Concurrent Programming

Yih-Kuen Tsay

Dept. of Information Management

National Taiwan University

Processes and Threads

- Thread:

  The operating system abstraction of a processing activity; a sequence of steps executed one at a time; a single sequential flow of control; …

- Process:

  An executing program (application) that involves single or multiple processing activities

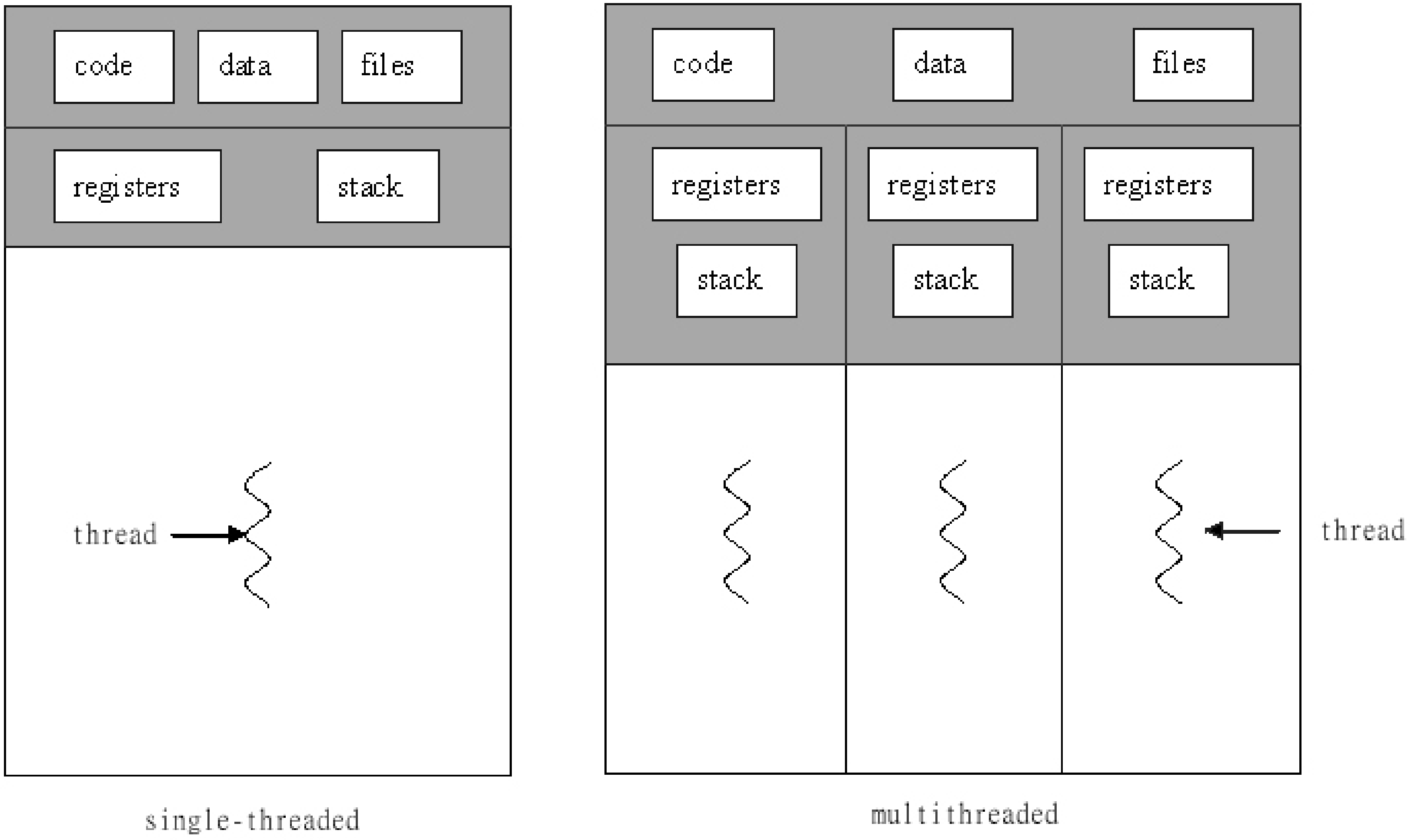
* single-threaded process

traditional UNIX-like process

* multi-threaded process

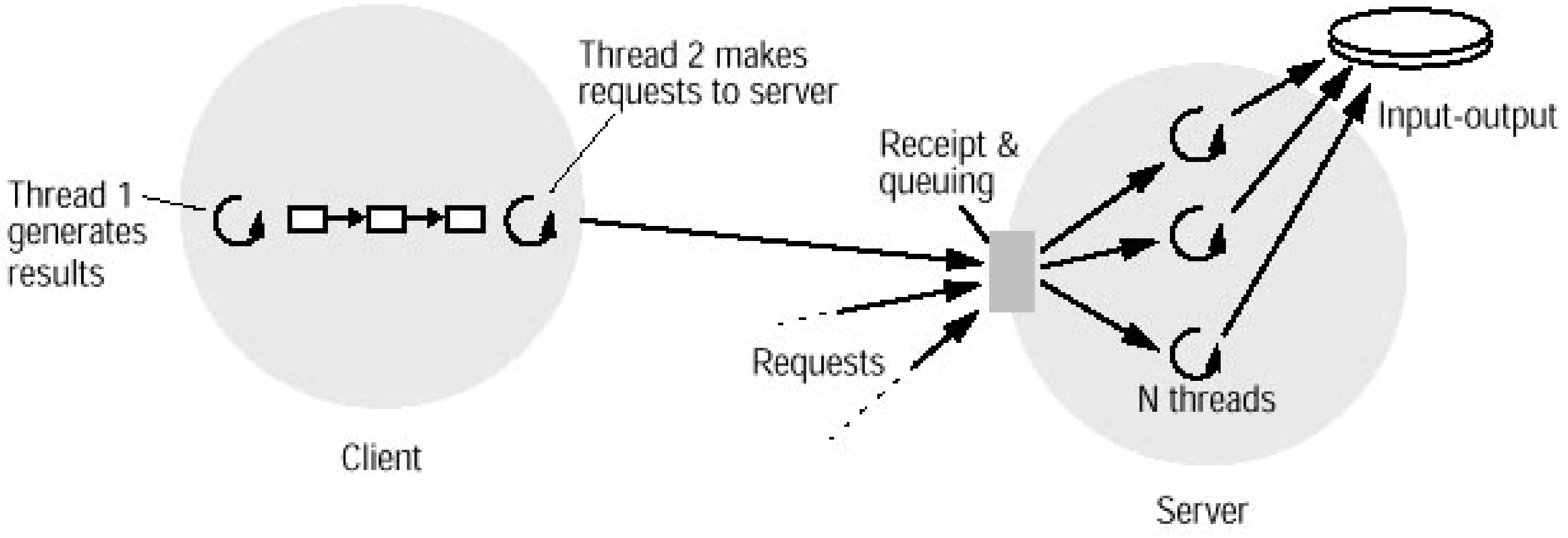
A *process* consists of an ***execution environment*** together with one or more ***threads***.

