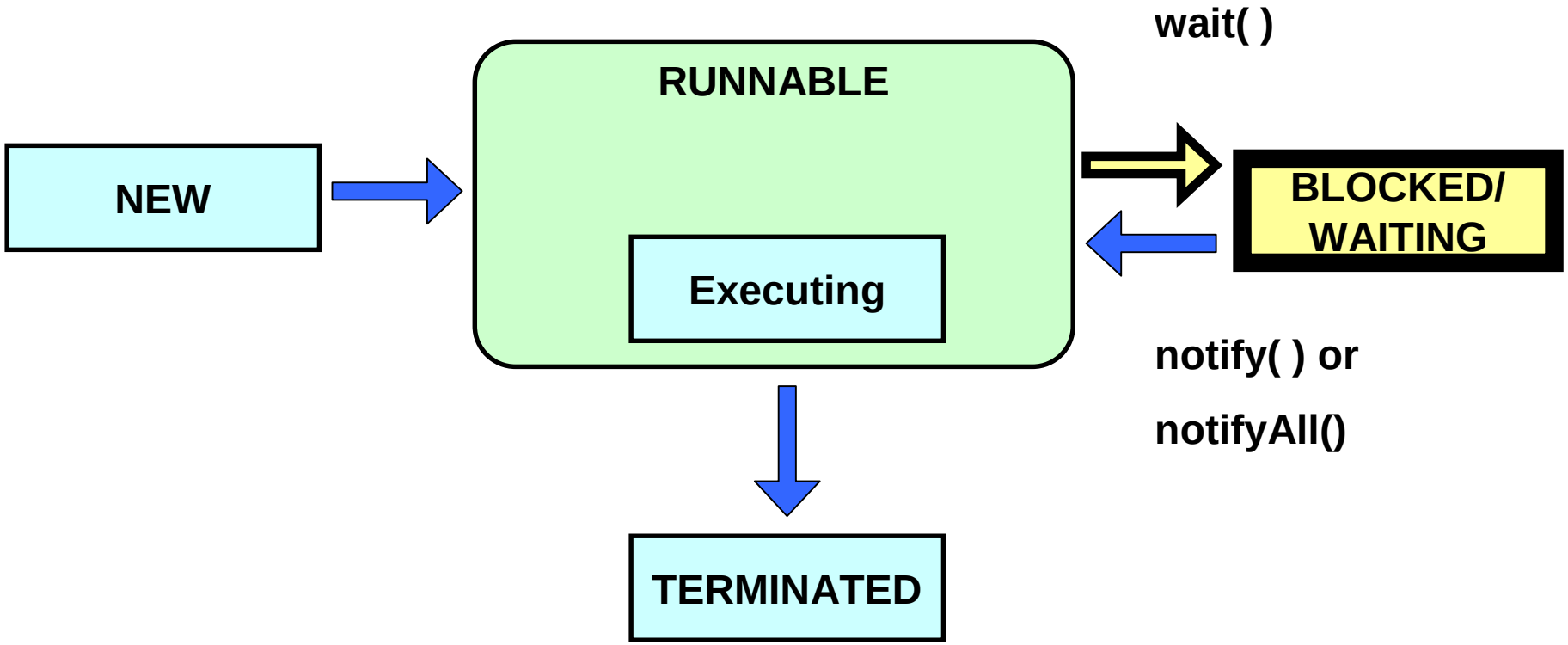
A sleeping Thread ...

A Thread may be told to sleep a certain number milliseconds (and nanoseconds depending on precision of system timers) and does not become runnable again until the time has elapsed.

