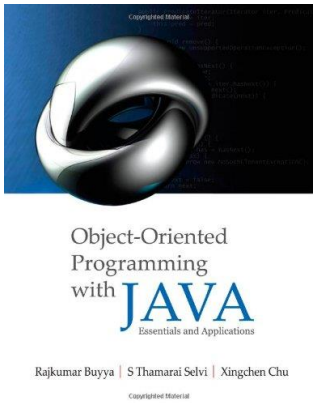
OS Support for Building Distributed Applications: Multithreaded Programming using Java Threads



Dr. Rajkumar Buyya

Cloud Computing and **D**istributed **S**ystems (CLOUDS) Laboratory

School of Computing and Information Systems

The University of Melbourne, Australia

http://www.buyya.com

Agenda

- Introduction
- Thread Applications
- Defining Threads
- Java Threads and States
- Examples

Introduction

- In a networked world, it is common practice to share resources among multiple users.
- Even on desktop computers, users typically run multiple applications and carry out some operations in the background (e.g., printing) and some in the foreground (e.g., editing) simultaneously.
- Modern programming languages and operating systems are designed to support the development of applications containing multiple activities that can be executed concurrently
- Preemptive (processor responsibility) vs. cooperative (process responsibility) multitasking?
- **Since the processor is switching between the applications at intervals of milliseconds, you feel that all applications run concurrently**

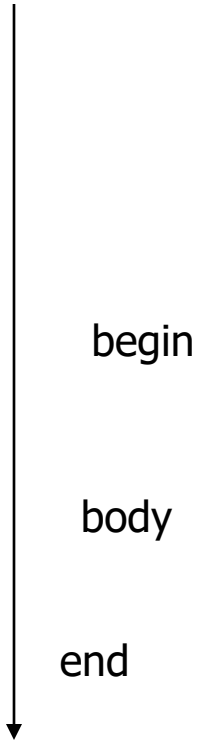
Threaded Applications

- Modern Systems
  - Multiple applications run concurrently!
  - This means that… there are multiple processes on your computer

Multitasking

A single threaded program

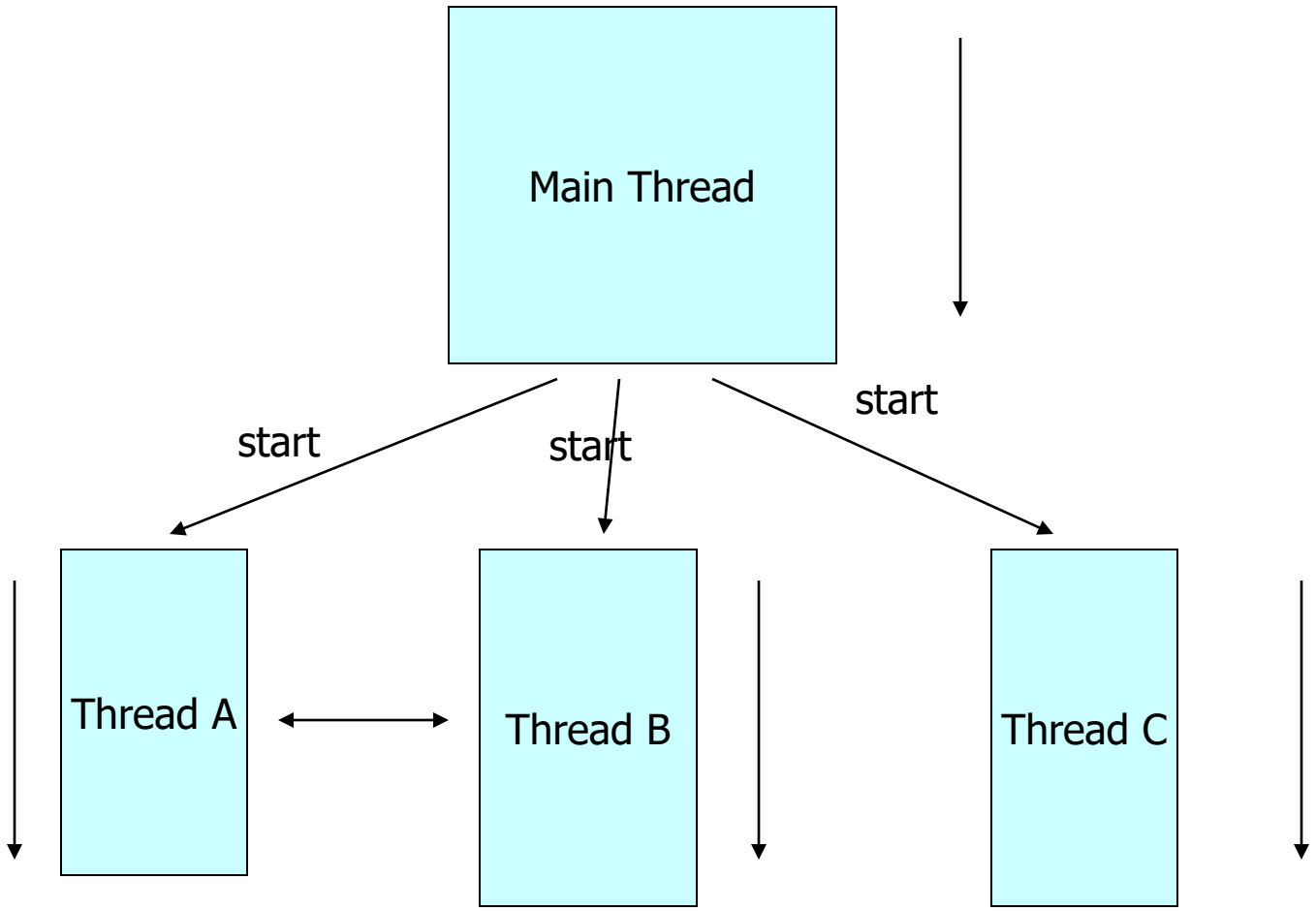
```
class ABC
{
....
    public void main(..)
    {
    ...
    ..
    }
}
```



Threaded Applications

- Modern Systems
  - Applications perform many tasks at once!
  - This means that… there are multiple threads within a single process.