Single vs. Multi-Threaded Processes



Redrawn from: Silberschatz et al., *Operating System Concepts, Sixth Edition*

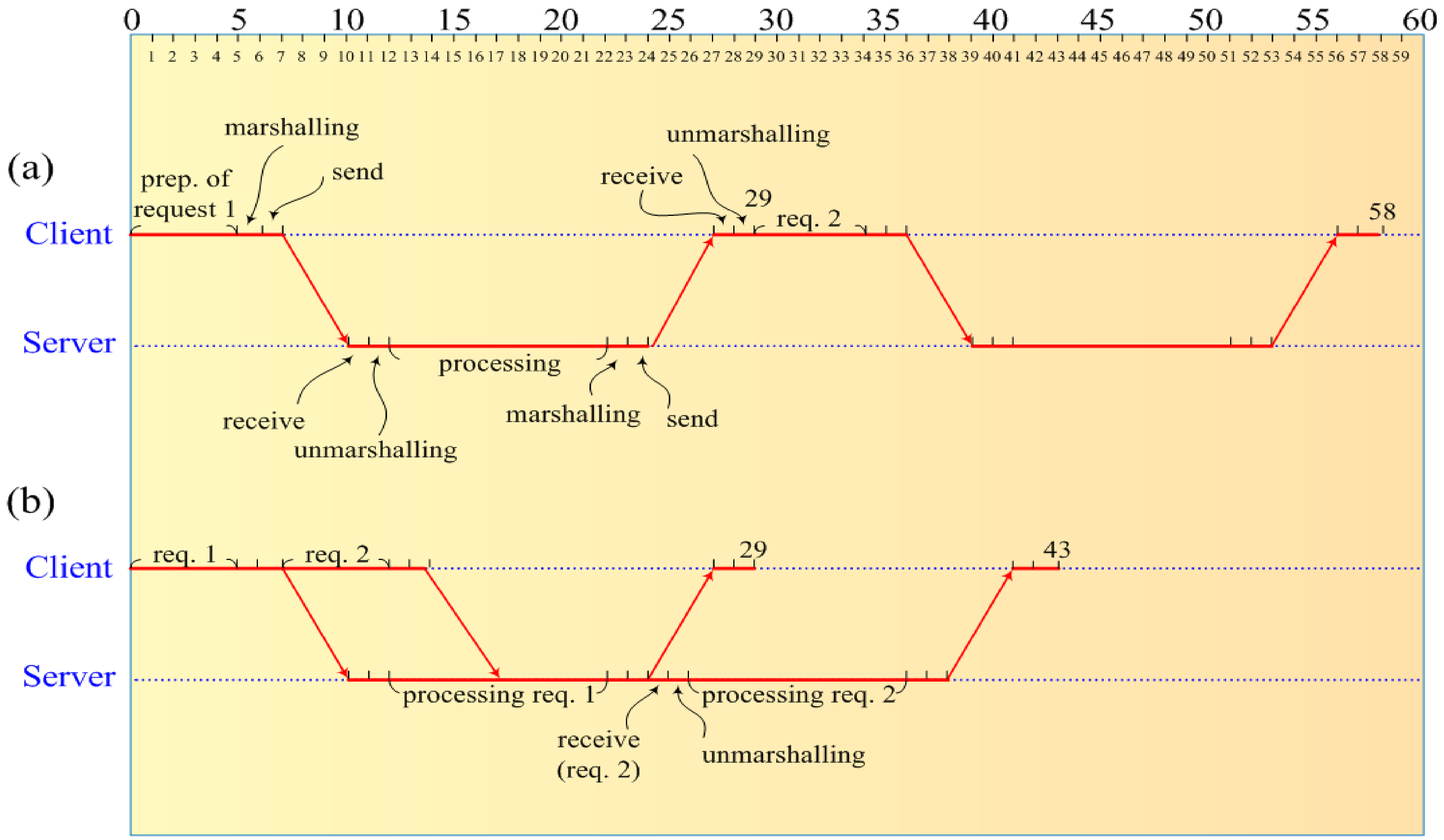
Client and Server with Threads



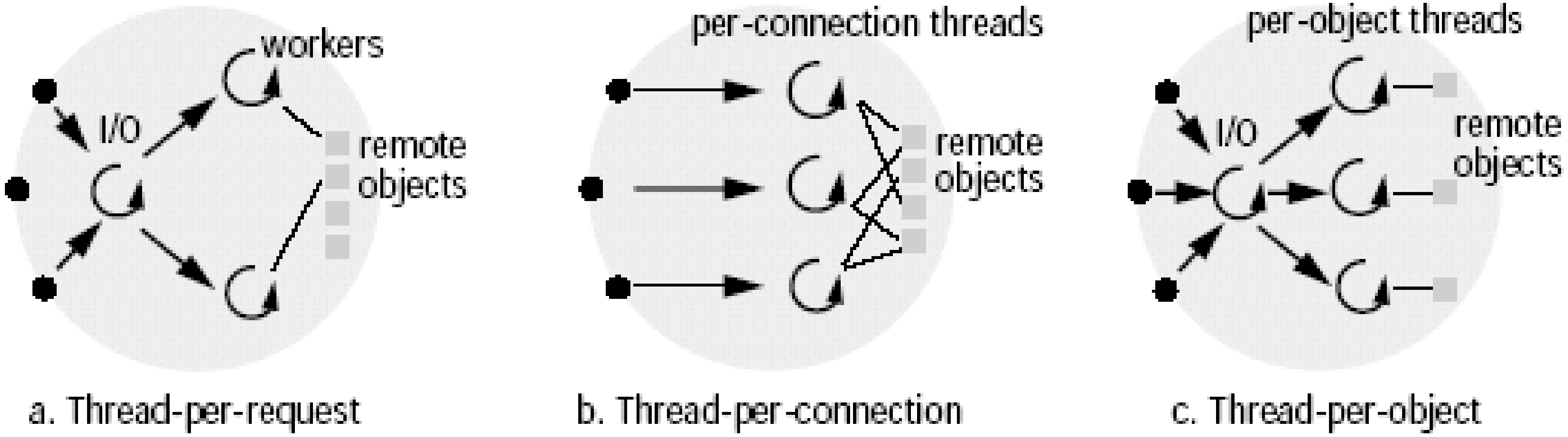
Note 1: recall the producer-consumer model.
Note 2: disk block caching helps increase the server's throughput.

Source: Coulouris et al., *Distributed Systems: Concepts and Design, Fourth Edition*

Performance Gain with Threads



Other Threading Architectures



Note: we are looking at the server side.

Source: Coulouris et al., *Distributed Systems: Concepts and Design, Fourth Edition*

Summary: When Are Multiple Threads Useful?

- Multiple physical devices, such as CPU and I/O device, should be present (possibly at different locations).
- The different processing activities, i.e., threads, use different physical devices some of the time.
- There is a need of prioritizing the different processing activities.

