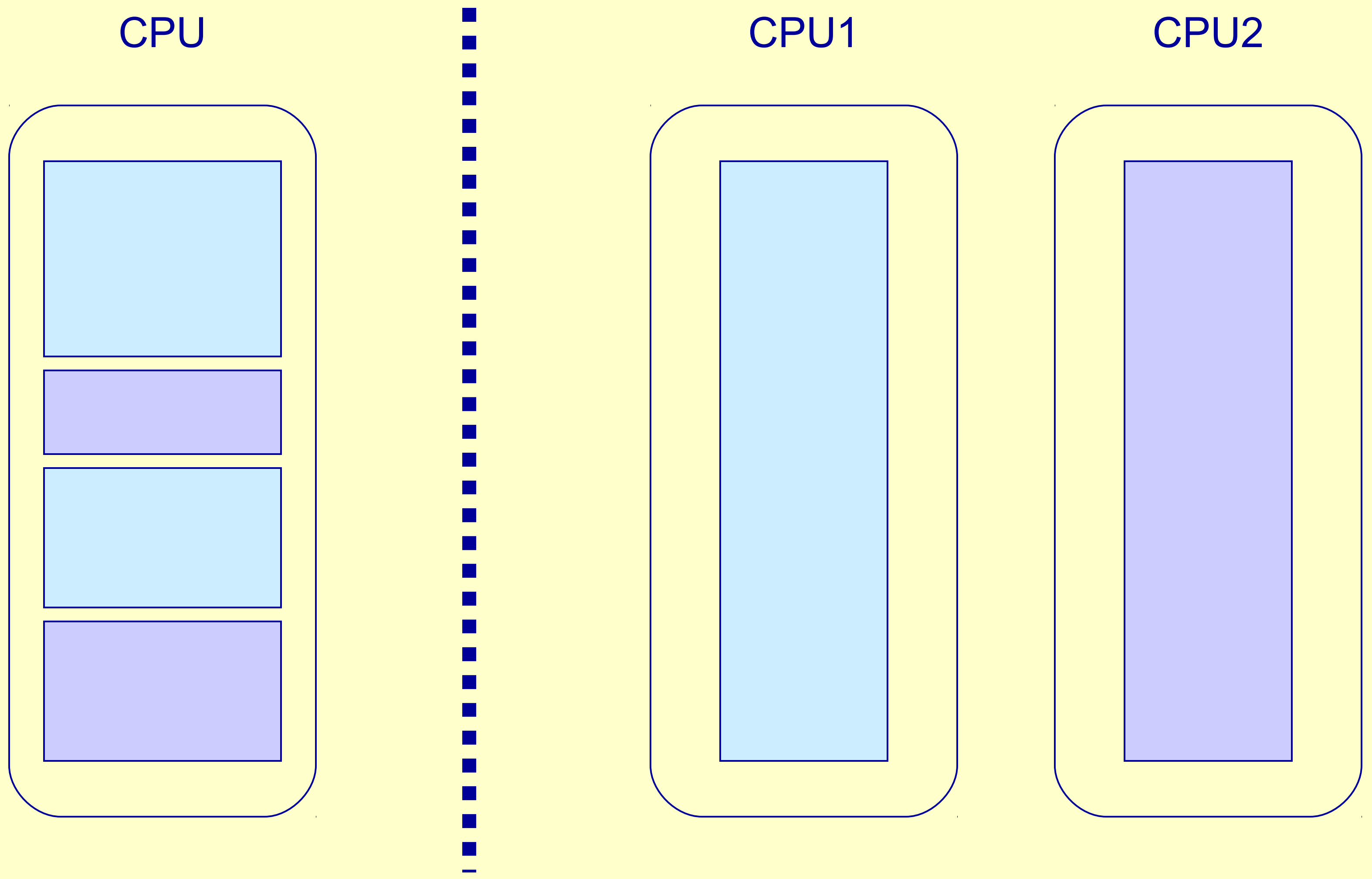
Java Threads

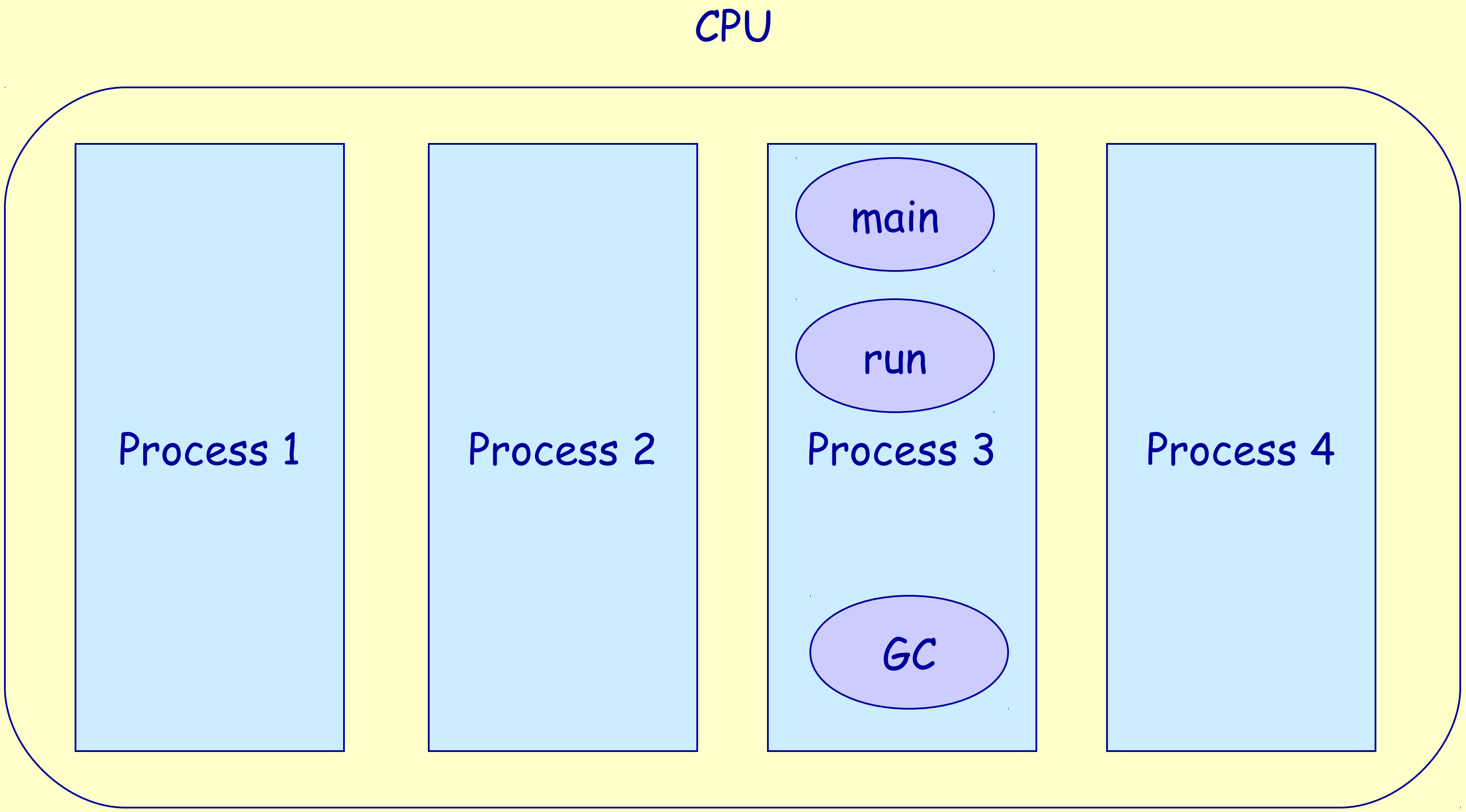
Multitasking and Multithreading

- Multitasking:
  - refers to a computer's ability to perform multiple jobs concurrently
  - more than one program are running concurrently, e.g., UNIX
- Multithreading:
  - A thread is a single sequence of execution within a program
  - refers to multiple threads of control within a single program
  - each program can run multiple threads of control within it, e.g., Web Browser

Concurrency vs. Parallelism



Threads and Processes



What are Threads Good For?