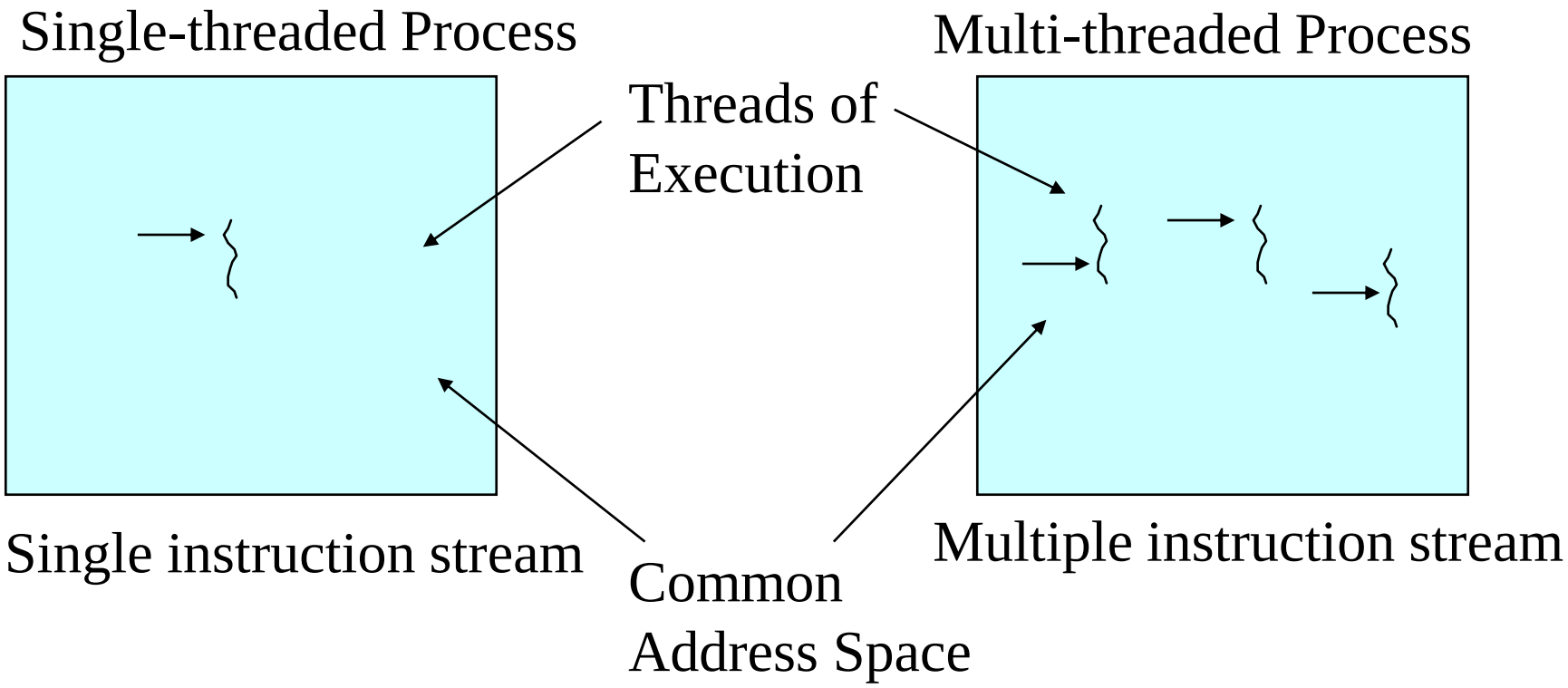
A Multithreaded Program



Threads may switch or exchange data/results

Single and Multithreaded Processes

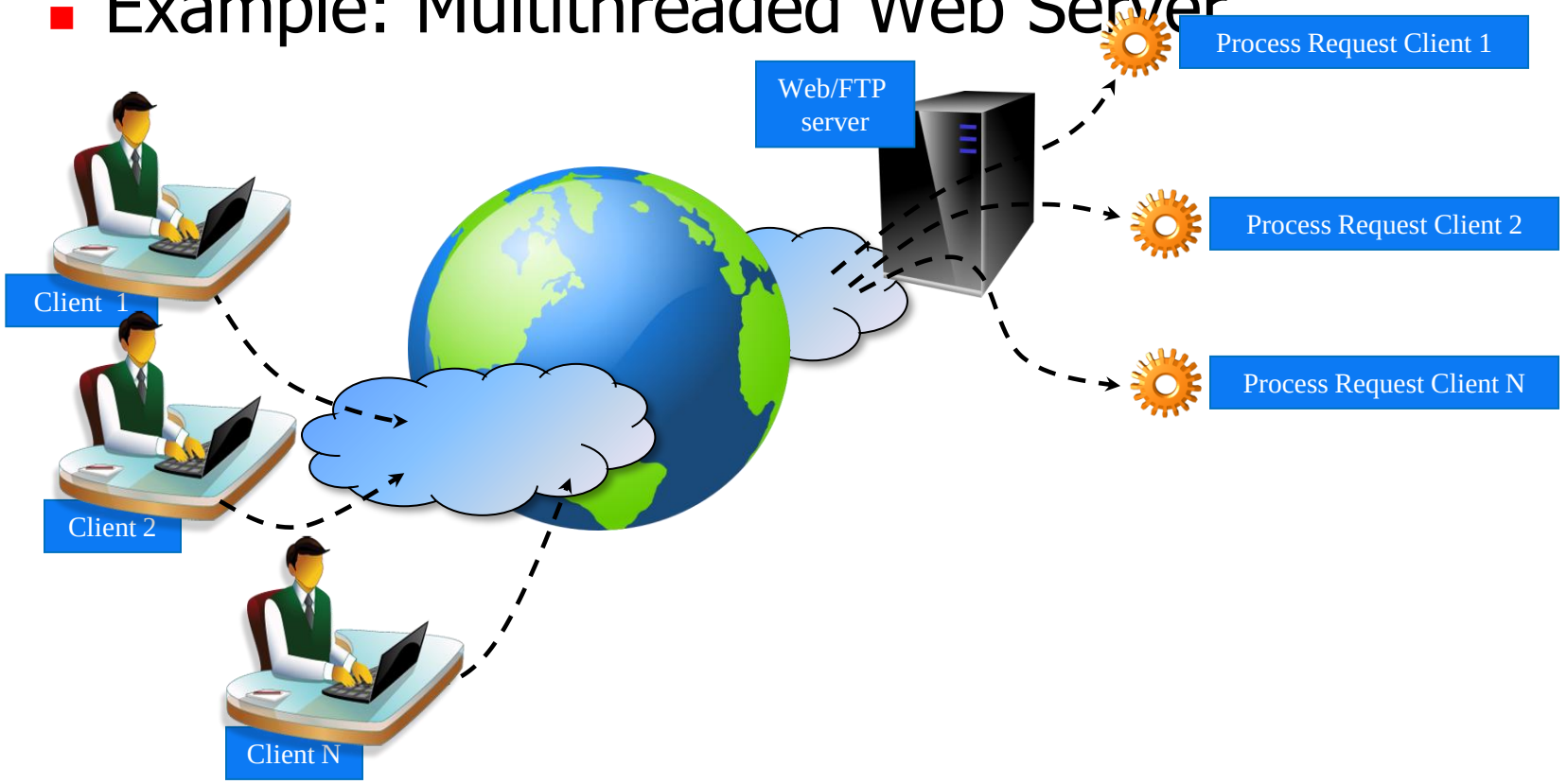
threads are light-weight processes within a process



- A process?
- A thread?
- Process-based Multi-tasking vs. thread-based multitasking?

Multithreaded Server: For Serving Multiple Clients Concurrently

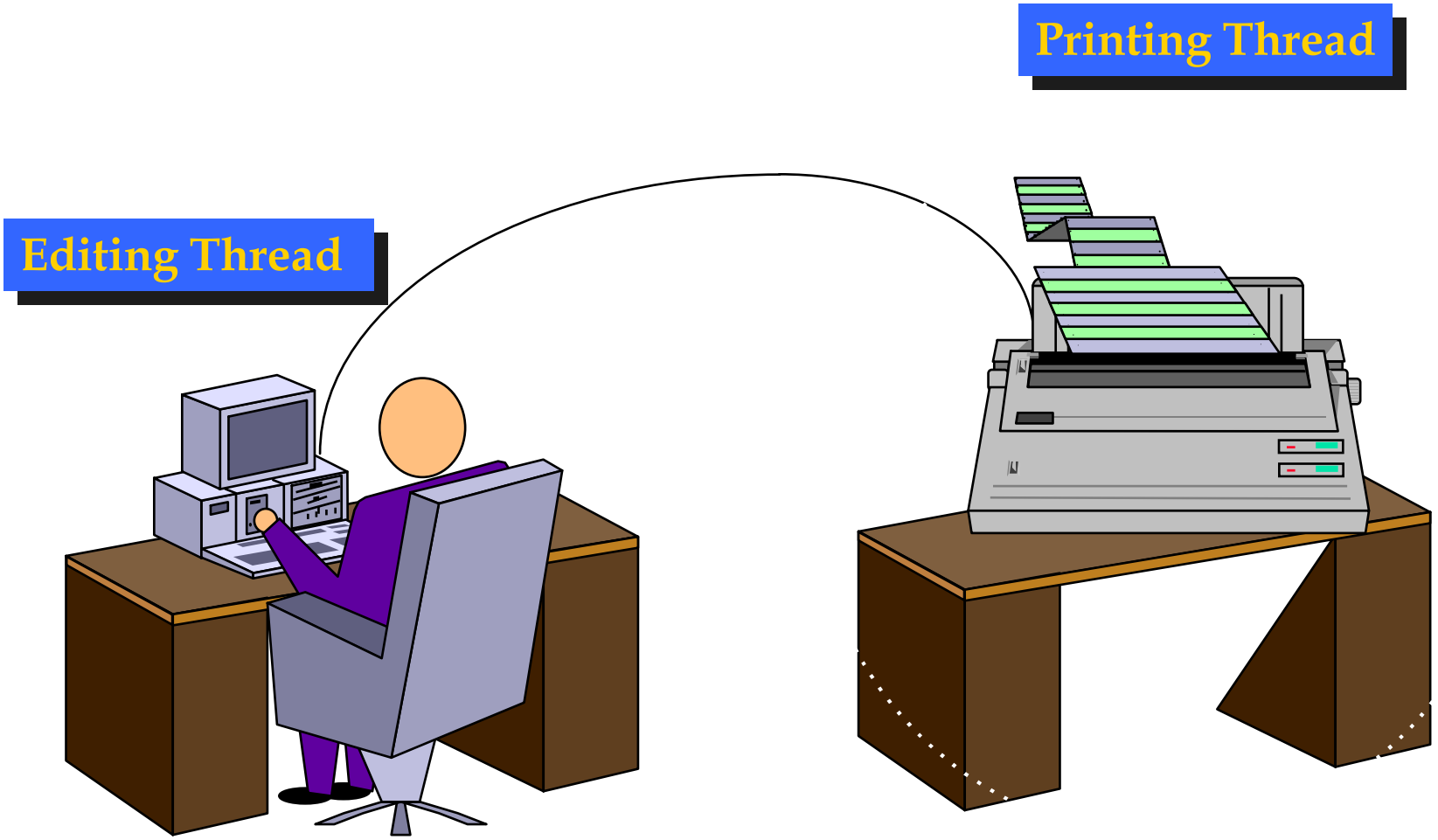
- Modern Applications
  - Example: Multithreaded Web Server



Threaded Applications

- Modern Applications
  - Example: Internet Browser + Youtube

Modern Applications need Threads (ex1): Editing and Printing documents in background.



A Process

- A process consists of **an execution environment** together with one or more threads
- An execution environment is a collection of local kernel-managed resources to which its threads have access.
- An execution environment includes:
  - An address space.
  - Thread synchronization and communication resources (e.g., semaphores, sockets).
  - Higher-level resources such as open files and windows.

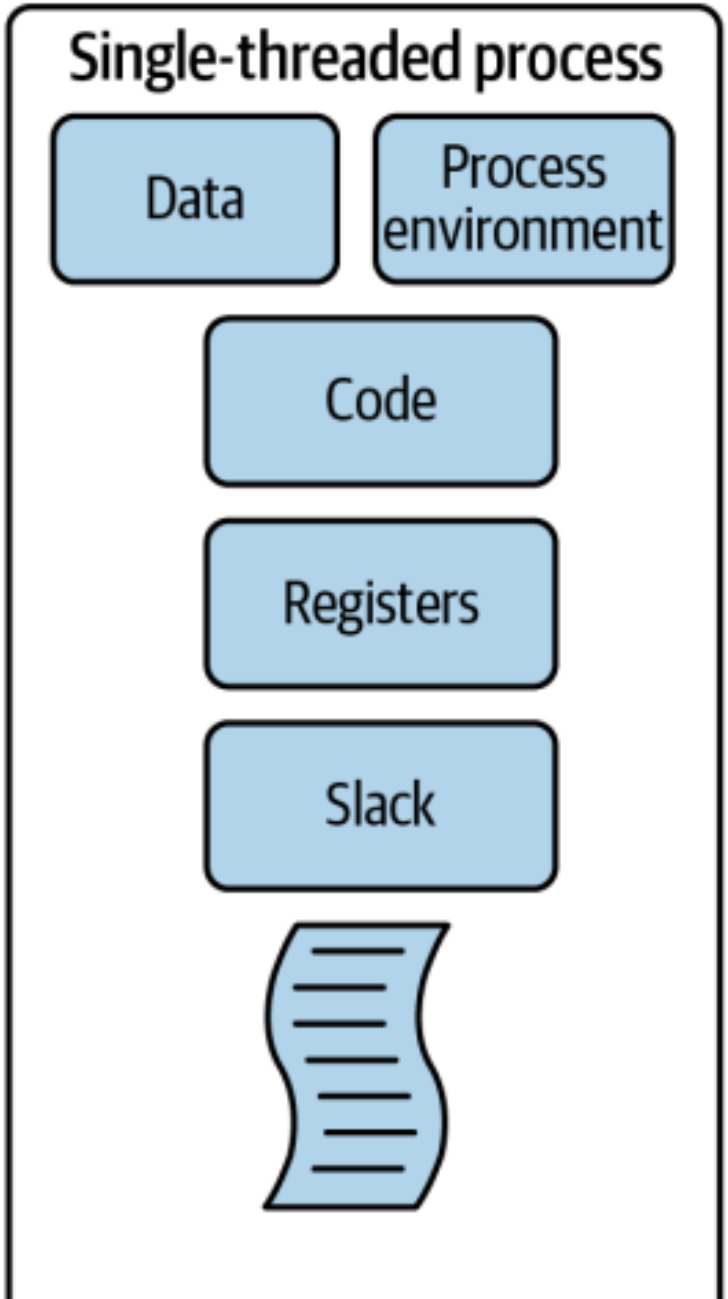
A Process

- Threads can share all resources accessible within an execution environment.
- An execution environment provides protection from threads outside it
- The data and other resources contained in an execution environment are by default inaccessible to threads residing in other execution environments.
- Example:

A Single-threaded Process

- Every software process has a single thread of execution by default.
- This is the thread that the operating system manages when it schedules the process for execution.
- This single thread has access to the program's environment and resources such as open file handles and network connections.
- As the program calls methods in objects instantiated in the code, the program's runtime stack is used to pass parameters and manage variable scopes

A Single-threaded Process



A Multi-threaded Process

- You can use programming languages to create and execute additional threads within the same process.
- Each thread is an independent sequence of execution
- What info is shared and what info is thread-specific?

A Single threaded vs. Multi-threaded Process

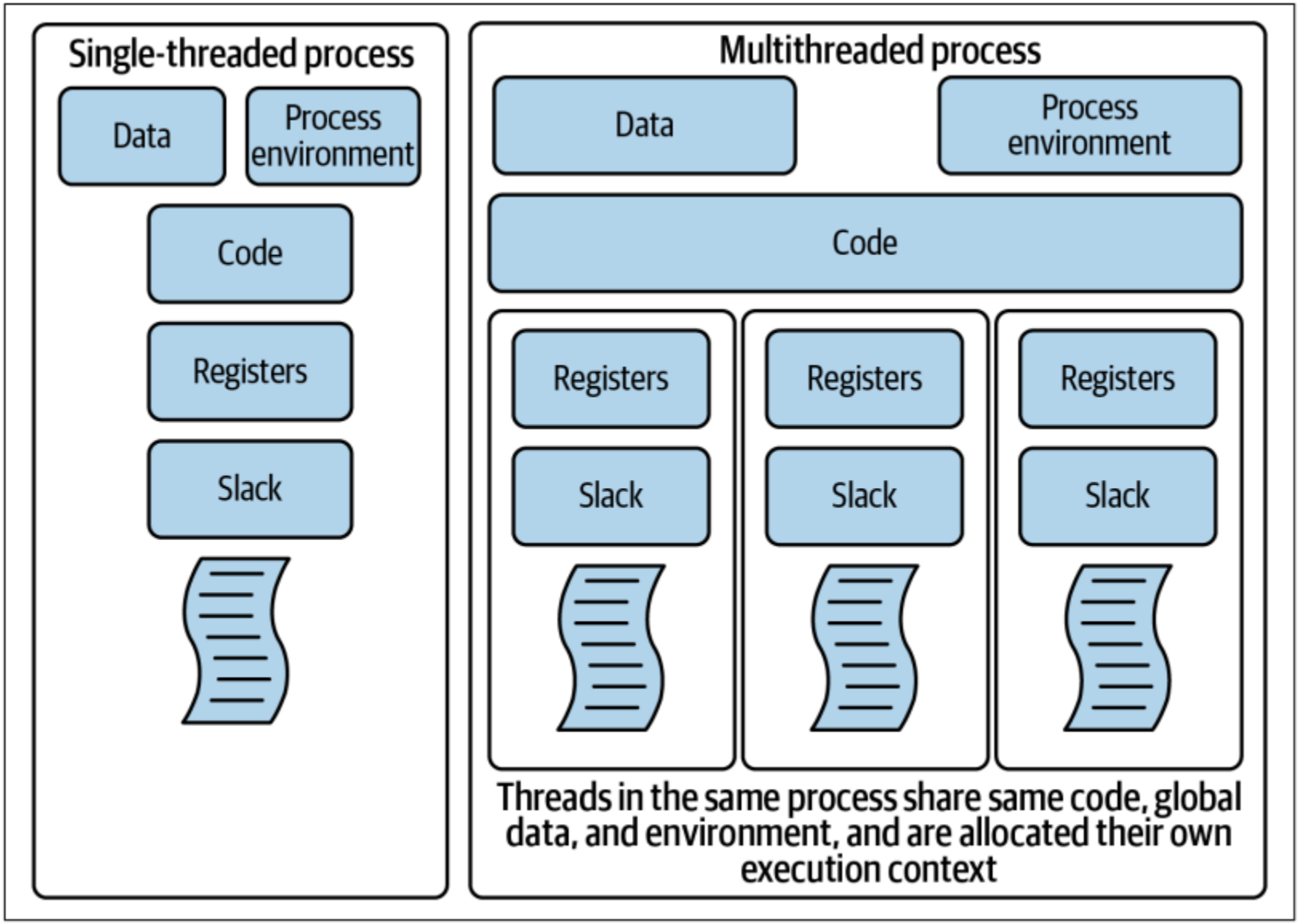


Figure 4-2. Comparing a single-threaded and multithreaded process

Process vs. Thread

- Communication
- Context switching
- Security

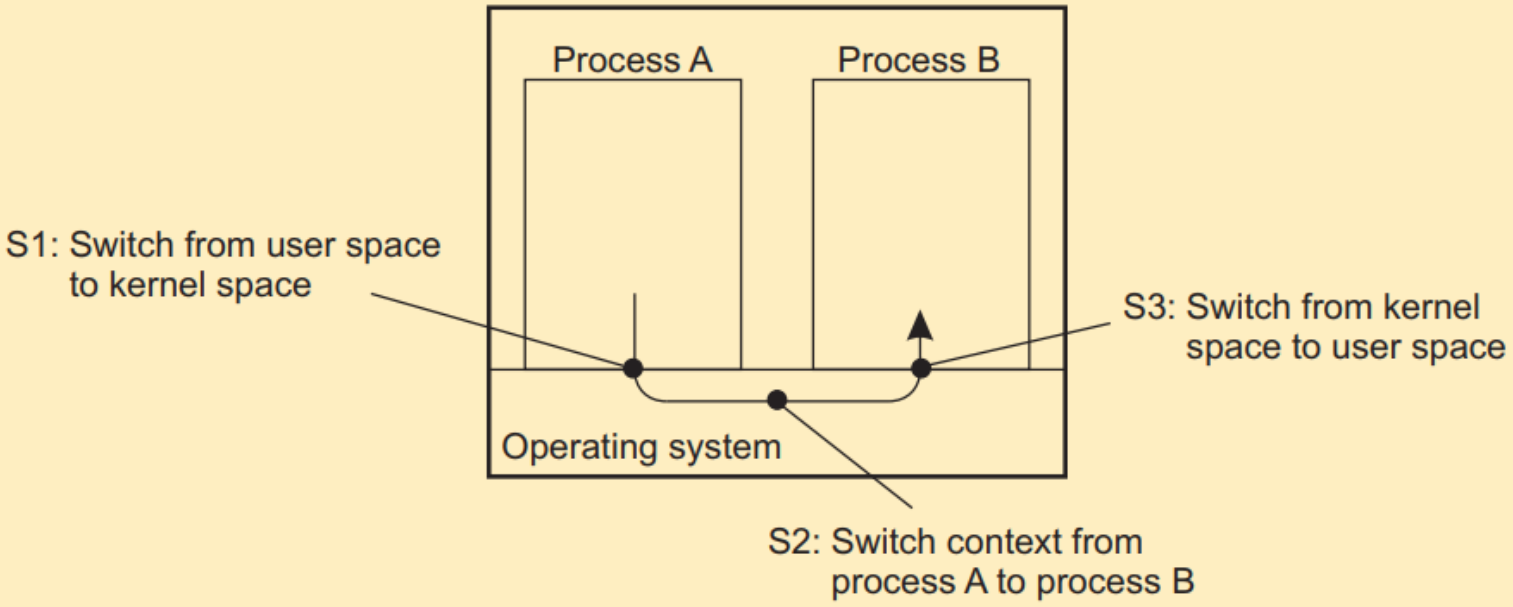
Context switching

Observations

1. Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
2. Process switching is generally (somewhat) more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.
3. Creating and destroying threads is much cheaper than doing so for processes.