Execution Environments

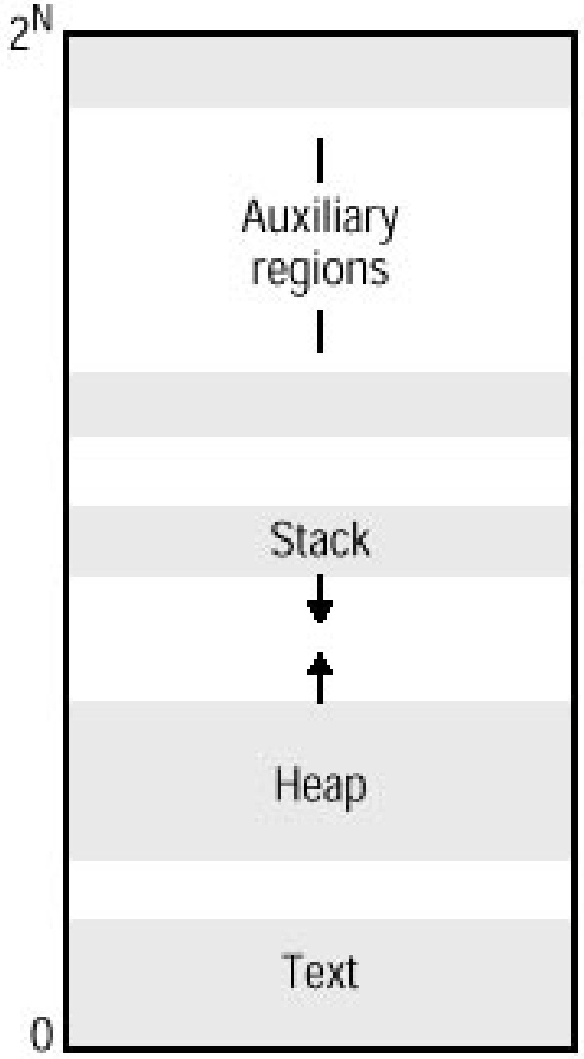
An *execution environment* is a collection of kernel-managed resources:

- an address space
- synchronization and communication resources (semaphores, sockets, etc.)
- higher-level resources (files, windows, etc.)

An environment is normally expensive to create and manage, but can be shared by several threads.

It is protected (from threads residing in other execution environments).

Address Space



* The auxiliary regions are for **mapped files** and **stacks of additional threads** etc.

Source: Coulouris et al., *Distributed Systems: Concepts and Design, Fourth Edition*

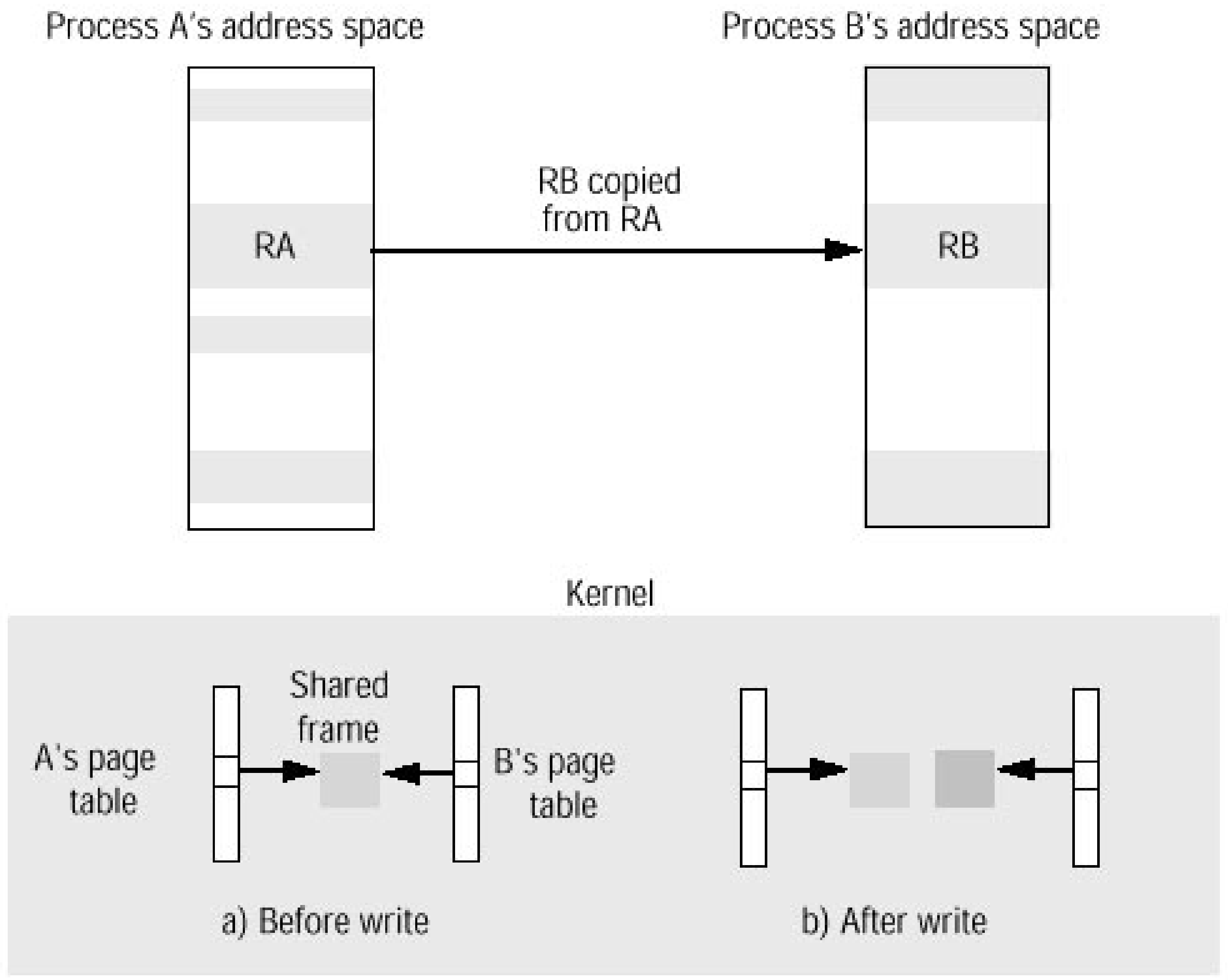
Shared Regions

A *shared region* is one that is backed by the same physical memory as one or more regions belonging to other address spaces.

