
SYSC 3303 Real-Time Concurrent Systems

Life Cycle of Java Threads

Scheduling Java Threads

- Copyright © 2001-2003 D.L. Bailey and L.S. Marshall, Systems and Computer Engineering, Carleton University
- revised July 16th, 2003

Every Thread Has a Life Cycle

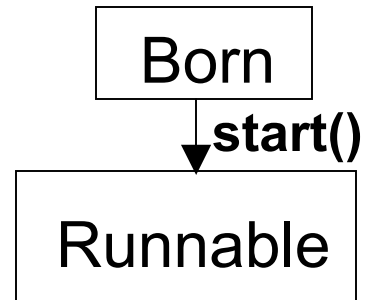
- A thread's life cycle can be modelled by a *finite state machine* (FSM) that represents the various states that the thread can be in and the thread operations that cause the transitions between the states
- We'll now look at each of these states and state transitions

Initial Thread State

Born

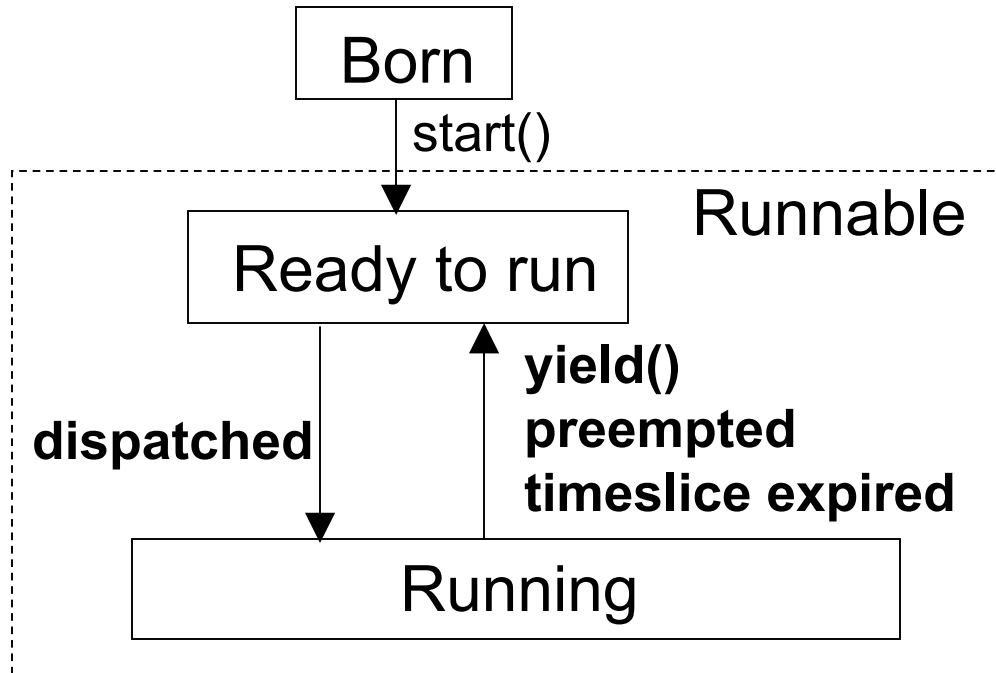
A newly created thread
is in the *born* state

Starting the Thread



Invoking `start()` places the thread in the *Runnable* state.

Ready To Run vs. Running



A Thread's code is executed by the processor only when the thread is in the Running state