Avoid process switching

Avoid expensive context switching

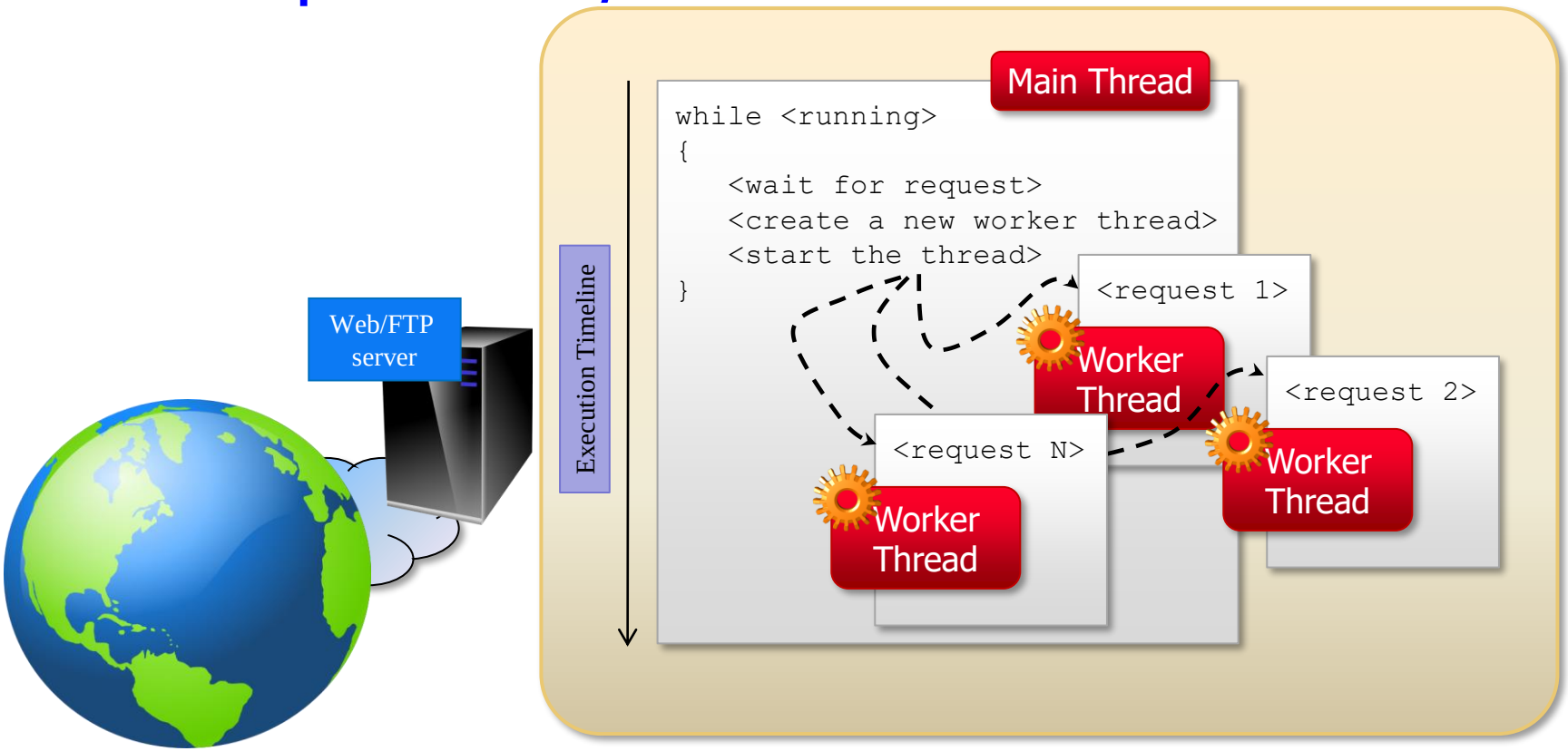


Defining Threads

- Applications – Threads are used to perform:
  - Parallelism and concurrent execution of independent tasks / operations.
  - Non blocking I/O operations.
  - Asynchronous behavior.

Defining Threads

- Example: Web/FTP Server



Defining Threads

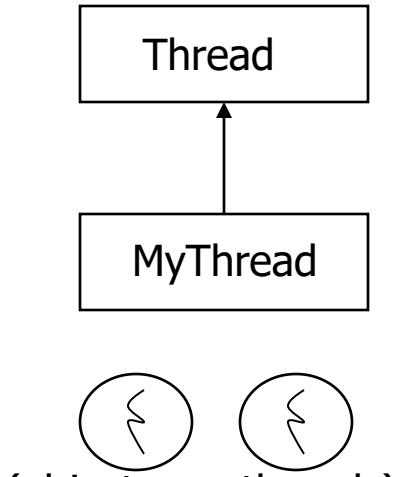
- Summing Up
  - A Thread is a piece of code that runs in concurrent with other threads.
  - Each thread is a statically ordered sequence of instructions.
  - Threads are used to express concurrency on both single and multiprocessors machines.
  - Programming a task having multiple threads of control – Multithreading or Multithreaded Programming.

Java Threads

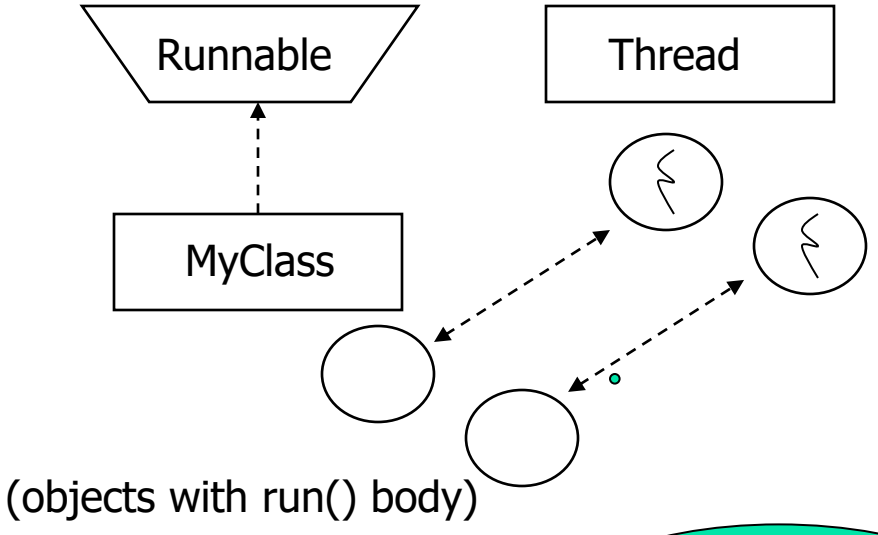
- Java has built in support for Multithreading
- Synchronization
- Thread Scheduling
- Inter-Thread Communication:
  - currentThread
  - yield
  - sleep
  - resume
  - start
  - run
  - stop
  - setPriority
  - getPriority
  - suspend
- Java Garbage Collector is a low-priority thread.

Threading Mechanisms...

- Create a class that extends the Thread class
- Create a class that implements the Runnable interface



[a]



[b]

Which is better?

1st method: Extending Thread class

- Create a class by extending Thread class and override run() method:

```
class MyThread extends Thread
{
     public void run()
    {
        // thread body of execution
    }
}
```

- Create a thread:

```
MyThread thr1 = new MyThread();
```

- Start Execution of threads:

```
thr1.start();
```

- Create and Execute:

```
new MyThread().start();
```

An example

```
class MyThread extends Thread {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
}

class ThreadEx1 {
        public static void main(String [] args  ) {
            MyThread t = new MyThread();
             t.start();
       }
}
```

2nd method: Threads by implementing Runnable interface

- Create a class that implements the interface Runnable and override run() method:

```
class MyThread implements Runnable
{
   .....
   public void run()
   {
      // thread body of execution
   }
}
```

- Creating Object:

```
MyThread myObject = new MyThread();
```

- Creating Thread Object:

```
Thread thr1 = new Thread( myObject );
```

- Start Execution:

```
thr1.start();
```

An example

```
class MyThread implements Runnable  {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
}

class ThreadEx2 {
        public static void main(String [] args  ) {
                Thread t = new Thread(new MyThread());
                 t.start();
        }
}
```

Questions from the Previous Lecture

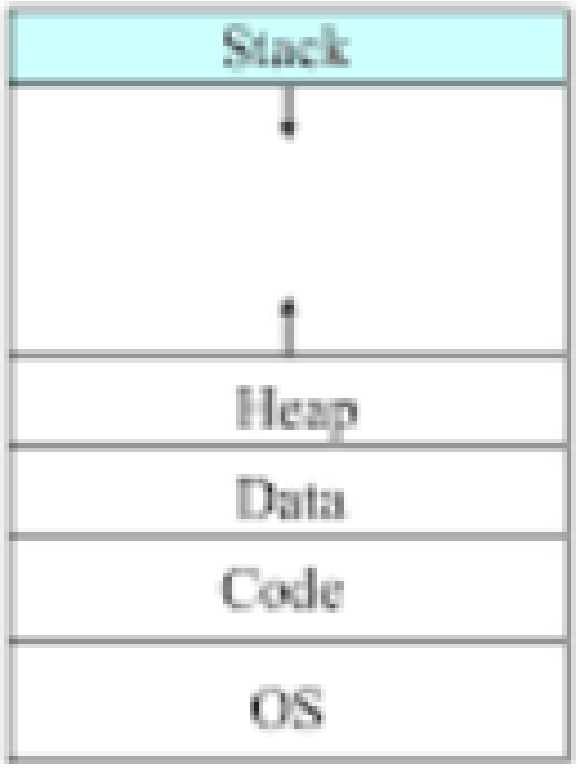
- Threading using Thread class versus Runnable interface?

Questions from the Previous Lecture

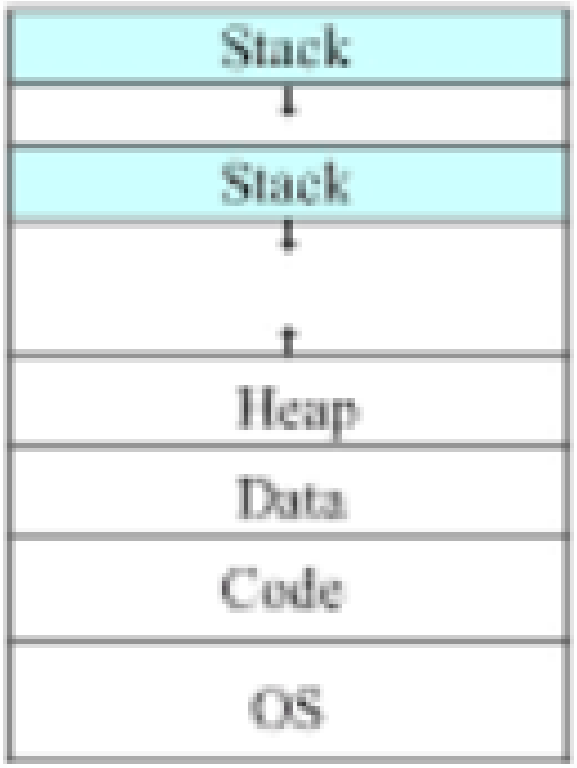
- Security breach within threads?
- Attackers exploit buffer overflow issues by overwriting the memory of an application
- If attackers know the memory layout of a program, they can overwrite areas that hold executable code, replacing it with their own code.
- For example, an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload, to gain control over the program

Questions from the Previous Lecture

- Security breach within threads?