The uses of shared region include:

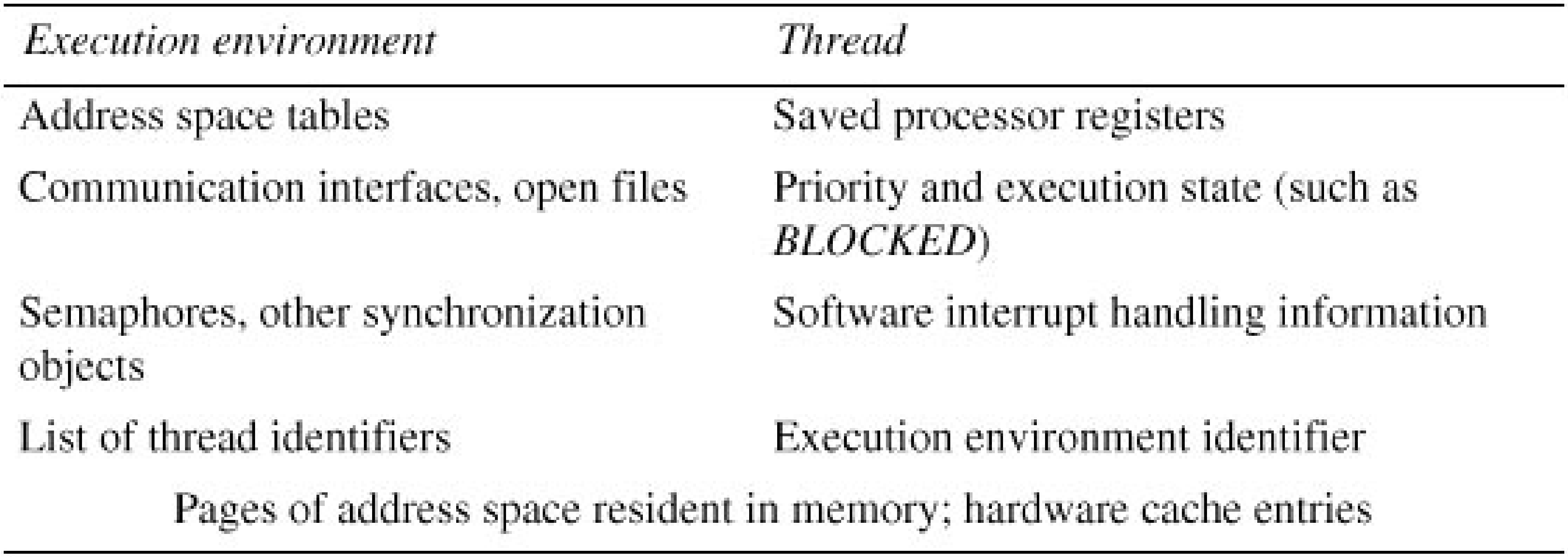
- Libraries
- Kernel
- Data sharing and communication

Copy on Write



Source: Coulouris et al., *Distributed Systems: Concepts and Design, Fourth Edition*

An Execution Environment and Its Threads



Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Source: Coulouris et al., *Distributed Systems: Concepts and Design, Fourth Edition*

Multiple Threads vs. Multiple Processes

Categories of comparison:

- Creation cost
- Scheduling and context switching cost
- Convenience of data and resources sharing
- Protection

Thread Scheduling

- **Preemptive**: A thread may be suspended to make way for another thread, even when it is otherwise runnable.

  Advantage: suitable for real-time applications, when combined with an appropriate priority scheme.
- **Non-preemptive**: A thread runs until it makes a call that causes it to be descheduled and another to be run.

  Advantage: process synchronization is simpler.

Essential Concepts in Concurrent (Threads) Programming

- Race condition
- Critical section (region)
- Monitor
- Condition variable
- Semaphore

A Multithreaded Java Program

```
public class PingPong extends Thread {
    private String word;    // what word to print
    private int delay;      // how long to pause
    public PingPong(String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
    public void run() {
        try {   for (;;) {
                    System.out.print(word + " ");
                    Thread.sleep(delay);  // wait until next time
                }
        } catch (InterruptedException e) {
                return;             // end this thread
        }
    }
    public static void main(String[] args) {
        new PingPong("ping", 33).start();        // 1/30 second
        new PingPong("PONG", 100).start();       // 1/10 second
    }
}
```

Threads and Runnables

- Threads abstract the concept of a worker.
  - A worker is an entity that gets something done.
  - The work done by a thread is packaged up in its run method.
- The Runnable interface abstracts the concept of work and allows that work to be associated with a worker.

Threads and Runnables (cont.)

```
class RunPingPong implements Runnable {
    private String word;    // what word to print
    private int delay;      // how long to pause
    RunPingPong(String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
    public void run() {
        try {     for (;;) {
                      System.out.print(word + " ");
                      Thread.sleep(delay);  // wait until next time
                  }
        } catch (InterruptedException e) {
                  return;            // end this thread
        }
    }
    public static void main(String[] args) {
        Runnable ping = new RunPingPong("ping", 33);
        Runnable pong = new RunPingPong("PONG", 100);
        new Thread(ping).start();
        new Thread(pong).start();
    }
}
```

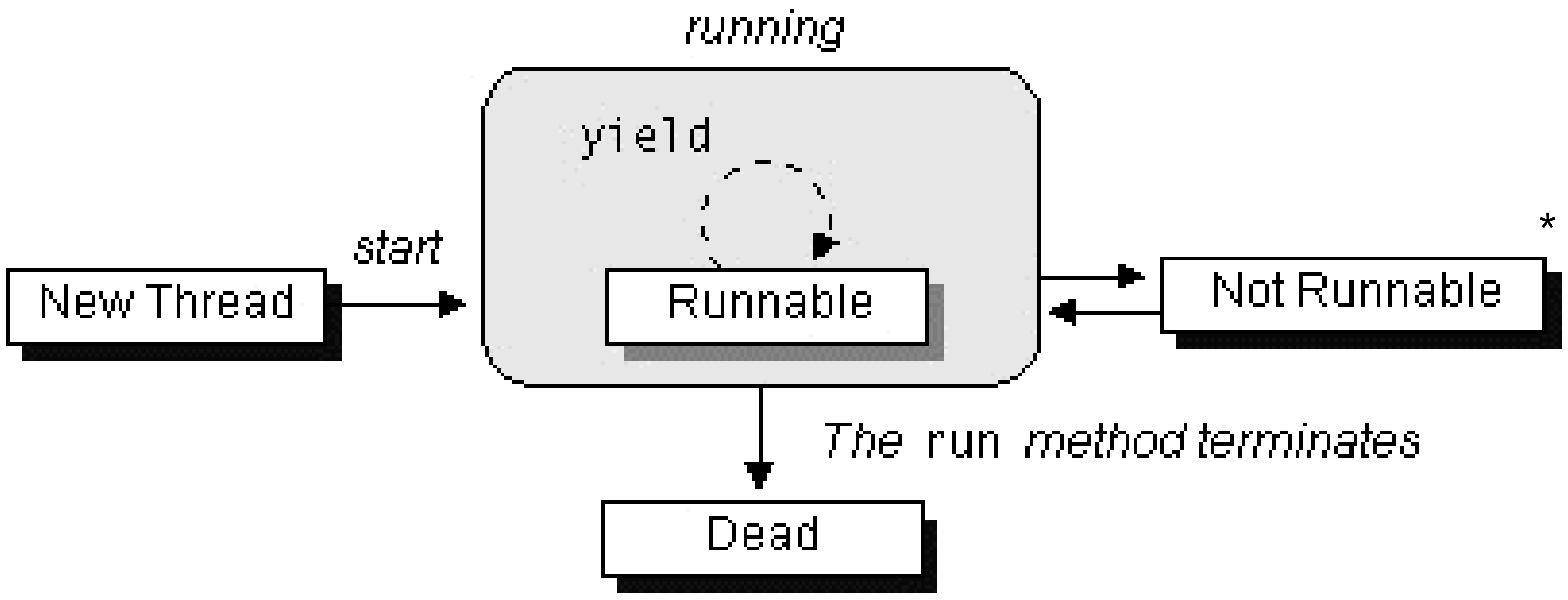
A Print Server

```
class PrintServer implements Runnable {
    private Queue requests = new Queue();
    public PrintServer() {
        new Thread(this).start();
    }
    public void print(PrintJob job) {
        requests.add(job);
    }
    public void run() {
        for (;;)
            realPrint((PrintJob) requests.take());
    }
    private void realPrint(Printjob job) {
        // do the real work of printing
    }
}
```