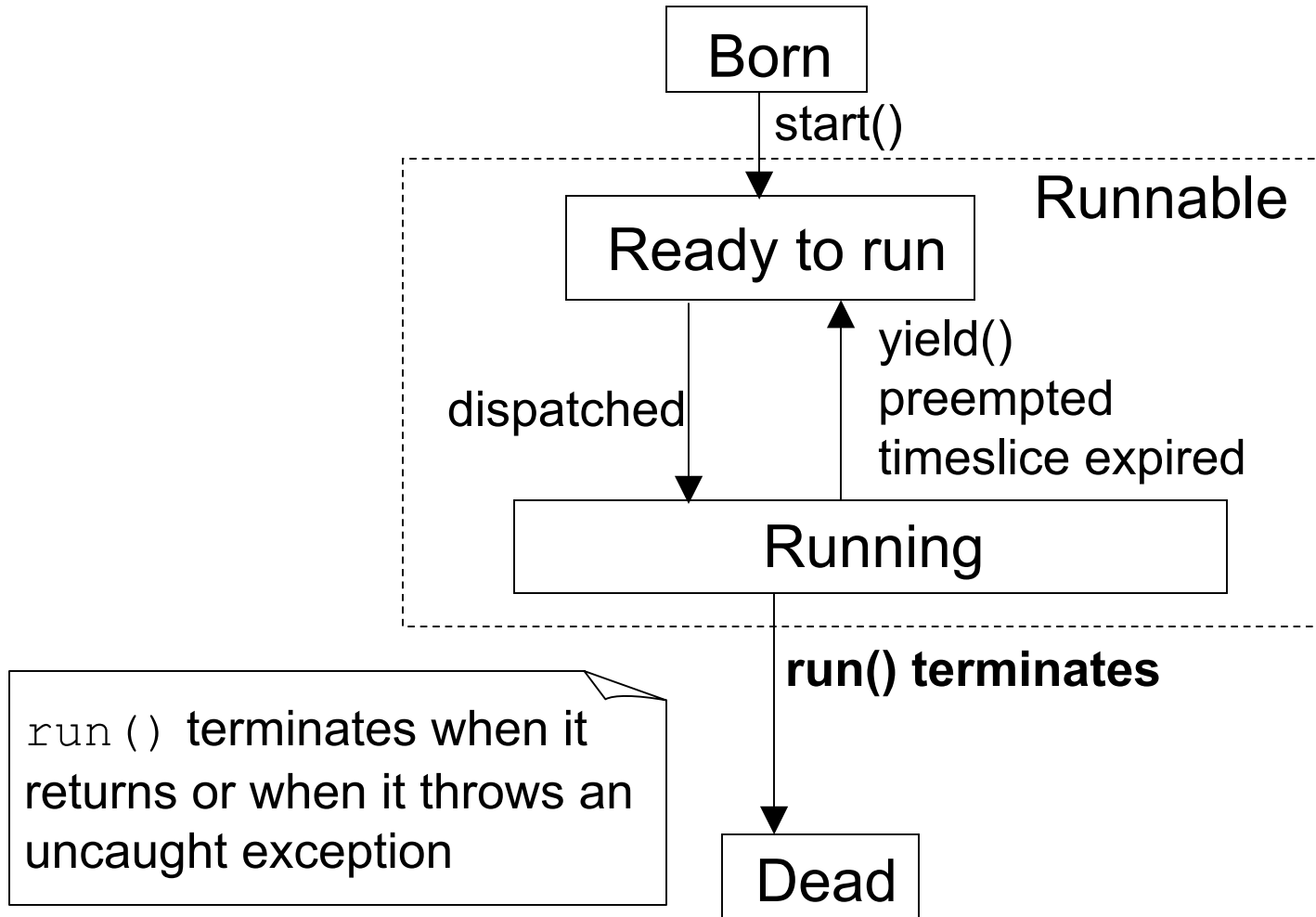
Ready To Run vs. Running

- If we had true concurrency (parallel thread execution) a thread would be running (a.k.a. *active*, *current*) whenever it is in the Runnable state
- On a single processor system, concurrency is realized by sharing the processor between the Runnable threads, so we distinguish between threads being *Ready to run* and *Running*
 - on a single processor computer, at most one thread at a time is in the Running state: the thread that is currently executing

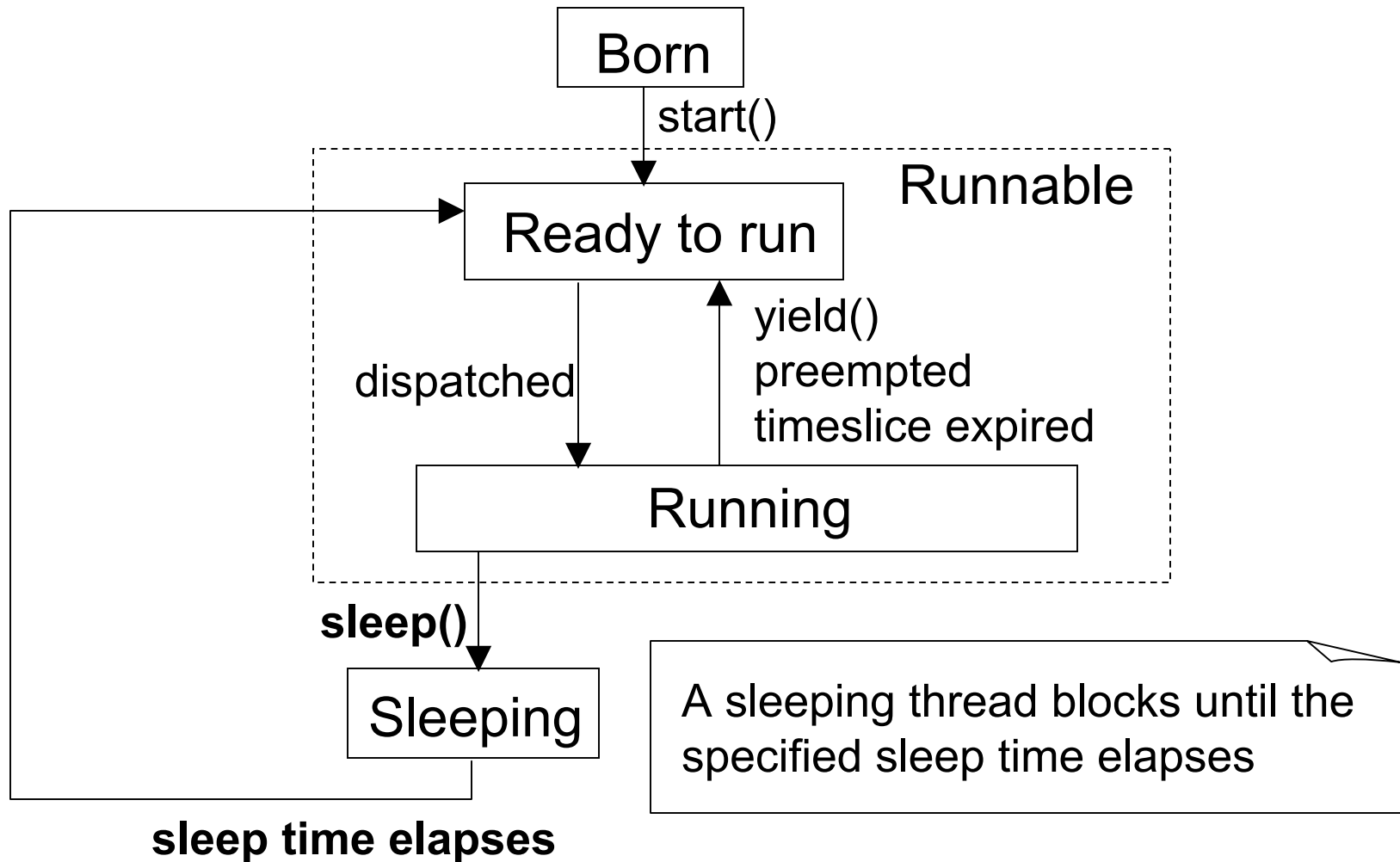
Ready To Run vs. Running

- a Ready to run thread's state changes to Running when it is scheduled for execution (*dispatched*) by the thread scheduler
- The currently executing thread can move itself from the Running state to the Ready to run state by invoking `Thread's yield()` method to indicate that it is willing to relinquish the processor
- The other transitions from Running to Ready to Run ("preempted", timeslice expired") are caused by the JVM
 - more later, when we look at thread scheduling