- To maintain responsiveness of an application during a long running task
- To enable cancellation of separable tasks
- Some problems are intrinsically parallel
- To monitor status of some resource (e.g., DB)
- Some APIs and systems demand it (e.g., Swing)

Application Thread

- When we execute an application:
  1. The JVM creates a Thread object whose task is defined by the `main()` method
  2. The JVM starts the thread
  3. The thread executes the statements of the program one by one
  4. After executing all the statements, the method returns and the thread dies

Multiple Threads in an Application

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory, i.e., heap (are there variables on the heap?)
- Two different threads can act on the same object and same static fields concurrently

Creating Threads

- There are two ways to create our own **Thread** object
  1. Subclassing the **Thread** class and instantiating a new object of that class
  2. Implementing the **Runnable** interface
- In both cases the **run()** method should be implemented

Extending Thread

```
public class ThreadExample extends Thread {
    public void run () {
        for (int i = 1; i <= 100; i++) {
            System.out.println("---");
        }
    }
}
```

Thread Methods

void start()

- Creates a new thread and makes it runnable
- This method can be called only once

void run()

- The new thread begins its life inside this method

void stop() (deprecated)

- The thread is being terminated

Thread Methods

void yield()

- Causes the currently executing thread object to temporarily pause and allow other threads to execute
- Allow only threads of the same priority to run

void sleep(int *m*) or sleep(int *m*, int *n*)

- The thread sleeps for *m* milliseconds, plus *n* nanoseconds

Implementing Runnable

```
public class RunnableExample implements Runnable {
   public void run () {
      for (int i = 1; i <= 100; i++) {
            System.out.println ("***");
      }
   }
}
```