Improved Print Server

```
class PrintServer2 {
    private Queue requests = new Queue();
    public PrintServer2() {
        Runnable service = new Runnable() {
                public void run() {
                        for (;;)
                                realPrint((PrintJob)requests.take());
                }
        };
        new Thread(service).start();
    }
    public void print(PrintJob job) {
        requests.add(job);
    }
    private void realPrint(PrintJob job) {
        // do the real work of printing
    }
}
```

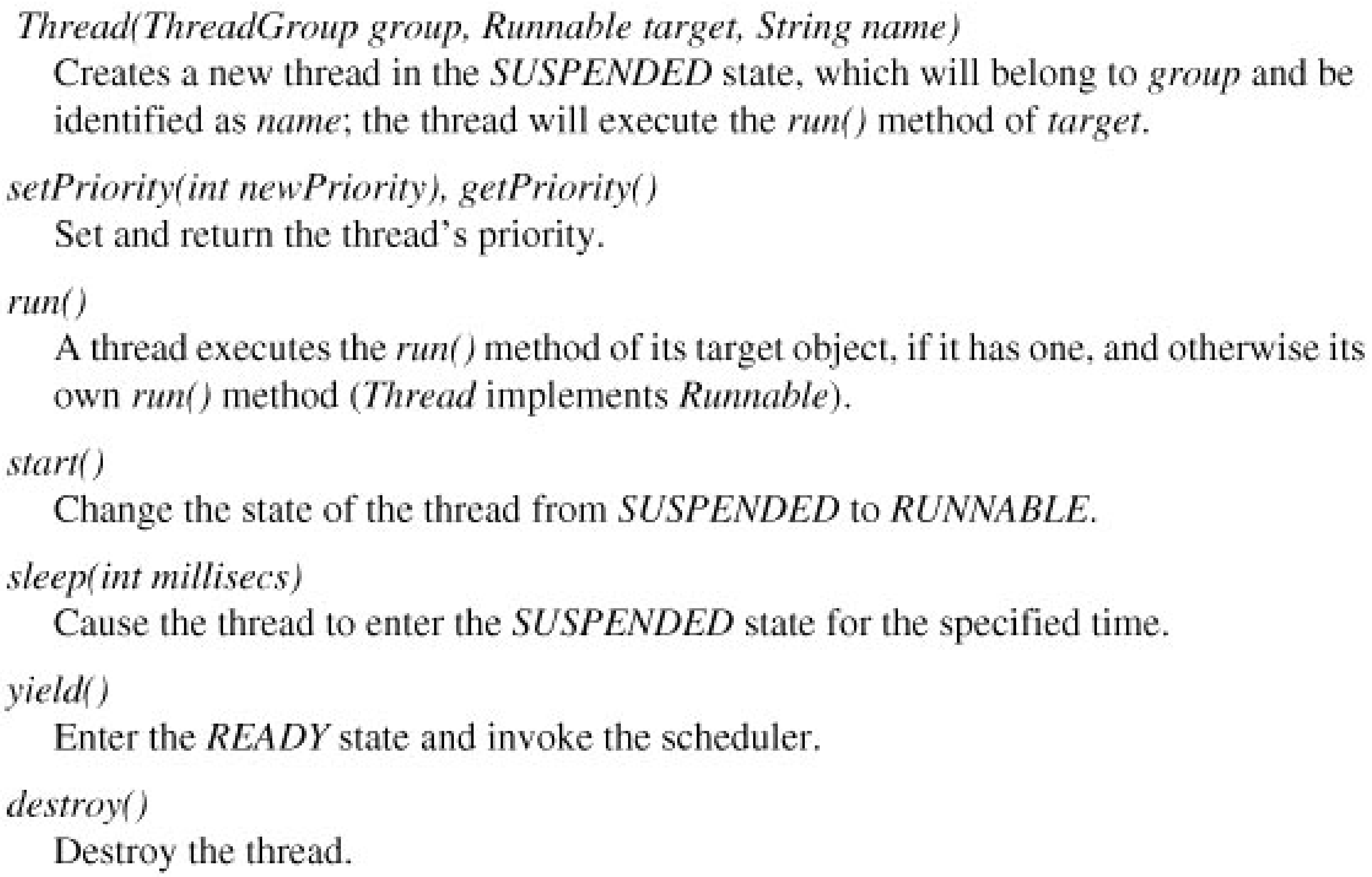
Thread States



* A thread becomes not runnable when (1) its sleep or suspend method is invoked, (2) it invokes its wait, or (3) it is blocking on I/O.

Source: Sun Microsystems, Inc., *The Java Tutorial*

Java Thread's Methods (partial)



Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Set and return the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
Enter the *READY* state and invoke the scheduler.

destroy()
Destroy the thread.

Source: Coulouris et al., *Distributed Systems: Concepts and Design, Fourth Edition*

More about Thread's Methods

- run():

  If this thread was constructed using a separate Runnable run object, then calls that Runnable object's run method; otherwise, does nothing.

- start():

  Begins the execution of this thread; the Java Virtual Machine calls the run method of this thread.

- sleep(long millis):

  Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

- yield():

  Causes the currently executing thread object to temporarily pause and allow other threads to execute.

- destroy():

  Destroys this thread, without any cleanup.