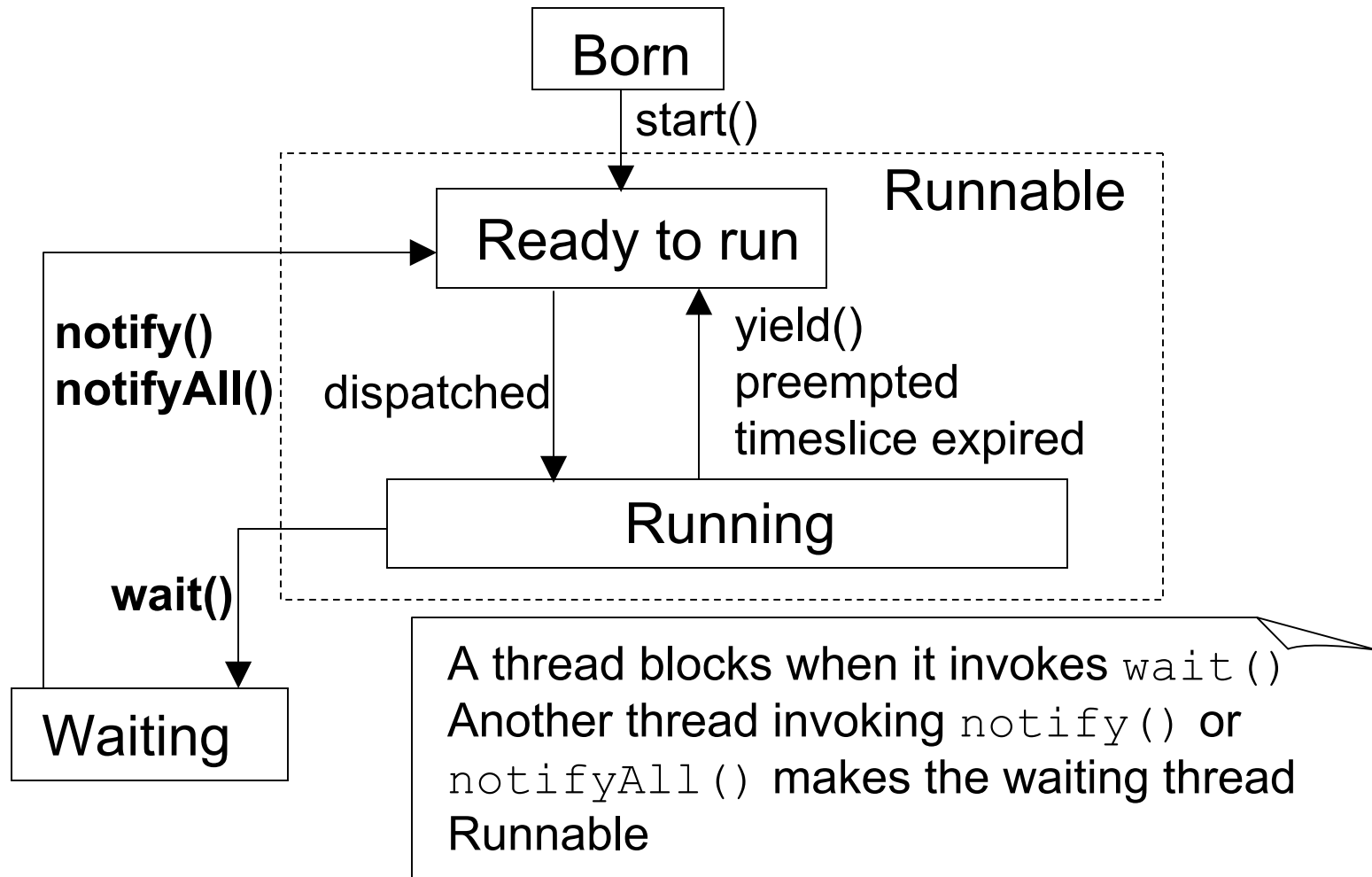
Thread Death



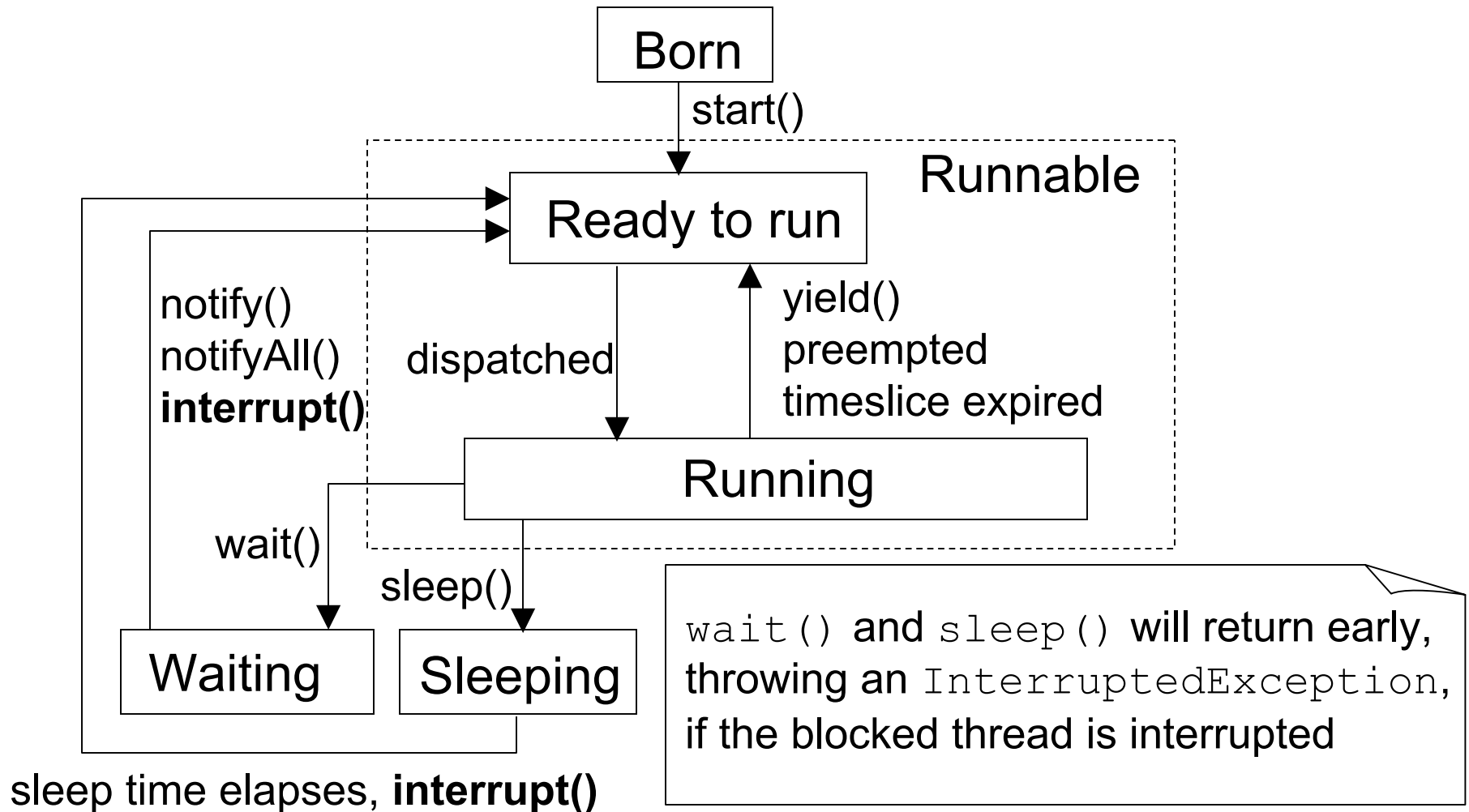
Thread Sleeping



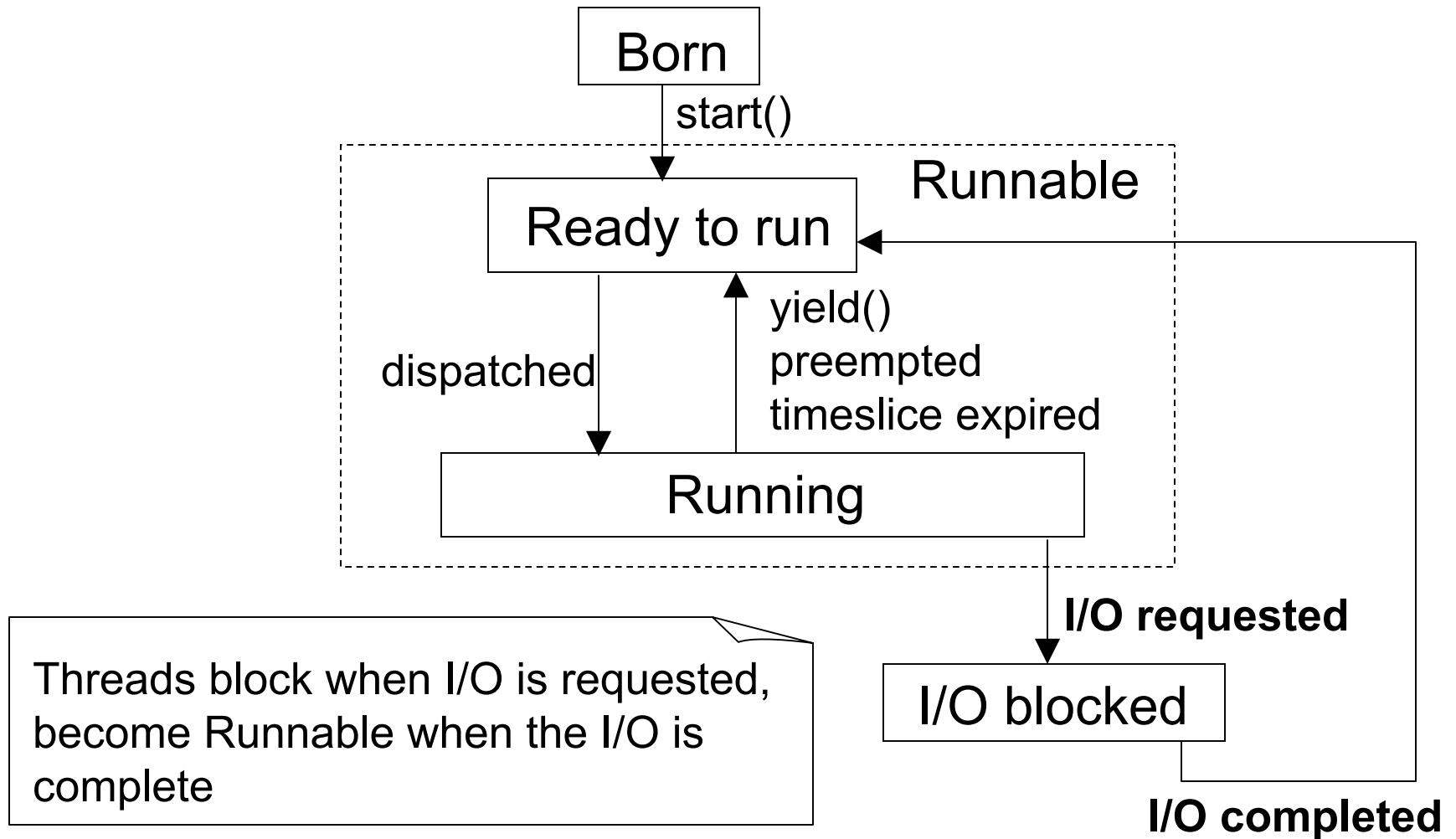
Thread Waiting



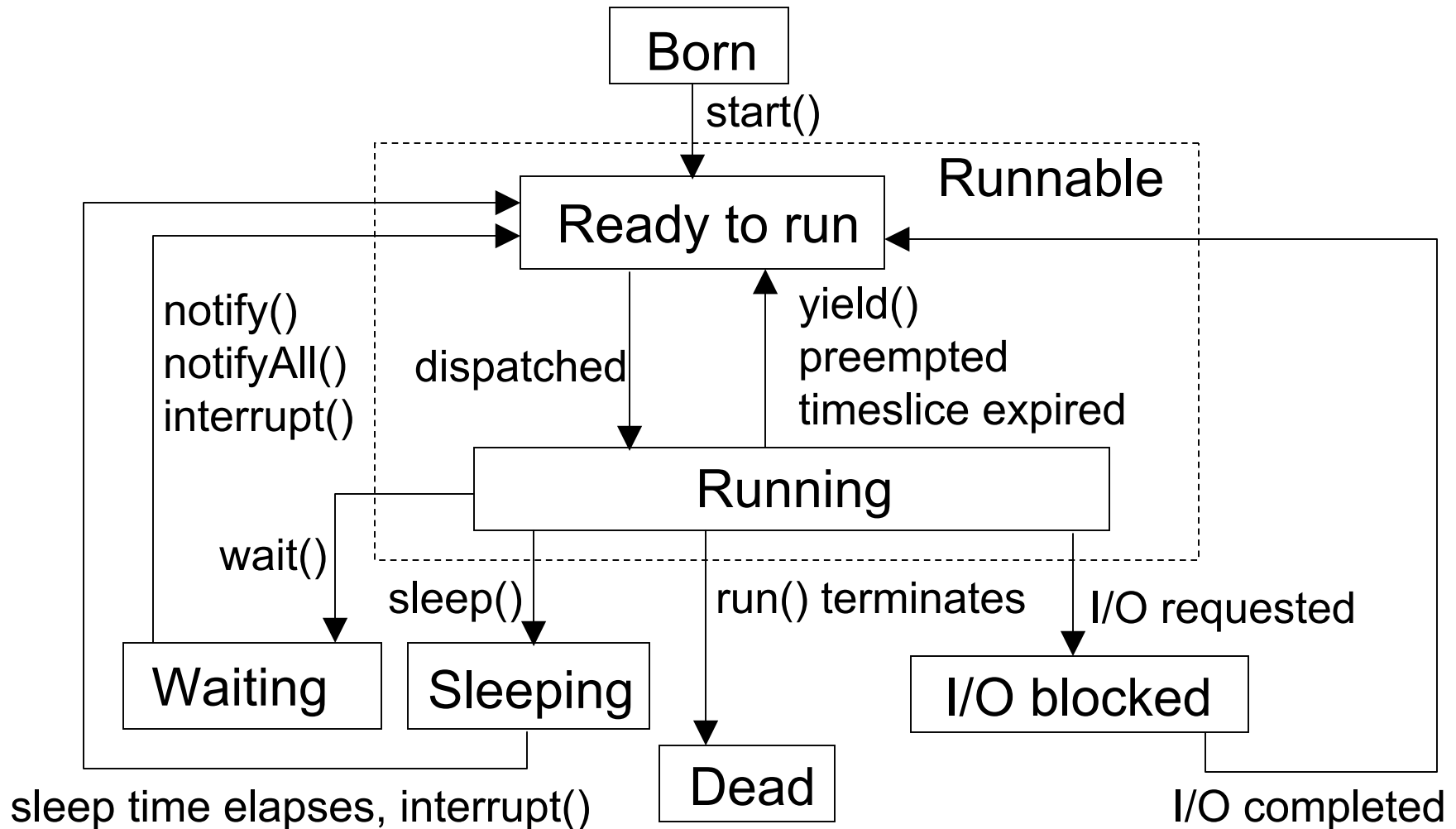
Thread Interruption



Thread I/O Blocking



Thread Life Cycle



Key Ideas About Thread Execution

- At any time, only one thread is executing
- A thread can invoke a method only when it is executing
- Methods are always executed in the invoking thread's context
- If a method invocation causes a thread to relinquish the processor, that method does not return until the thread is once again scheduled for execution and is running