A Runnable Object

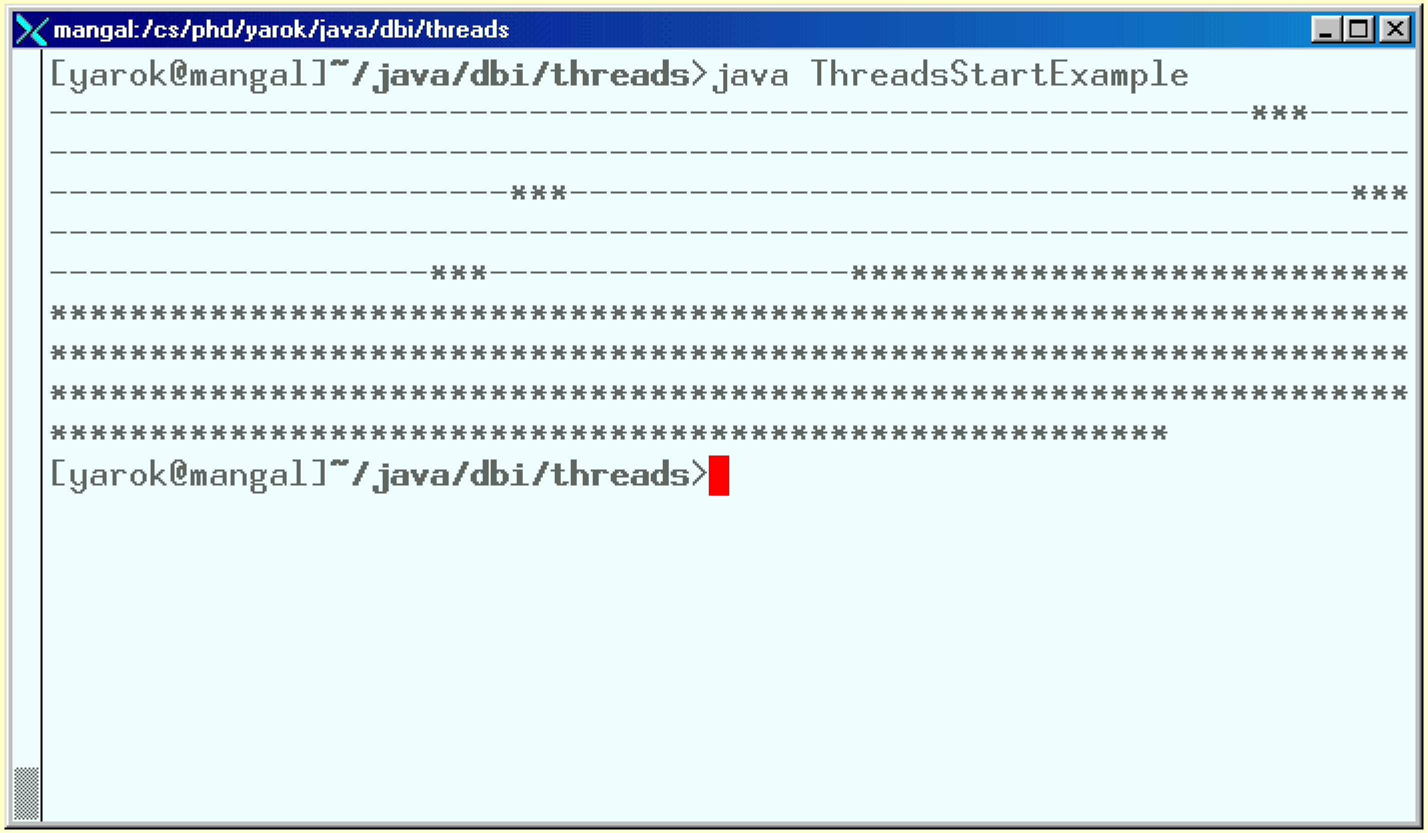
- When running the Runnable object, a Thread object is created from the Runnable object
- The Thread object's **run()** method calls the Runnable object's **run()** method
- Allows threads to run inside any object, regardless of inheritance