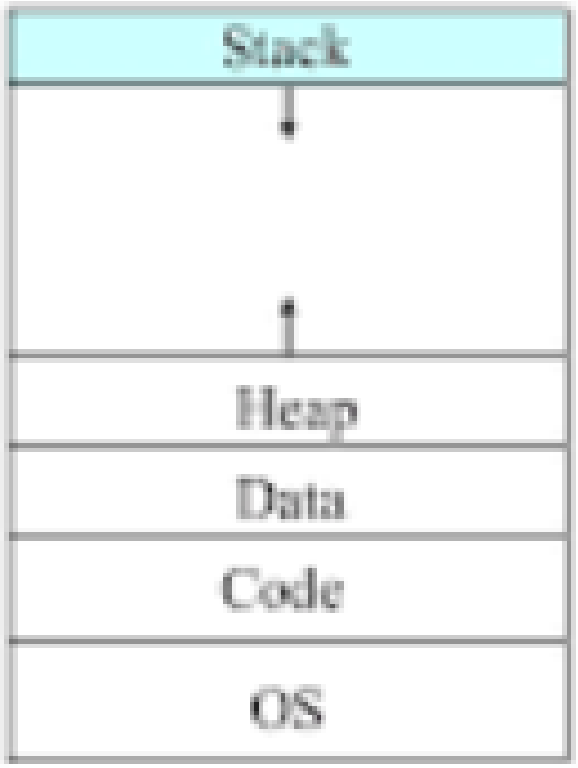
(a) Process with Single Thread



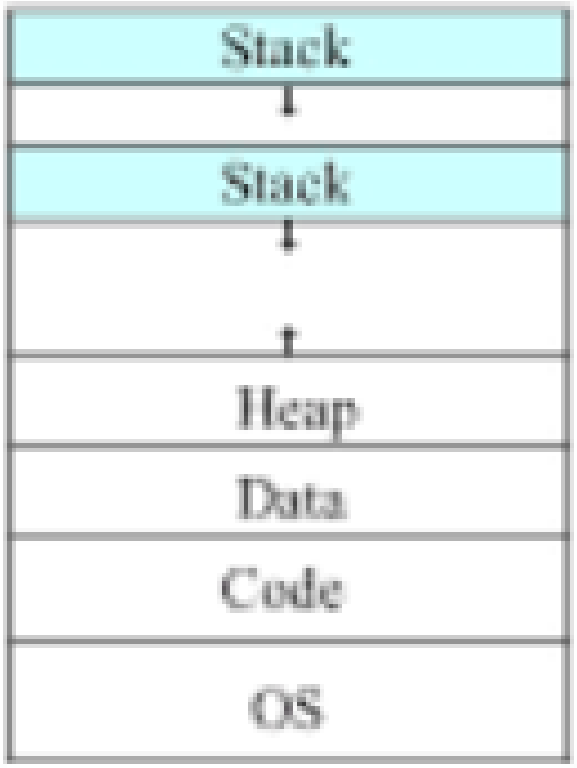
(b) Process with Two Threads

Questions from the Previous Lecture

- Security breach within threads?