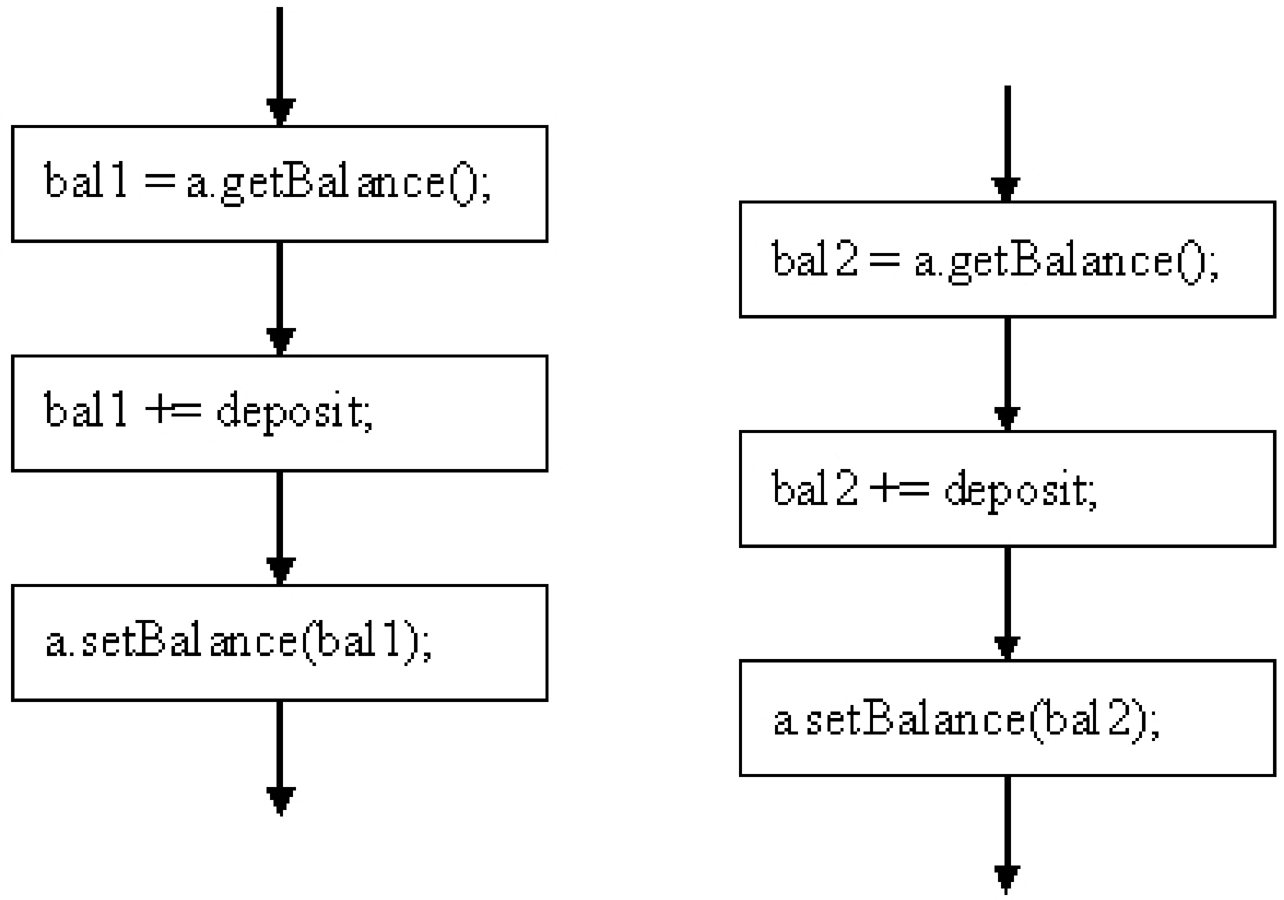
Using yield() (Part I)

```
class Babble extends Thread {
    static boolean doYield;       // yield to other threads?
    static int howOften;          // how many times to print

    private String word;          // my word

    Babble(String whatToSay) {
        word = whatToSay;
    }

    public void run() {
        for (int i = 0; i < howOften; i++) {
            System.out.println(word);
            if (doYield)
                yield(); // give another thread a chance
        }
    }
```

Using yield() (Part II)

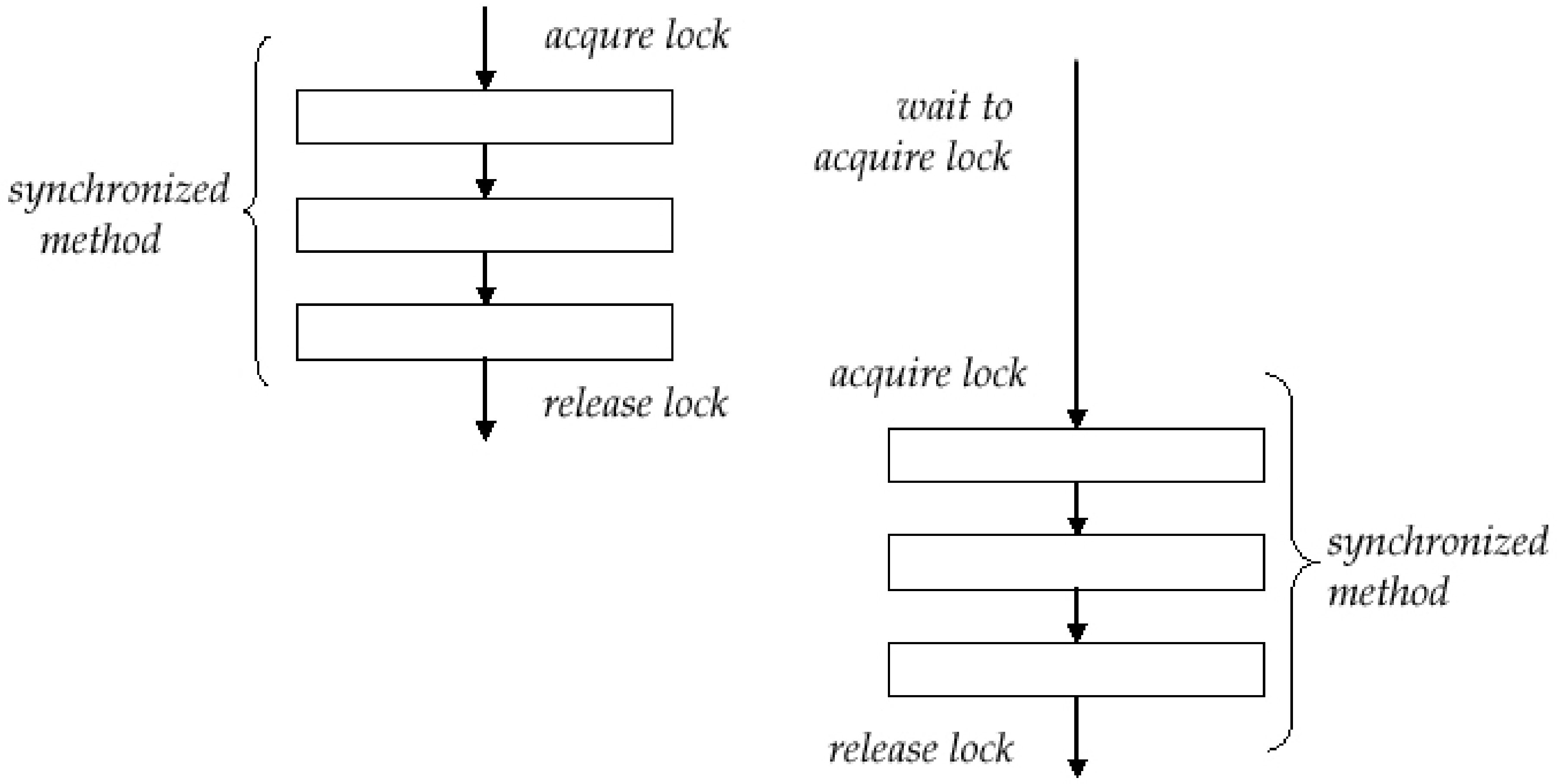
```
    public static void main(String[] args) {
        doYield = new Boolean(args[0]).booleanValue();
        howOften = Integer.parseInt(args[1]);

        // create a thread for each word
        for (int i = 2; i < args.length; i++)
                new Babble(args[i]).start();
    }
}
```