Example – an applet that is also a thread

Starting the Threads

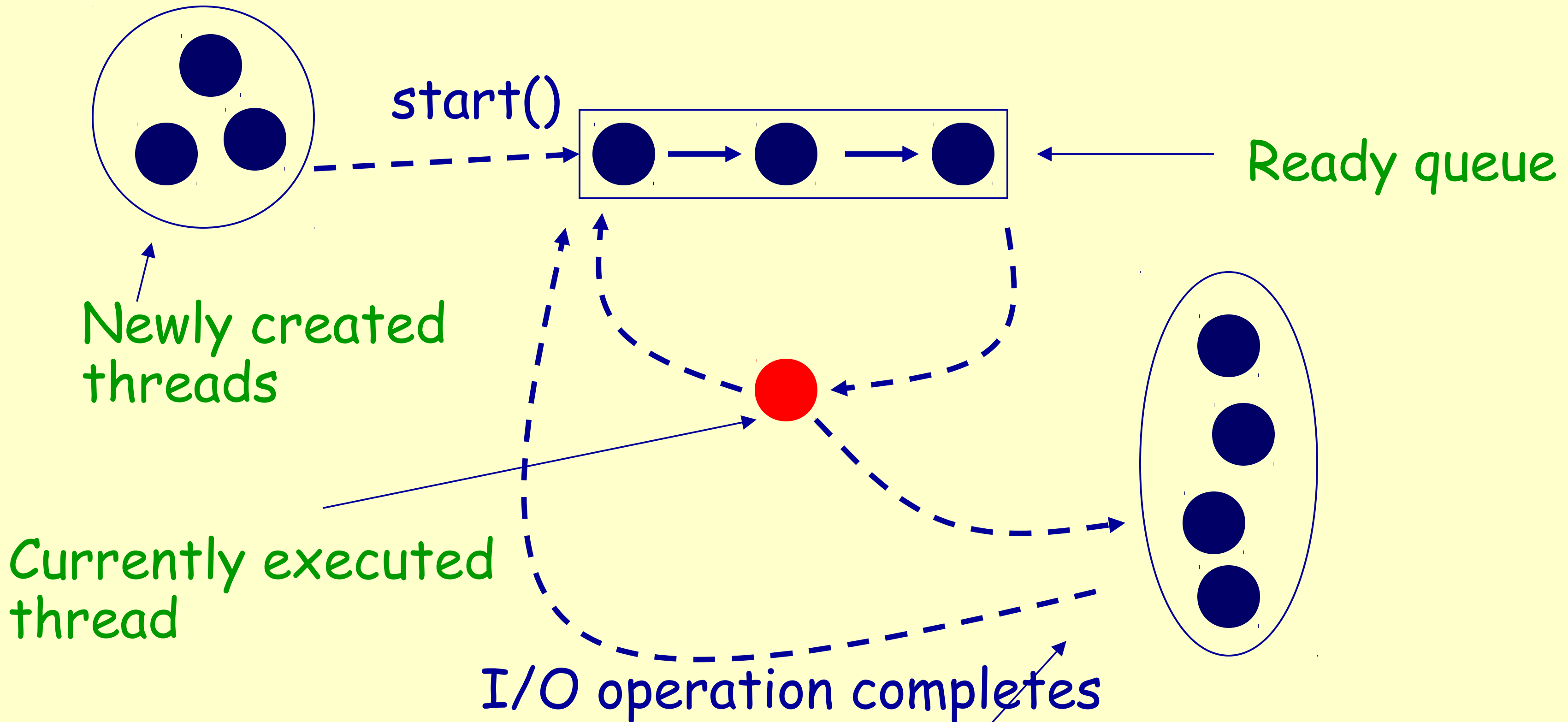
```
public class ThreadsStartExample {
       public static void main (String argv[]) {
         new ThreadExample ().start ();
         new Thread(new RunnableExample ()).start ();
       }
}
```

What will we see when running ThreadsStartExample?



```
[yarok@mangal]~/java/dbi/threads>java ThreadsStartExample
------------------------------------------------------------***-----
--------------------------------------------------------------------
-----------------------***---------------------------------------***
--------------------------------------------------------------------
-------------------***------------------****************************
********************************************************************
********************************************************************
********************************************************************
********************************************************
[yarok@mangal]~/java/dbi/threads>
```