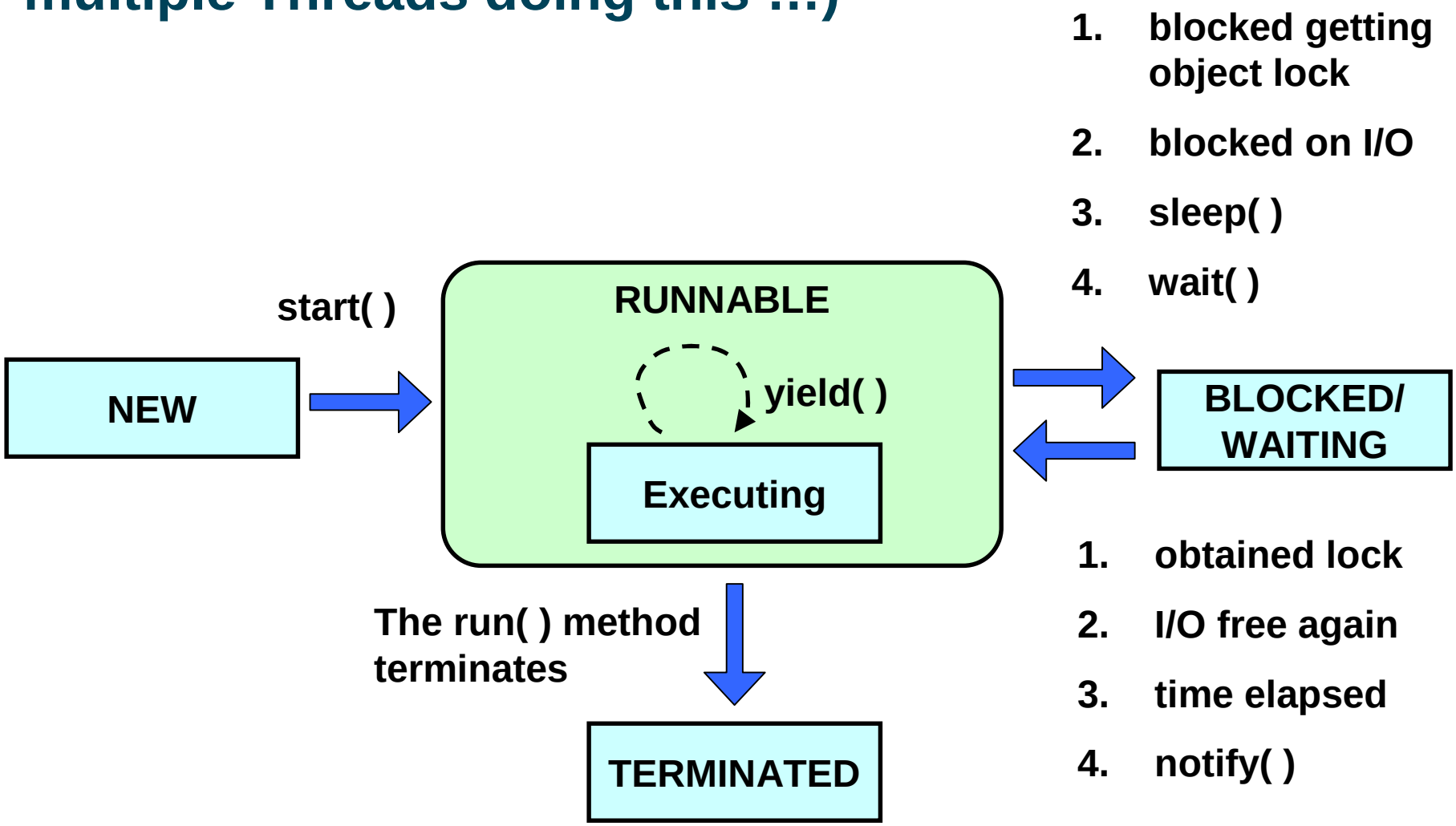


Waiting for another Thread …

A Thread may become ‘Not Runnable’ since it is waiting to be notified by another Thread to continue (dealt with in a future lecture).

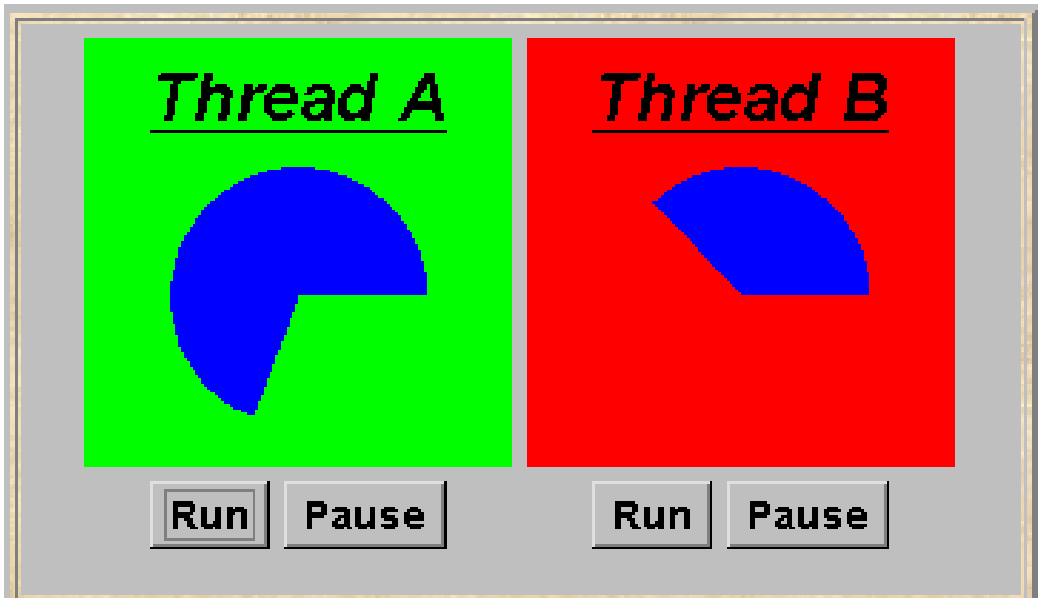


Combining all these aspects together gives the complete Thread Lifecycle (remember there are multiple Threads doing this …)