A Race Condition



Source: Arnold, et al., *The Java Programming Language, Third Edition*

Synchronization



Source: Arnold, et al., *The Java Programming Language, Third Edition*

Synchronized Methods

```
class BankAccount {
    private long number;
    private long balance;
    public BankAccount(long initialDeposit) {
        balance = initialDeposit;
    }
    public synchronized long getBalance() {
        return balance;
    }
    public synchronized long deposit(long amount) {
        balance += amount;
    }
}
```

Synchronized Statements

```
/** make all elements in the array non-negative */
public static void abs(int[] values) {
   synchronized (values) {
        for (int i = 0; i < values.length; i++) {
              if (values[i] < 0)
                    values[i] = - values[i];
        }
   }
}
```

Synchronized Methods vs. Statements

```
public synchronized long deposit(long
  amount) {
      balance += amount;
   }
```

is in effect the same as

```
public long deposit(long amount) {
  synchronized (this) {
      balance += amount;
   }
```

Granularity of Synchronization (Part I)

```
class SeparateGroups {
    private double aVal = 0.0;
    private double bVal = 1.1;
    protected Object lockA = new Object();
    protected Object lockB = new Object();

    public double getA() {
        synchronized (lockA) {
            return aVal;
        }
    }
    public void setA(double val) {
        synchronized (lockA) {
            aVal = val;
        }
    }
```

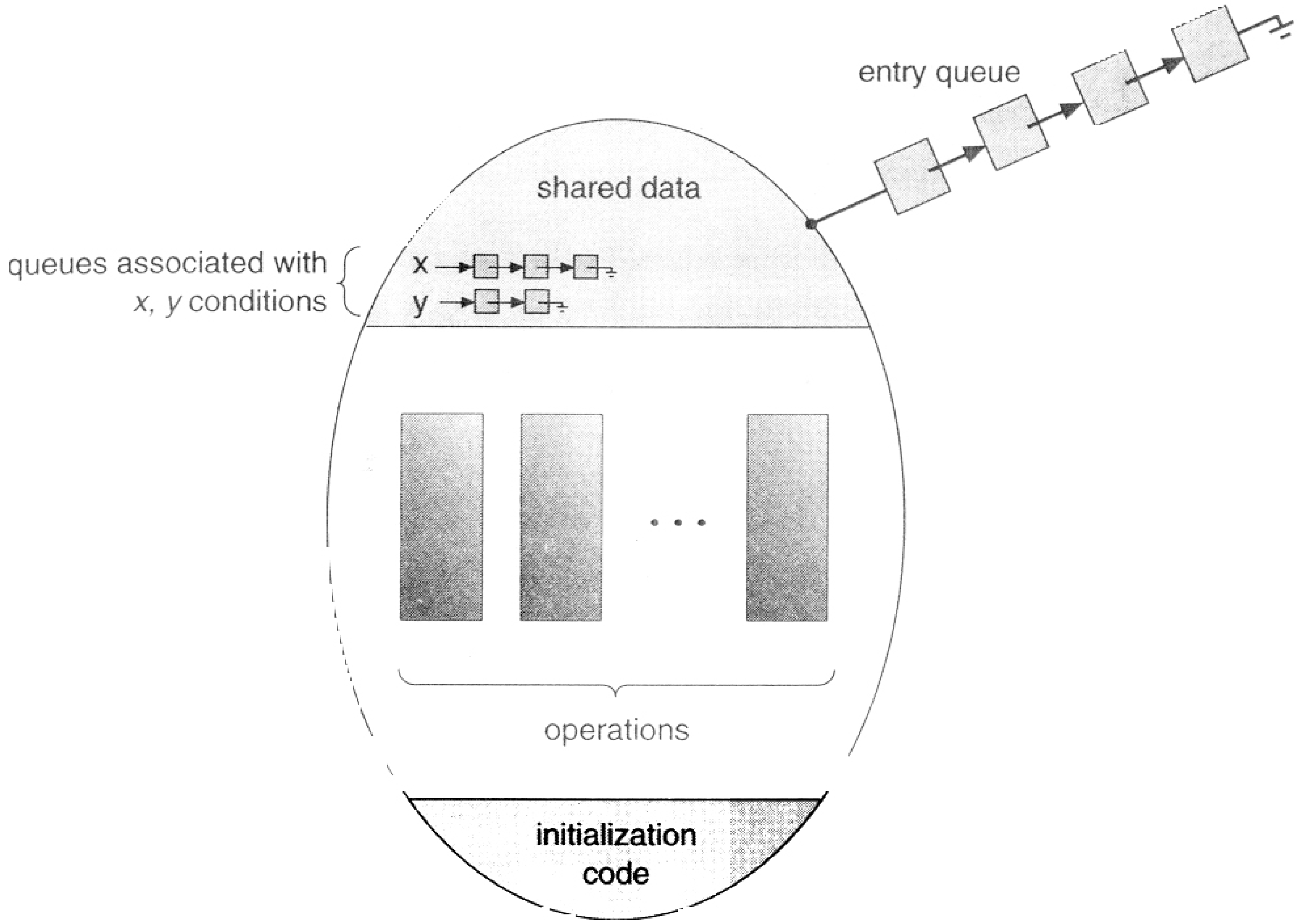
Granularity of Synchronization (Part II)

```
public double getB() {
      synchronized (lockB) {
            return bVal;
      } }
public void setB(double val) {
      synchronized (lockB) {
            bVal = val;
      } }
public void reset() {
      synchronized (lockA) {
            synchronized (lockB) {
                  aVal = bVal = 0.0;
            }
      } }
```

Synchronizing on an Enclosing Object

```
public class Outer {
    private int data;
    // ...
    private class Inner{
        void setOuterData() {
            synchronized (Outer.this) {
                data = 12;
            }
        }
    }
}
```

Monitor: A Synchronization Abstraction



Source: Silberschatz et al., *Operating System Concepts, Sixth Edition*

wait and notify

```
synchronized void doWhenCondition() {
    while (!condition)
        wait();
    ... Do what must be done when the condition is
    true ...
}

synchronized void changeCondition() {
    ... change some value used in a condition test ...
    notifyAll();  // or notify()
}
```

