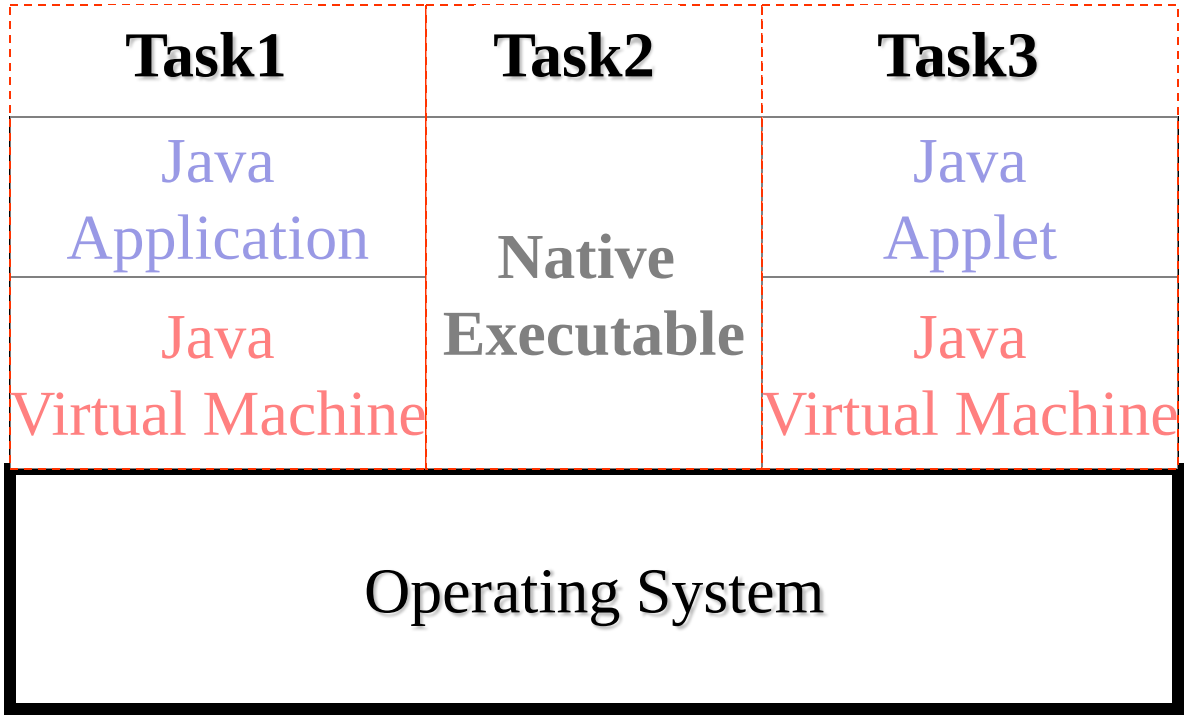
Multithreading Basics

Bob Singh

Multitasking



Multitasking

- The ability to run multiple things at the same time.
- What is True Multitasking?
- Non-preemptive – once CPU gives attention to the process, it must complete the allocated block of time
- Pre-emptive – If CPU has given attention to the process, it doesn't have to complete the allocated block of time, if the next process of higher priority comes into picture.