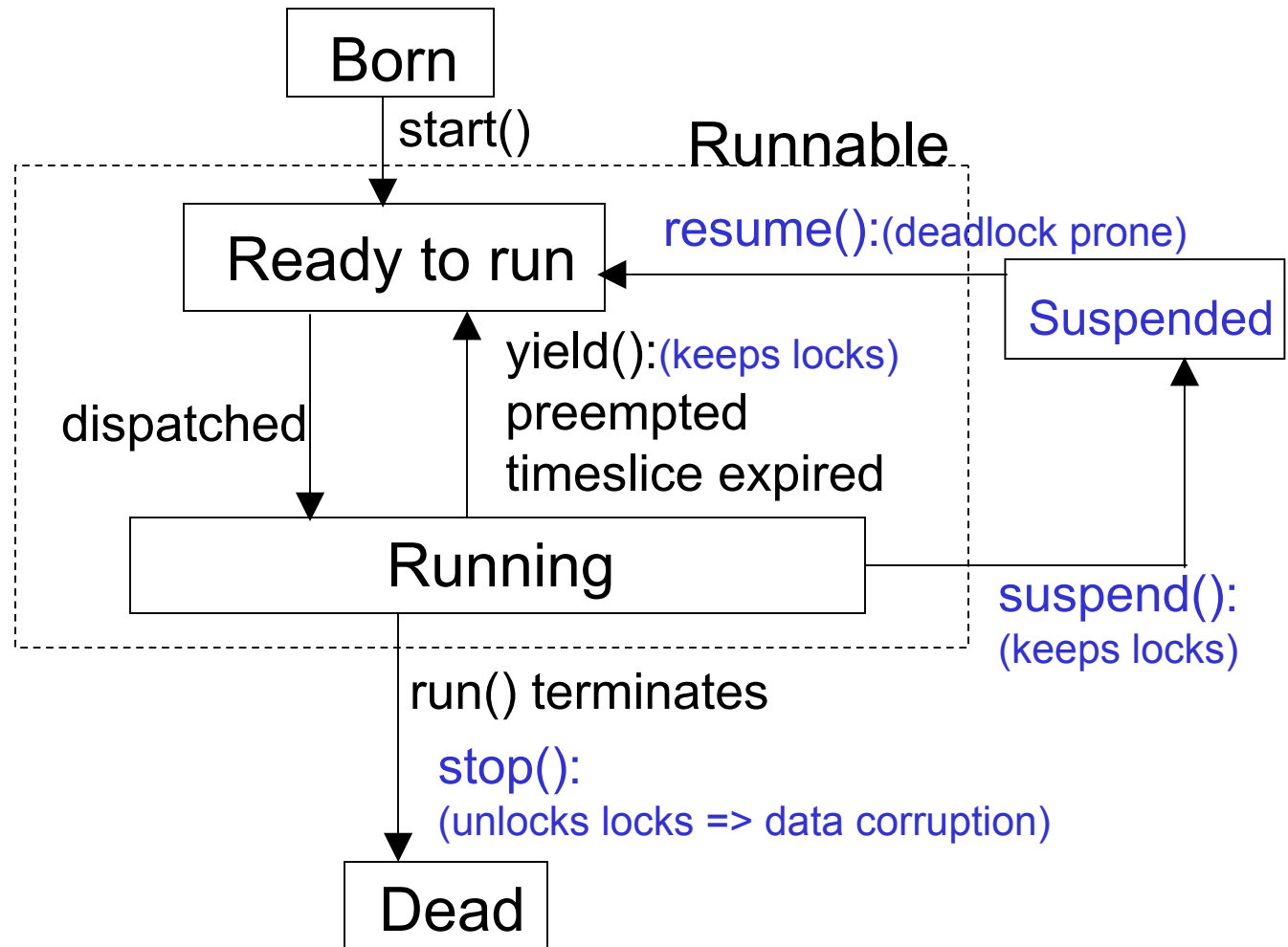
Deprecated Thread Methods

- We won't consider the effect of the following methods on a thread's life cycle: `stop()`, `suspend()`, and `resume()`, because they have been *deprecated* and should never be invoked
 - `stop()` - forces the thread to stop, regardless of what it is doing
 - this method is dangerous
 - all locks held by the thread are unlocked, and the objects protected by those locks may be in an inconsistent state, but can now be accessed by other threads

Deprecated Thread Methods

- `suspend()` - suspends the thread
- `resume()` - resumes the suspended thread
- these two methods are deprecated because `suspend()` is deadlock prone
 - the suspended thread holds onto any locks that it owns when it is suspended
 - consider what would happen if the thread that invokes `resume()` to resume the suspended thread first tried to acquire the lock held by the suspended thread

Yield vs Suspend/Resume vs Stop



Thread Scheduling

- We're now ready to look at how the Java runtime system schedules thread execution; i.e., shares a single processor between multiple threads that are in the Runnable state
- Before we look at thread scheduling in detail, we need to introduce the concept of *thread priorities*

Thread Priorities

- Every thread has a *priority*
- Thread defines:
 - Thread.MAX_PRIORITY (currently 10)
 - Thread.NORM_PRIORITY (currently 5)
 - Thread.MIN_PRIORITY (currently 1)
- A thread's priority must be an integer in the range Thread.MIN_PRIORITY to Thread.MAX_PRIORITY, inclusive

Thread Priorities

- When a thread creates a new `Thread` object, the new thread's priority is set equal to the priority of the thread that created it
- Thread priorities may be changed at run-time
 - `setPriority()` sets a thread's priority to a specified value
 - `getPriority()` returns the thread's priority
- The currently executing thread is (normally) the thread in the `Runnable` state with the highest priority
 - later, we'll look at what happens when two or more threads have the highest priority

Sharing the Processor Between Threads

- When the JVM schedules a thread for execution, the thread executes until:
 - it voluntarily relinquishes the processor; e.g.,
 - `Thread.yield()` causes the currently executing thread to relinquish the processor
 - `Thread.sleep()` causes the currently executing thread to temporarily cease execution for a specified amount of time
 - `Object.wait()` causes the currently executing thread to wait for a notification
 - cont'd on next slide

Sharing the Processor Between Threads

- it terminates
 - by running off the end of its `run()` method
 - by throwing an exception that propagates beyond its `run()` method
- it is preempted by a higher-priority thread becoming ready to run
- if timeslicing is supported, its timeslice expires
- This is called *preemptive, priority-based scheduling*
 - it relies on the preemption of lower-priority threads to ensure that (high-priority) threads that need processor time urgently get it