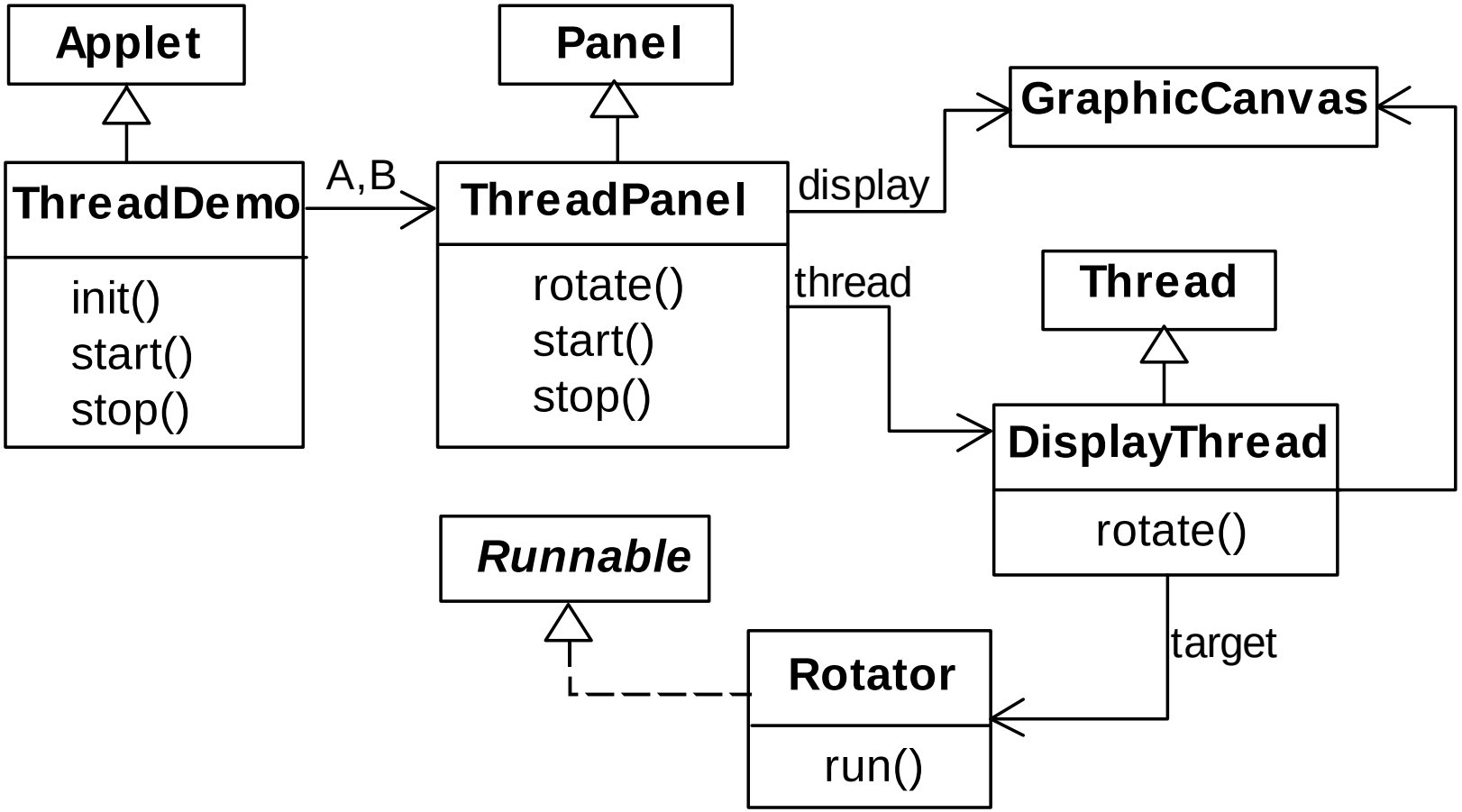
Simple Thread Demo Example (Magee & Kramer)

http://www.doc.ic.ac.uk/~jnm/book/book_applets/ThreadDemo.html



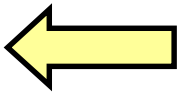
Let's see what it does …

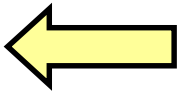
Thread Demo Example: structured to be used as reusable components of other concurrency demos



Thread Demo Example

```
public class ThreadPanel extends Panel {

  // construct display with title and segment  color c
  public ThreadPanel(String title, Color c) {…}

  // rotate display of currently running thread 6 degrees
  // return value not used in this example
  public static boolean rotate()
         throws InterruptedException {…}

  // create a new thread with target r and start it running
  public void start(Runnable r) {
        thread = new DisplayThread(canvas,r,…);
        thread.start();
   }

  // stop the thread using Thread.interrupt()
  public void stop() {thread.interrupt();}
}
```





Summary

- Relationship between OS Processes, JVM & Threads of execution.

- The two ways of creating Java threads.
- Methods within the Thread class.
- Thread lifecycle.

- You can now write a program which can do two or more things simultaneously … using Threads.

- In the next lecture we will look at what happens when Threads interact with each other – and what mechanisms you should be using to guarantee correctness and portability in your concurrent programs.