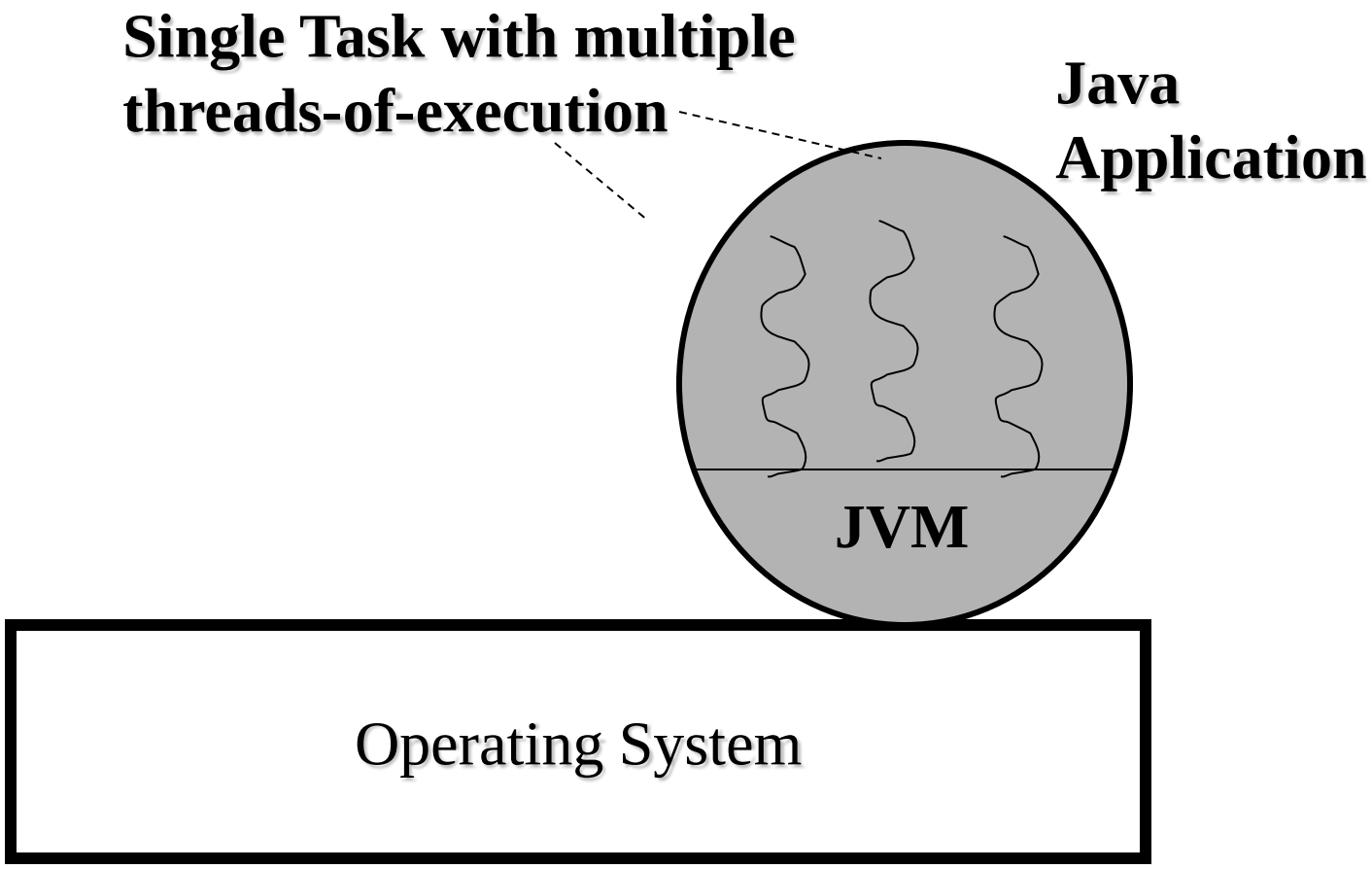
Multithreading

- Context Switching
- Threads exist within the context of a task or process
- Threads are independent execution paths within the process
- There can be many threads per process
- Threads may be software controlled and usually non pre-emptive
- Threads may be native OS threads and are usually pre-emptive

Java Multithreading

- The JVM and Packages are thread safe
- The JVM and Packages are re-entrant
- Where possible the Java Runtime Environment (JRE) uses native OS threads
  - For example native NT threads
- Alternatively a third party thread's package is used
  - For example Solaris Green Threads Package