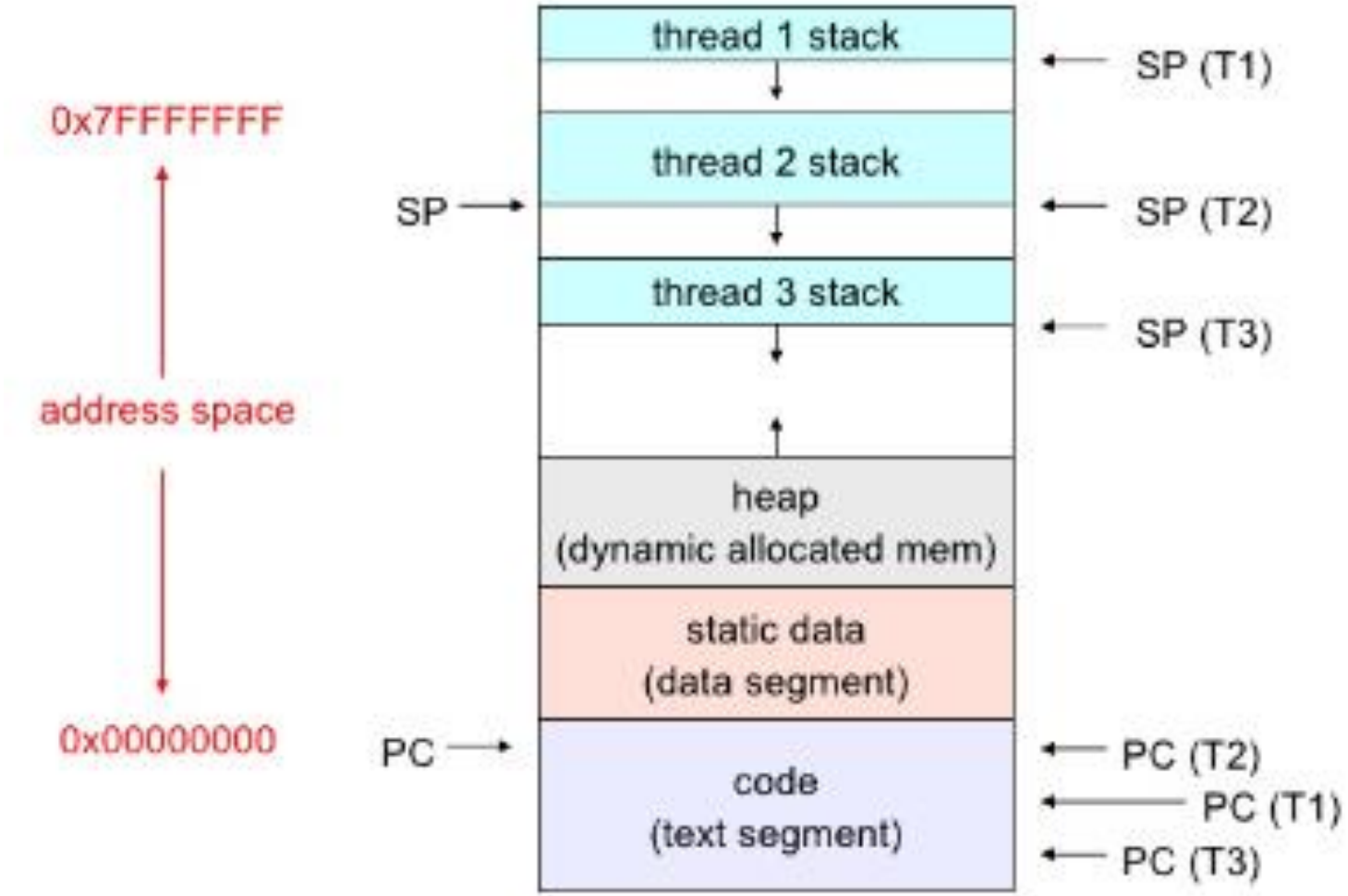
(a) Process with Single Thread



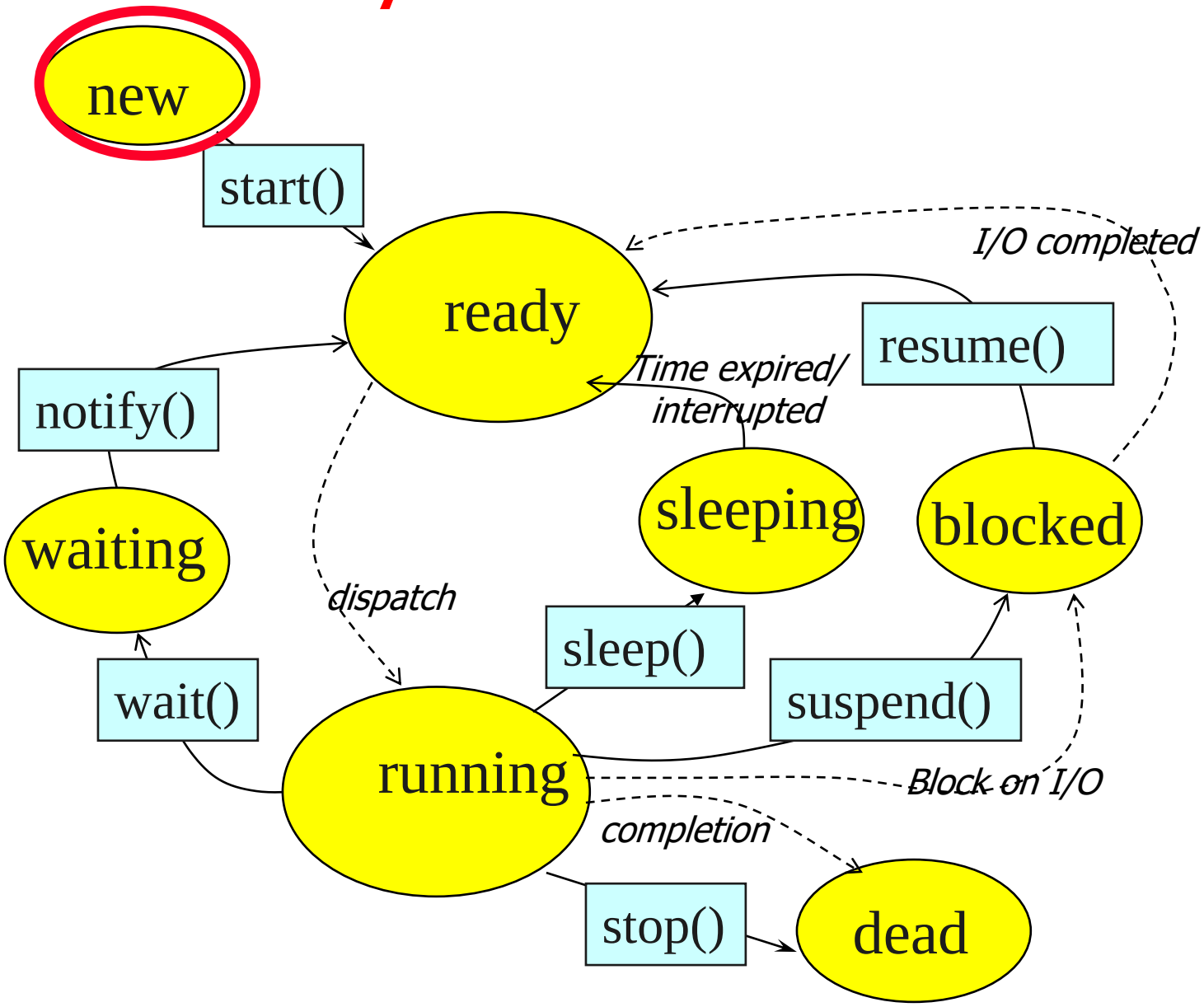
(b) Process with Two Threads

Questions from the Previous Lecture

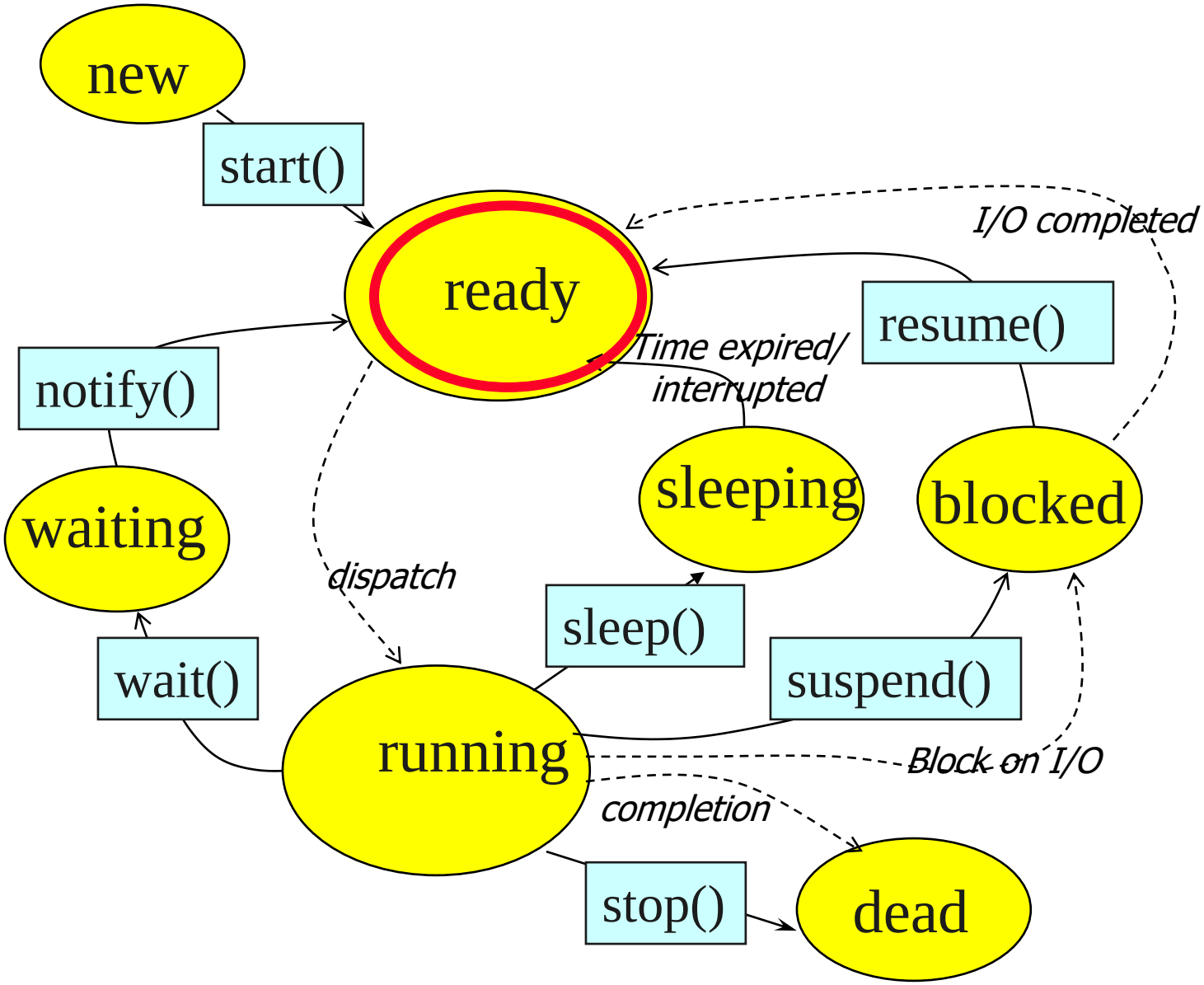
- Security breach within threads?