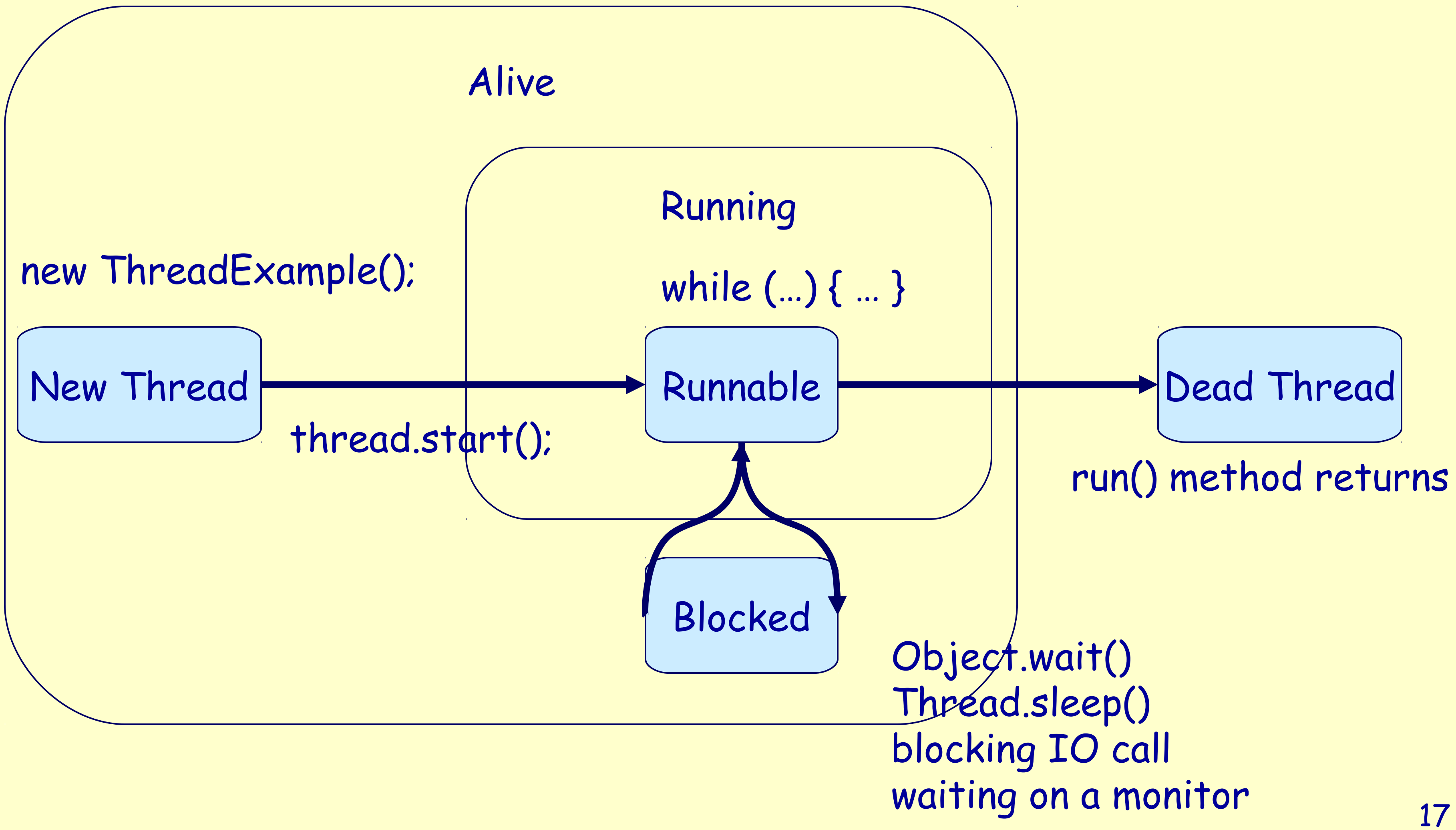
Scheduling Threads



What happens when a program with a ServerSocket calls accept()?

- Waiting for I/O operation to be completed
- Waiting to be notified
- Sleeping
- Waiting to enter a synchronized section