Java Multi Threading



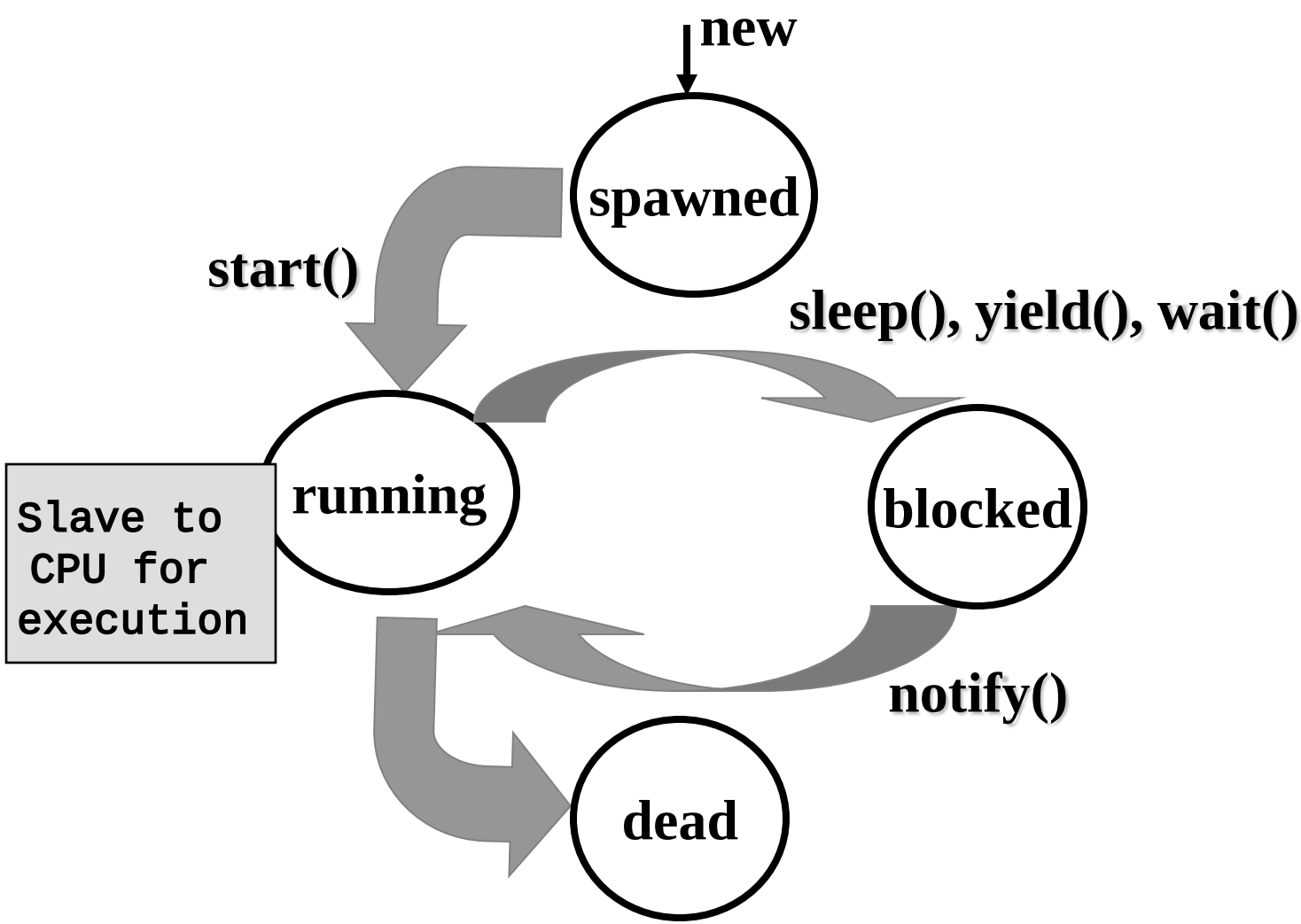
Thread Scheduling Schemes

- Windows 95/NT implement *timeslicing*
  - multiple threads of equal priority share the CPU in round-robin fashion
- Other systems don't implement timeslicing
  - thread of highest priority runs to completion or until it yields
- Java runtime *does not implement timeslicing*
  - some of the threads packages on which it's built do support it, in which case you'll see time-sliced behavior (eg, Windows NT/95)
  - don't count on timeslicing!
    - keep your threads somewhat altruistic--let them yield() occasionally
- Java runtime is pre-emptive
  - higher priority threads will grab the CPU

java.lang.Thread

- Support for multi-threading in java is through the Thread class
- A thread can occupy one of four states
  - spawned, running, blocked, dead
- Java supports thread priorities
- Java supports thread synchronization
- Java supports explicit thread control
- Some important thread support is found in java.lang.Object
  - wait, notify, notifyAll

Thread life cycle



stop(), suspend(), resume() have been deprecated in Java 2

Creating threads

- Any object instance of a class can become a thread through sub-classing
  - class Widget **extends Thread**{}
- Any object instance of a class can become a thread through interface implementation
  - class Widget **implements Runnable** {}
- The run method is the body of the thread and should be overridden
- run is defined in Thread and has a null-action if not overridden

Sub-class example 1 (Program 1)

```
class Widget extends Thread
{
	public void run() { // do something }
}

class Main
{
	public static void main(String[] args)
	{
		Widget w = new Widget();

		// thread spawned but not running

		w.start(); // enter running phase
	}
}
```

Sub-class (Program 2)

```
class Widget extends Thread
{
      public void run() { // do something }
}

class Main
{
      public static void main(String[] args)
      {
         Widget w = new Widget();

         //  2 separate widget threads
         Thread t1 = new Thread(w); t1.start();
         Thread t2 = new Thread(w); t2.start();
      }
}
```

Runnable (Program 3 with Priority)

```
class Widget extends Gromit implements Runnable
{
	public void run() { // do something }
}

class Main
{
	public static void main(String[] args)
	{
		Widget w = new Widget();
		Thread t = new Thread(w);
		t.start(); // enter running phase
	}
}
```

A word about monitors

- Monitors are *locks* on objects implemented by the JVM to facilitate synchronization
  - we'll learn a little bit later on how to use the *synchronized* keyword to create a monitor on an object
  - all the flavors of Object.wait() can cause a thread to yield a monitor it has obtained on an object and wait to be notified when the monitor is available again
  - the methods Object.notify() and Object.notifyAll() are used to tell threads waiting for a monitor that it's about to become available.
- For now, just be aware that one thread can obtain exclusive access to an object via a monitor, give it up if necessary, and wait to reacquire it