Scheduling Equal Priority Threads

- Suppose we have two ready to run Java threads, $t1$ and $t2$, with equal priority, and no higher priority thread is ready to run
- Suppose $t1$ is scheduled for execution
- What does the JVM do in this case?
- With some versions of the JVM, $t1$ will run until it relinquishes the processor, terminates, or is preempted by a higher priority thread
 - $t2$ will not be scheduled for execution until $t1$ (and any higher priority threads) are not eligible to run

Timeslicing

- Other versions of the JVM support *timeslicing*
 - thread scheduler schedules equal-priority, ready to run threads with the highest priority in a round-robin fashion, allowing each thread to run for a fixed amount of time
- The JVM will *timeslice* between $t1$ and $t2$, but will not schedule any lower priority ready to run threads until $t1$ and $t2$ are no longer eligible to run
- Either approach is permitted by the Java specification

Thread Priorities: Example

- Here is a version of the factorial/fibonacci thread example that uses priorities
 - factorialThread has priority
`Thread.NORM_PRIORITY - 1`
 - secondFactorialThread has priority
`Thread.NORM_PRIORITY - 3`
 - notice that these two threads execute the same code but have different priorities
 - fibonacciThread has priority
`Thread.NORM_PRIORITY - 2`

Thread Priorities: Example

- None of the threads relinquish the processor (the calls to `sleep()` have been removed)
- `factorialThread` has higher priority than the other two threads, so it runs to completion before the other two are executed
- `fibonacciThread` has higher priority than `secondFactorialThread`, so it runs to completion before `secondFactorialThread` is executed
- Here is the code...

Thread Priorities: Example

```
class ThreadPriorityExample
{
    public static void main(String[] args) {
        Thread factorialThread =
            new Thread(new Factorial(),
                "Factorial calculator");
        factorialThread.setPriority(
            Thread.NORM_PRIORITY - 1);
        System.out.print("Created: " +
            factorialThread + '\n');
```

Thread Priorities: Example

```
Thread secondFactorialThread =  
    new Thread( new Factorial(),  
                "Second factorial calculator");  
secondFactorialThread.setPriority(  
    Thread.NORM_PRIORITY - 3);  
System.out.print("Created: " +  
    secondFactorialThread + '\n');  
  
Thread fibonacciThread =  
    new Thread(new Fibonacci(),  
                "Fibonacci calculator");  
fibonacciThread.setPriority(  
    Thread.NORM_PRIORITY - 2);  
System.out.print("Created: " +  
    fibonacciThread + '\n');
```

Thread Priorities: Example

```
System.out.print("Starting threads\n");
factorialThread.start();
secondFactorialThread.start();
fibonacciThread.start();
    }
}
```

Thread Priorities: Example

```
/**
 * This thread calculates 0! through 10!, where
 * 0! = 1
 * n! = n * (n-1)!, n > 0
 */
class Factorial implements Runnable
{
    public void run() {
        // 0! = 1
        long factorial = 1;
        System.out.print(Thread.currentThread() +
                        ": 0! = " + factorial +
                        '\n');
        for (int n = 1; n <= 10; n++) {
```

Thread Priorities: Example

```
// n! = n * (n-1)!
factorial = n * factorial;
System.out.print(
    Thread.currentThread() +
    ": " + n + "! = " +
    factorial + '\n');
}
System.out.print(Thread.currentThread() +
    " finished\n");
}
}
```

Thread Priorities: Example

```
/**
 * This thread calculates fib(1) through fib(10),
 * where
 * fib(1) = 1
 * fib(2) = 1
 * fib(n) = fib(n-1) + fib(n-2), n > 2
 */
class Fibonacci implements Runnable
{
    public void run() {
        // fib(1) = 1
        long firstFib = 1;
        System.out.print(Thread.currentThread() +
                          ": fib(1) = " +
                          firstFib + '\n');
```

Thread Priorities: Example

```
// fib(2) = 1
long secondFib = 1;
System.out.print(Thread.currentThread() +
                  ": fib(2) = " +
                  secondFib + '\n');

for (int n = 3; n <= 10; n++) {

    // fib(n) = fib(n-1) + fib(n-2)
    long fibN = firstFib + secondFib;
    System.out.print(
        Thread.currentThread() +
        ": fib(" + n + ") = " +
        fibN + '\n');
    secondFib = firstFib;
    firstFib = fibN;
}
System.out.print(Thread.currentThread() +
                  " finished\n");
}
}
```

Relinquishing the Processor & Thread Preemption

- The next example has one higher priority factorial thread and one lower priority fibonacci thread
- Before calculating $n!$ for the current value of n , the factorial thread relinquishes the processor by sleeping for 2 ms
- While the factorial thread sleeps, the lower priority fibonacci thread executes
- When the factorial thread wakes up, it preempts the fibonacci thread
- Here is the code (Fibonacci is unchanged from slides 32-33)

Relinquishing the Processor & Thread Preemption

```
class ThreadPreemptionExample
{
    public static void main(String[] args) {
        Thread factorialThread =
            new Thread(new Factorial(),
                "Factorial calculator");
        factorialThread.setPriority(
            Thread.NORM_PRIORITY - 1);
        System.out.print("Created: " +
            factorialThread + '\n');
```

Relinquishing the Processor & Thread Preemption

```
Thread fibonacciThread =
    new Thread(new Fibonacci(),
                "Fibonacci calculator");
fibonacciThread.setPriority(
    Thread.NORM_PRIORITY - 2);
System.out.print("Created: " +
                  fibonacciThread + '\n');

System.out.print("Starting threads\n");
factorialThread.start();
fibonacciThread.start();
    }
}
```

Relinquishing the Processor & Thread Preemption

```
/**
 * This thread calculates 0! through 10!, where
 * 0! = 1
 * n! = n * (n-1)!, n > 0
 */
class Factorial implements Runnable
{
    public void run() {
        // 0! = 1
        long factorial = 1;
        System.out.print(Thread.currentThread() +
                        ": 0! = " + factorial +
                        '\n');
        for (int n = 1; n <= 10; n++) {
```