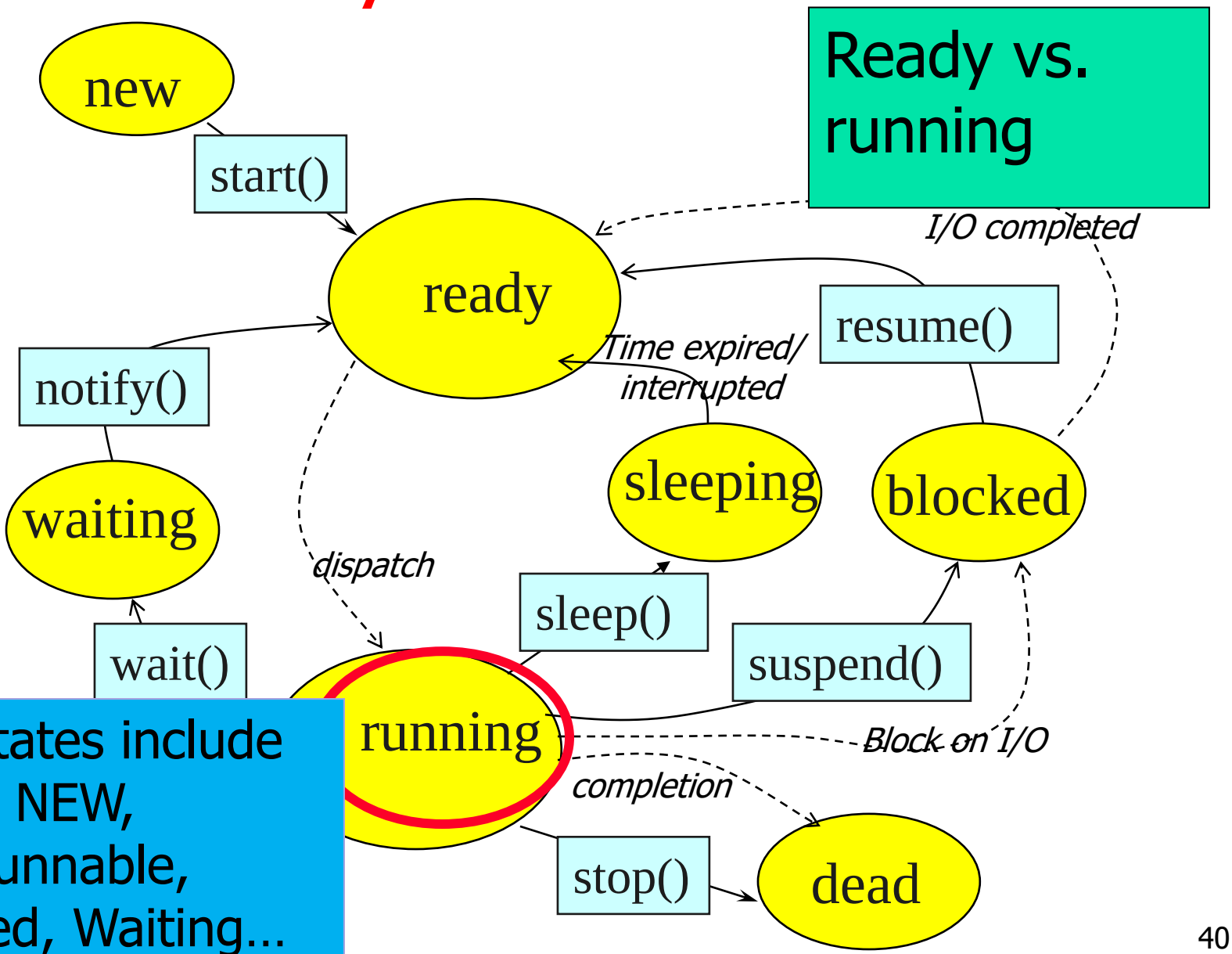
Life Cycle of Thread



Life Cycle of Thread

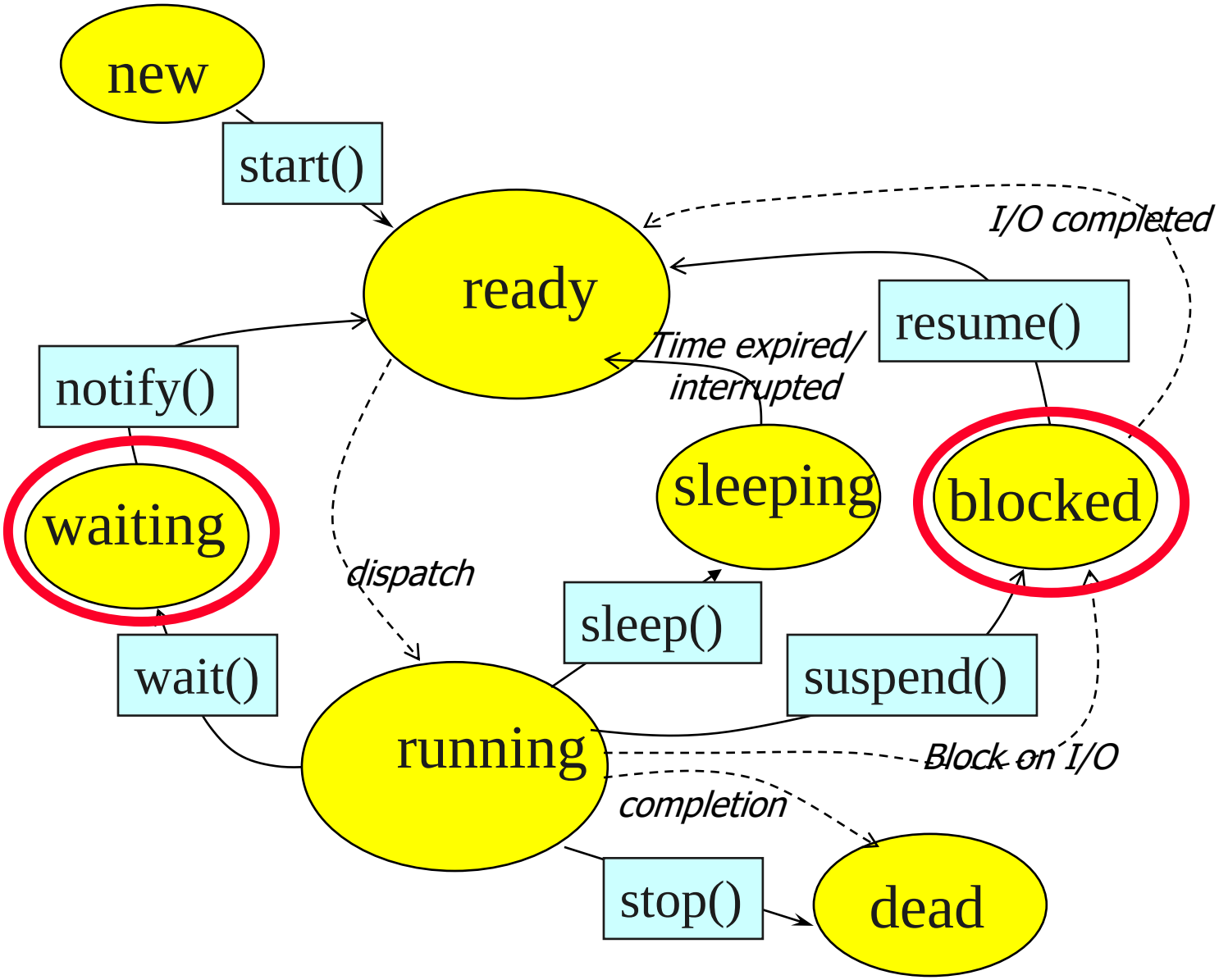


Life Cycle of Thread



Java states include
NEW,
Runnable,
Blocked, Waiting…

Life Cycle of Thread



Thread Operations

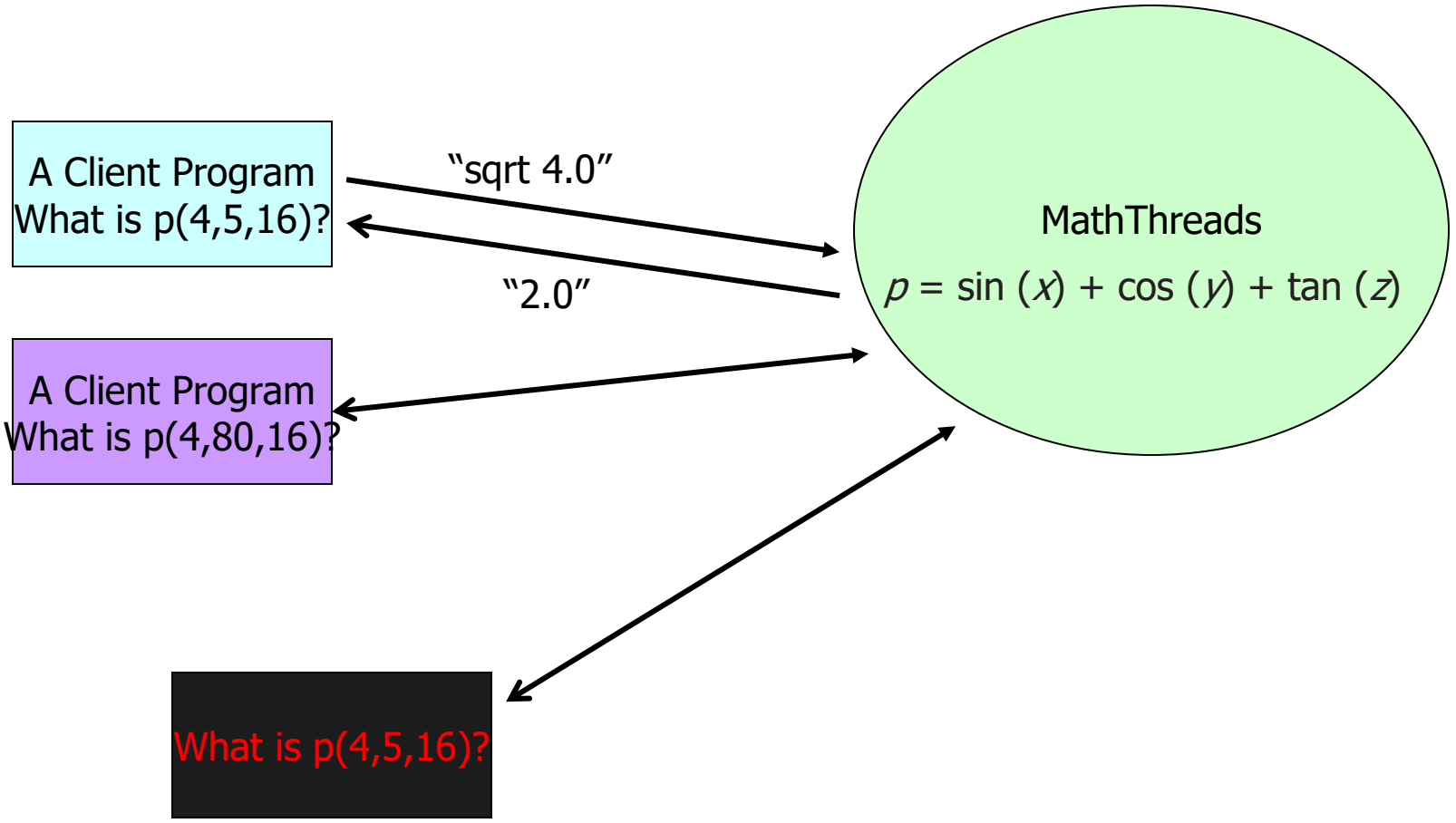
- `public static void sleep(long millis) throws InterruptedException`

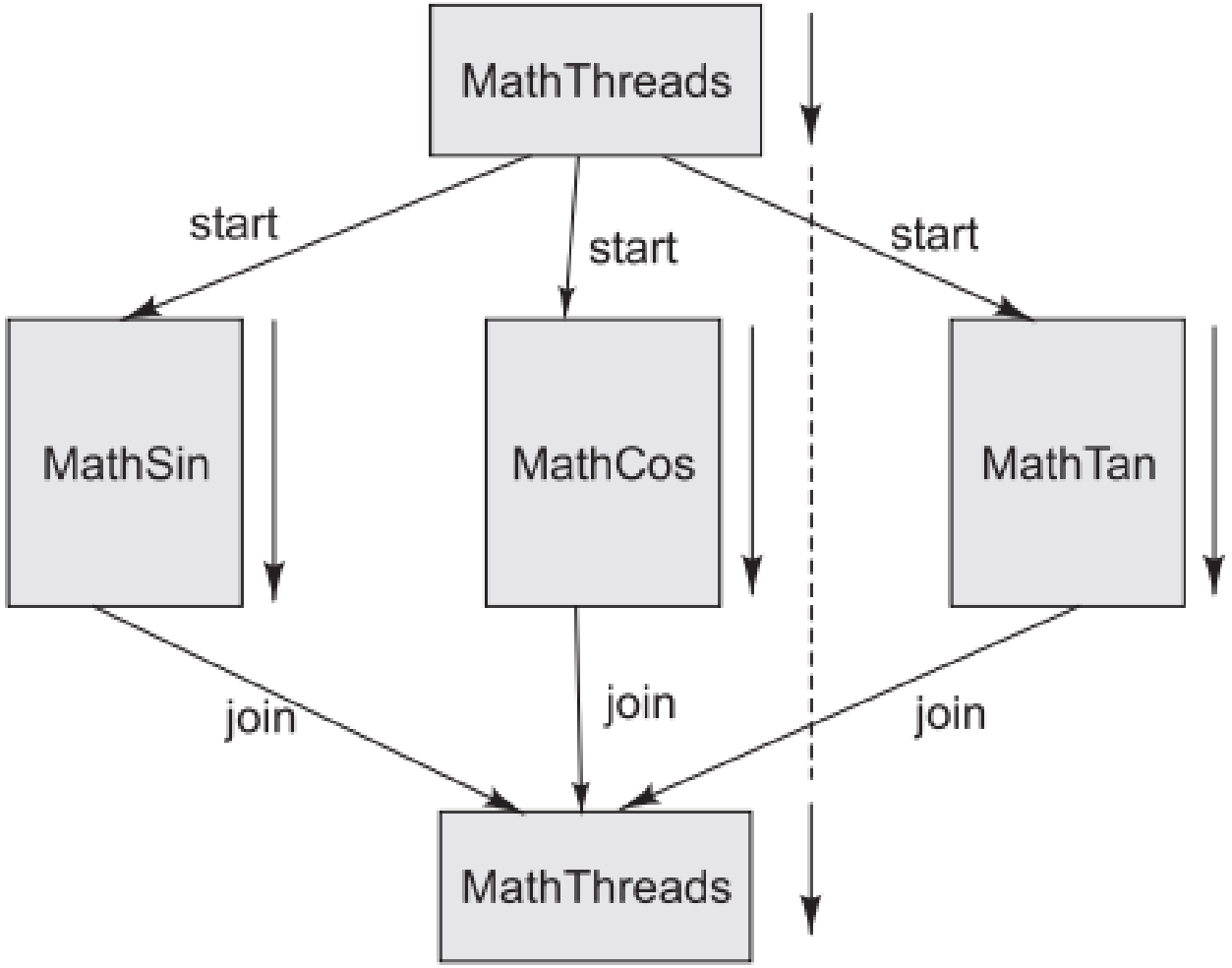
- For example, the code below puts the thread in sleep state for 3 minutes
- try {
  Thread.sleep(3 * 60 * 1000); // thread sleeps for 3 minutes
  } catch(InterruptedException ex){}