Thread State Diagram



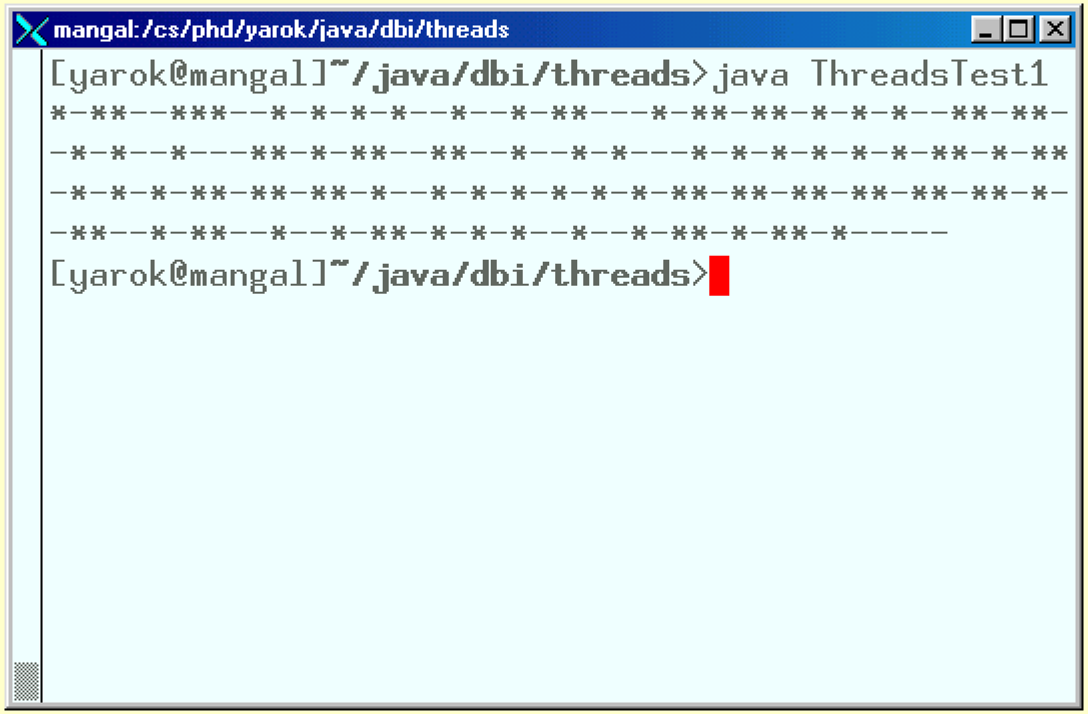
Example

```
public class PrintThread1 extends Thread {
    String name;
    public PrintThread1(String name) {
        this.name = name;
    }
    public void run() {
        for (int i=1; i<100 ; i++) {
            try {
                sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }
            System.out.print(name);
        }
}
```

Example (cont)

```
   public static void main(String args[]) {
        PrintThread1 a = new PrintThread1("*");
        PrintThread1 b = new PrintThread1("-");

        a.start();
        b.start();
   }
}
```



mangal:/cs/phd/yarok/java/dbi/threads

```
[yarok@mangal]~/java/dbi/threads>java ThreadsTest1
*-**--***--*-*-*-*--*--*-**---*-**-**-*-*-*--**-**-
-*-*--*---**-*-**--**--*--*-*---*-*-*-*-*-*-**-*-**
-*-*-*-**-**-**-*--*-*-*-*-*-*-**-**-**-**-**-**-*-
-**--*-**--*--*-**-*-*-*--*--*-**-*-**-*-----
[yarok@mangal]~/java/dbi/threads>
```

Scheduling

- Thread scheduling is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either preemptive or nonpreemptive

Preemptive Scheduling

- Preemptive scheduling – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
- Nonpreemptive scheduling – the scheduler never interrupts a running thread
- The nonpreemptive scheduler relies on the running thread to yield control of the CPU so that other threads may execute

Thread Priority

- Every thread has a priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority values range from 1 to 10, in increasing priority

Thread Priority (cont.)

- The priority can be adjusted subsequently using the **setPriority()** method
- The priority of a thread may be obtained using **getPriority()**
- Priority constants are defined:
  - MIN_PRIORITY=1
  - MAX_PRIORITY=10
  - NORM_PRIORITY=5

The **main** thread is created with priority NORM_PRIORITY

Daemon Threads

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
- The Java VM will not exit if non-Daemon threads are executing
- The Java VM will exit if only Daemon threads are executing
- Daemon threads die when the Java VM exits
- Q: Is the **main** thread a daemon thread?

Thread and the Garbage Collector

- Can a Thread object be collected by the garbage collector while running?
  - If not, why?
  - If yes, what happens to the execution thread?
- When can a Thread object be collected?

ThreadGroup

- The ThreadGroup class is used to create groups of similar threads. Why is this needed?

“Thread groups are best viewed as an unsuccessful experiment, and you may simply ignore their existence.”

Joshua Bloch, software architect at Sun