Relinquishing the Processor & Thread Preemption

```
// Sleep for 2 ms before calculating n!
try {
    Thread.sleep(2);
} catch (InterruptedException e) {}

// n! = n * (n-1)!
factorial = n * factorial;
System.out.print(
    Thread.currentThread() +
    ": " + n + "! = " +
    factorial + '\n');
}
System.out.print(Thread.currentThread() +
    " finished\n");
}
}
```

Unbounded Priority Inversion

- Unbounded priority inversion is a problem that can occur in concurrent systems that provide some form of lock to protect critical sections, and use priority-based preemptive scheduling
 - it is not unique to Java

Bounded Priority Inversion: Overview

- Suppose we have a low priority thread that has obtained a lock and entered a critical section (e.g., in Java, is executing a synchronized method)
- The lower priority thread is preempted by a higher priority thread
- The high priority thread wants to enter the critical section, so it attempts to acquire the lock and blocks, and the low priority thread is again dispatched
- The high priority thread remains blocked until the low priority thread leaves the critical section and releases the lock

Bounded Priority Inversion: Overview

- In effect, the priority of the high priority thread has been temporarily reduced to that of the low priority thread
- This is called *bounded priority inversion*
 - bounded, because we can determine the amount of time that the low priority thread will block the high priority thread from entering the critical region

Unbounded Priority Inversion: Overview

- Now suppose that while the low priority thread is in the critical section, it is preempted by one or more medium priority threads
- The lower priority thread does not execute for an indeterminate amount of time
- So, the higher priority thread that is waiting to enter the critical section also waits for an indeterminate amount of time, while the medium priority threads execute
- This is *unbounded* priority inversion

Solving the Priority Inversion Problem

- The thread scheduler could disable preemption of the lower priority thread while it is in the critical section
 - this is not the best approach, as it disables the execution of other threads (with high priority) that will not attempt to enter the critical section
- The thread scheduler could increase the priority of the lower priority thread while it is in the critical section
 - two ways to do this are commonly used:
 - priority inheritance protocol
 - priority ceiling (priority protect) protocol

Priority Inheritance Protocol

- The lower priority thread starts executing the critical section at its assigned (low) priority
- When a higher priority thread attempts to enter the critical section (e.g., invokes a synchronized method and blocks attempting to acquire object's lock), the low priority thread's priority is raised to be the same as that of the higher priority thread
- So, the lower priority thread cannot be preempted by a medium priority thread, and the time that the higher priority thread is blocked is bounded

Priority Inheritance Protocol

- The low priority thread's priority is restored to its original value when it leaves the critical section
- *Summary*: the priority of a thread holding a lock is the higher of
 - its assigned priority
 - the priority of the highest priority thread that is blocked, waiting to acquire the lock
- Many implementations of the JVM use priority inheritance to prevent unbounded priority inversion of threads waiting to execute synchronized methods

Priority Ceiling Protocol

- Locks used to protect critical sections are allocated a priority when the lock is created, called the *ceiling priority*, that is at least as high as the priority of the highest priority thread that can acquire the lock
- When a low priority thread acquires the lock, its priority is always raised to the lock's ceiling priority
- As long as the low priority thread doesn't relinquish the processor, no threads of any priority that need to acquire the lock will execute while the low priority thread is in the critical section

Priority Ceiling Protocol

- Also, the low priority thread can't be preempted by medium priority threads (threads with priority $<$ the lock's ceiling priority) that aren't interested in the critical section
- The low priority thread's priority is restored to its original value when it leaves the critical section
- The priority ceiling protocol is not currently supported by Java

Comparison of the Protocols

- Priority inheritance is managed transparently by the system
- Priority ceiling requires the programmer to configure the priority ceilings of the locks
- Priority inheritance does not associate a priority with a critical section (priority promotion of the low priority thread executing the critical section occurs only when a higher priority thread attempts to acquire the lock)
- Priority ceiling associates a priority with all locks, and therefore with the critical sections that they protect

JVM Implementations - Green Threads

- The JVM runs on top of an operating system that is unaware of Java threads
 - the JVM is completely responsible for thread management; e.g., for every thread, the JVM maintains a stack, program counter, and other bookkeeping info
 - context switching between threads is handled by the JVM
- This approach is equivalent to using a *user-level* threading package with C - no calls to the underlying operating system are required to manage threads

JVM Implementations - Green Threads

- Early implementations of the JVM on many Unix platforms used the green-thread model
- Green-thread implementations support priority-based, preemptive thread scheduling, but most implementations do not support timeslicing
- The *reference implementation* of the green-thread JVM uses priority inheritance

JVM Implementations - Native Threads

- Java threads are implemented using the threads supported by the operating system that hosts the JVM
 - thread scheduling is handled by the operating system
 - differences between operating systems lead to subtle differences in Java thread scheduling

JVM Implementations - Windows Native Threads

- For most JVM implementations for 32-bit Windows operating systems, there is a one-to-one mapping between Java threads and Win-32 threads
 - these threads are scheduled by the operating system
- Priority-based, preemptive scheduling is supported
 - 10 Java thread priorities are mapped onto 6 Win-32 priorities
- Priority inheritance is provided
- Equal priority threads timeslice
- Thread priorities are temporarily increased (quietly) to reduce starvation

Other JVM Implementations

- Solaris (Sun) implementation uses Solaris native threads
 - native threads use both user-level threads and system-level threads (LWPs)
 - scheduling is complex and beyond the scope of this course
- Embedded systems - Java threads normally mapped to the native system threads/tasks/processes supported by the real-time operating system (RTOS) hosting the JVM
 - Java thread scheduling = RTOS thread scheduling