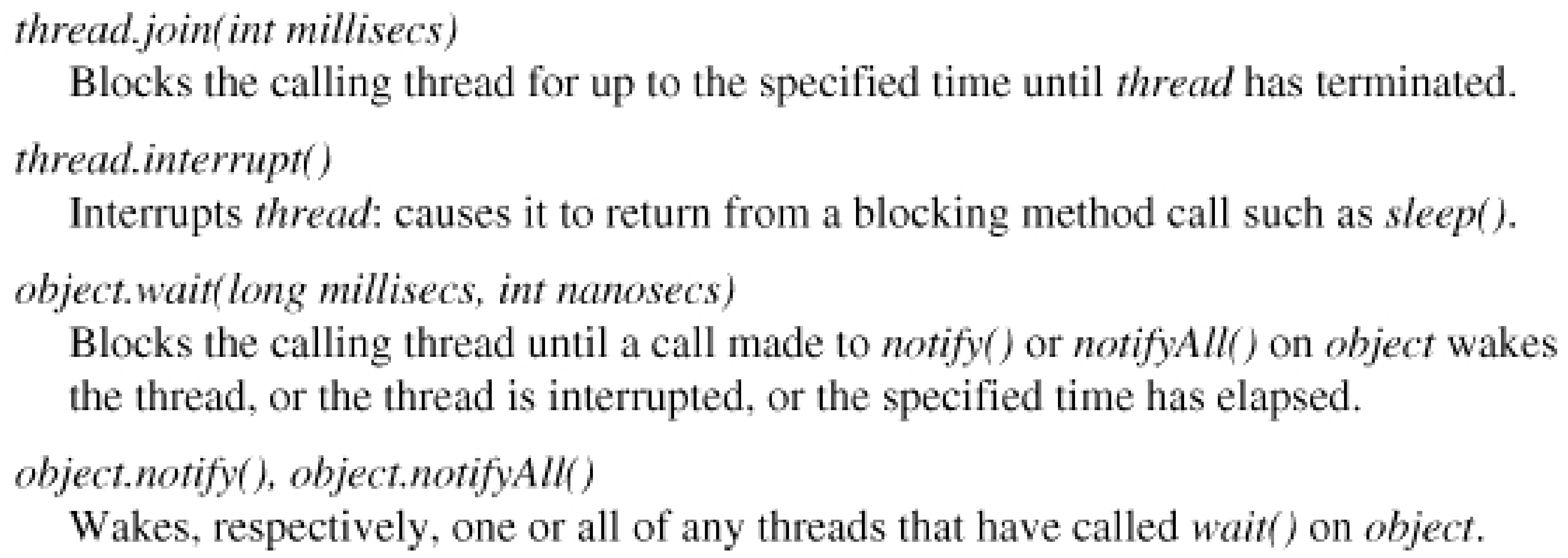
Implementation of a Shared Queue (Part I)

```
class Queue {
        // the first and last elements in the queue
   private Cell head, tail;

   public synchronized void add(Object o) {
        Cell p = new Cell(o);  // wrap o in a cell
        if (tail == null)
                head = p;
        else
                tail.next = p;
        p.next = null;
        tail = p;
        notifyAll();  // let waiters know something arrived}
```

Implementation of a Shared Queue (Part II)

```
  public synchronized Object take() throws
        InterruptedException
  {
        while (head == null)
              wait();           // wait for an element

        Cell p = head;          // remember first element
        head = head.next;       // remove it from the queue
        if (head == null)       // check for an empty queue
              tail = null;
        return p.item;
  }
}
```

Java Thread Synchronization Methods (partial)



thread.join(int millisecs)
Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()
Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)
Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), object.notifyAll()
Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Source: Coulouris et al., *Distributed Systems: Concepts and Design, Fourth Edition*

Using join() (Part I)

```
class CalcThread extends Thread {
    private double result;
    public void run() {
        result = calculate();
    }
    public double getResult() {
        return result;
    }
    public double calculate() {
        // … calculate a value for “result”
    }
}
```

Using join() (Part II)

```
class ShowJoin {
    public static void main(String[] args) {
        CalcThread calc = new CalcThread();
        calc.start();
        doSomethingElse();
        try {
            calc.join();
            System.out.println("result is "
                + calc.getResult());
        } catch (InterruptedException e) {
            System.out.println("No answer: interrupted");
        }
    }
}
```

Synchronization Designs

- Server-side Synchronization
  - A shared object protects access to itself by making its methods synchronized.
  - Generally better …
- Client-side Synchronization
  - All clients of a shared object agree to synchronize on that object (or some other associated object) before manipulating it.
  - More flexible for method combinations and operations on multiple objects …

Deadlocks

- Several threads may reach a state where each thread is waiting for some other thread to release a lock.
- The programmer is fully responsible for avoiding deadlocks.

An Example Deadlock (Part I)

```
class Friendly {
    private Friendly partner;
    private String name;
    public Friendly(String name) {
        this.name = name;
    }
    public synchronized void hug() {
        System.out.println(Thread.currentThread().getName() +
                " in " + name + ".hug() trying to invoke " +
                partner.name + ".hugBack()");
        partner.hugBack();
    }
    private synchronized void hugBack() {
        System.out.println(Thread.currentThread().getName() +
                " in " + name + ".hugBack()");
    }
    public void becomeFriend(Friendly partner) {
        this.partner = partner;
    }
}
```

An Example Deadlock (Part II)

```
public static void main(String[] args) {
   final Friendly jareth = new Friendly(“jareth”);
   final Friendly cory = new Friendly(“cory”);
   jareth.becomeFriend(cory);
   cory.becomeFriend(jareth);

   new Thread(new Runnable() {
        public void run() { jareth.hug(); }
        }, “Thread1”).start();

   new Thread(new Runnable() {
        public void run() { cory.hug(); }
        }, “Thread2”).start();
}
```

Thread Groups

- Thread groups provide a way to manipulate threads collectively.
- A thread group is a set of threads and thread groups.
- Every thread belongs to exactly one thread group.
- Thread groups form a tree with the "system thread group" at the root.

Using ThreadGroup's Methods

```
public class EnumerateTest {
    public void listCurrentThreads() {
        ThreadGroup currentGroup =
                    Thread.currentThread().getThreadGroup();
        int numThreads = currentGroup.activeCount();
        Thread[] listOfThreads = new
                                     Thread[numThreads];
        currentGroup.enumerate(listOfThreads);
        for (int i = 0; i < numThreads; i++)
            System.out.println("Thread #" + i + " = " +
                                   listOfThreads[i].getName());
    }
}
```

Using ThreadGroup's Methods (cont.)

```
public class MaxPriorityTest {
    public static void main(String[] args) {
        ThreadGroup groupNORM = new ThreadGroup(
                                         "A group with normal priority");
        Thread priorityMAX = new Thread(groupNORM,
                                    "A thread with maximum priority");
        // set Thread's priority to max (10)
        priorityMAX.setPriority(Thread.MAX_PRIORITY);
        // set ThreadGroup's max priority to normal (5)
        groupNORM.setMaxPriority(Thread.NORM_PRIORITY);
        System.out.println("Group's maximum priority = " +
                                groupNORM.getMaxPriority());
        System.out.println("Thread's priority = " +
                               priorityMAX.getPriority());
    }
}
```