Changing a Thread's running state

- Old methods are *stop()*, *suspend(), resume()*
  - used from outside the thread to change its state
    - stop() caused the thread to instantly give up any monitors it had which could leave the object in an inconsistent state
    - suspend()/resume() + synchronization was just asking for deadlock
- This is a Bad Thing which can lead to stealth bugs
- Solution (Program 4)
  - Have your threads check a variable every once in a while and use that to have them change their own state
  - use Object.notify(), Object.notifyAll(), Object.wait() and Thread.interrupt()

```
public void run() {
    while (aCondition == true) {
     // do something periodically
    }
}
```

Stopping a thread properly

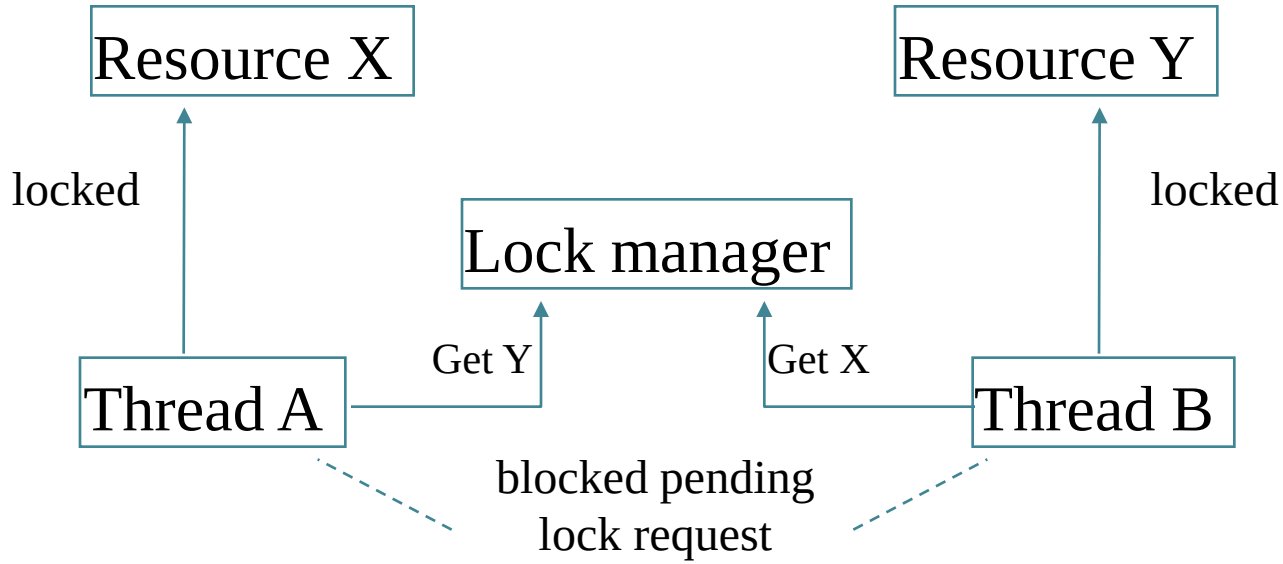
```
public class MyThread implements Runnable {
// The right way
    private Thread me;
    public void start() {
        me = new Thread(this);
        me.start();
    }

    public void stop() {
        me = null;
    }

    public void run() {
        while (me == Thread.currentThread()) {
            // do something
        }
    }

}
```

Synchronization (Program 4)

- Non-communicating threads aren't very interesting
- Threads need access to common data (*task* data)
- Need to *lock* data in *critical sections* of code to prevent two threads from modifying it at the same time
  - avoid *starvation*: one thread is locked out from access to a resource and can't do anything
  - avoid *deadlock*: two or more threads are waiting for each other to release resources the other has
    - the classic Dining Philosophers' problem
- This is where *monitors* and the *synchronized* modifier come into play

Deadlock (Program 5, Program 6)

- Deadlock occurs when two or more threads are waiting on common resources that are not exclusively owned
  - Thread A locks resource X and now requires resource Y
  - Thread B has resource Y locked and now requires resource X



Assumes that Threads A & B got X & Y locks before making next lock request

Thread Communication

- Threads can usually be grouped into consumers or producers
- Producers produce output
- Consumers require input
- Producer - Consumer relationships can be modeled as client - server threads
- Communication between them can be achieved through
  - PipedOutputStream and PipedInputStream

Piped Streams

- Producers use PipedOutputStreams to write data to a third party storage medium
- Consumers use PipedInputStreams to read data from a third party storage medium
- Using piped streams decouples the threads

Example

```
class Main
{
      try
      {
            PipedOutputStream producer =
            new PipedOutputStream();

            PipedInputStream consumer =
            new PipedInputStream( producer );

            producer.connect( consumer );
            producer.write('X');
            int i = consumer.read();
      }
}
```

Additional Examples

- **Another example of Synchronized (Program 7)**
- **Volatile (Program 9)**
- **join() (Program 10)**