Thread Operations

- public static void yield() → Back to Runnable!
- public final boolean isAlive()
- void join()
  void join(long millis)
  void join(long millis, int nanos)

Example 1: MathServer – Demonstrates the use of Threads




MathThreads
start
start
start
MathSin
MathCos
MathTan
join
join
join
MathThreads

```
/* MathThreads.java: A program with multiple threads performing concurrent
operations. */
import java.lang.Math;
class MathSin extends Thread {
   public double deg;
   public double res;

   public MathSin(int degree) {
     deg = degree;
   }
   public void run() {
            System.out.println("Executing sin of "+deg);
      double Deg2Rad = Math.toRadians(deg);
      res = Math.sin(Deg2Rad);
      System.out.println("Exit from MathSin. Res = "+res);
   }
}
class MathCos extends Thread {
   public double deg;
   public double res;
```

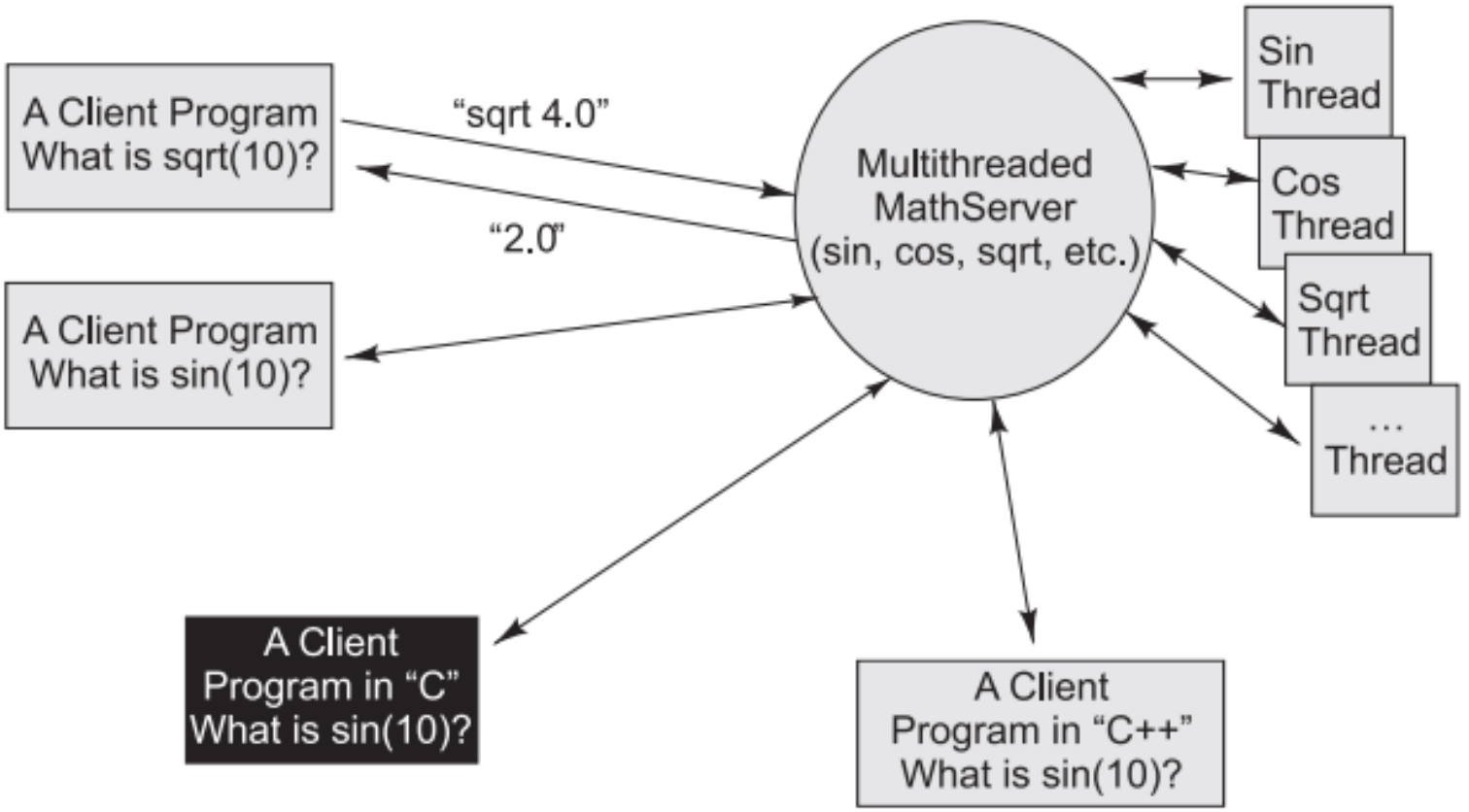
```
    public MathCos(int degree) {
       deg = degree;
    }
    public void run() {
       System.out.println("Executing cos of "+deg);
       double Deg2Rad = Math.toRadians(deg);
       res = Math.cos(Deg2Rad);
       System.out.println("Exit from MathCos. Res = "+res);
    }
}
class MathTan extends Thread {
    public double deg;
    public double res;

    public MathTan(int degree) {
       deg = degree;
    }
    public void run() {
       System.out.println("Executing tan of "+deg);
       double Deg2Rad = Math.toRadians(deg);
       res = Math.tan(Deg2Rad);
       System.out.println("Exit from MathTan. Res = "+res);
    }
}
```

```
class MathThreads {
   public static void main(String args[]) {
      MathSin st = new MathSin(45);
      MathCos ct = new MathCos(60);
      MathTan tt = new MathTan(30);
      st.start();
      ct.start();
      tt.start();
      try {   // wait for completion of all thread and then sum
        st.join();
        ct.join(); //wait for completion of MathCos object
        tt.join();
        double z = st.res + ct.res + tt.res;
        System.out.println("Sum of sin, cos, tan = "+z);
      }
      catch(InterruptedException IntExp) {
      }
   }
}
```

Run 1:

```
[raj@mundroo] threads [1:111] java MathThreads
Executing sin of 45.0
Executing cos of 60.0
Executing tan of 30.0
```

Multi-threaded Math Server



Multi-threaded Math Server

```
/* MultiThreadMathServer.java: A program extending MathServer which
allows concurrent client requests and opens a new thread for each socket
connection. */
import java.net.ServerSocket;
import java.net.Socket;
public class MultiThreadMathServer
             extends MathServer implements Runnable {
   public void run() {
```

Multi-threaded Math Server

```
        execute();
    }
    public static void main(String [] args)throws Exception {
      int port = 10000;
      if (args.length == 1) {
          try {
             port = Integer.parseInt(args[0]);
          }
           catch(Exception e) {
          }
      }
      ServerSocket serverSocket = new ServerSocket(port);
      while(true){
        //waiting for client connection
        Socket socket = serverSocket.accept();
        socket.setSoTimeout(14000);
        MultiThreadMathServer server = new MultiThreadMathServer();
        server.setMathService(new PlainMathService());
        server.setSocket(socket);
        //start a new server thread...
        new Thread(server).start();
      }
    }
}
```

How about the client code?

Problems with Threads

- The basic problem in concurrent programming is coordinating the execution of multiple threads so that whatever order they are executed in, they produce the correct answer.
- Given that threads can be started and preempted nondeterministically, any moderately complex program will have an infinite number of possible orders of execution.
- Two main problems that concurrent programs need to avoid are race conditions and deadlocks.

Problems with Threads (Cont'd)

Race conditions

- Nondeterministic execution of threads implies that the code statements that comprise the threads:
  - Will execute sequentially as defined within each thread.
  - Can be overlapped in any order across threads. (Why?)
- When would such non-determinism cause issues?

Problems with Threads (Cont'd)

```
public class RequestCounter {
  final static private int NUMTHREADS = 50000;
  private int count = 0;

  public  void inc() {
    count++;
  }

  public int getVal() {
    return this.count;
  }

  public static void main(String[] args) throws InterruptedException {
    final RequestCounter counter = new RequestCounter();

    for (int i = 0; i < NUMTHREADS; i++) {
      // lambda runnable creation
      Runnable thread = () -> {counter.inc(); };
        new Thread(thread).start();
    }

    Thread.sleep(5000);
    System.out.println("Value should be " + NUMTHREADS + "It is: " +
counter.getVal());
  }
}
```

Problems with Threads (Cont'd)

Race conditions

- To perform an increment of a counter, the CPU must:
  - Load the current value into a register.
  - Increment the register value.
  - Write the results back to the original memory location
- This is a sequence of **three** machine level instructions.
- However, such operations are not treated as a **single atomic operation.**

Problems with Threads (Cont'd)

Race conditions

Thread 1	Thread 2
Reads (x) into register	
Register value +6	
Writes register value to (x)	
	Reads (x) into register
	Register value +1
	Writes register value to (x)

Thread 1	Thread 2
Reads (x) into register	
Register value +6	
	Reads (x) into register
	Register value +1
	Writes register value to (x)
Writes register value to (x)	

Figure 4-3. Increments are not atomic at the machine level

Problems with Threads (Cont'd)

Deadlocks

- To ensure correct results in multithreaded code, we have to restrict the inherent nondeterminism to serialize access to critical sections.
- This avoids race conditions.
- But, we can write code that restricts nondeterminism so much that our program stops. And never continues. This is formally known as a deadlock..
- How would it happen within threads?

Required Readings

- CDK Book (Text Book)
  - Chapter 7 – "Operating System Support"
- Chapter 4: An Overview of Concurrent Systems, from: Gorton, Ian. *Foundations of Scalable Systems*. " O'Reilly Media, Inc.", 2022.
- Chapter 14: Multithread Programming, fromR. Buyya, S. Selvi, X. Chu, **"Object Oriented Programming with Java: Essentials and Applications",** McGraw Hill, New Delhi, India, 2009.