Concurrent Programming in Java

Vaidė Narváez

Computer Information Systems

July 12th, 2010

based on Net-Centric Computing@USI slides from W.Binder and A.Murphy

PAYAP UNIVERSITY

Processes

- Most modern operating systems support multiple, concurrent processes, called *multitasking*
- Processes allow applications run in parallel
- Typically OS + programming language solutions

PAYAP UNIVERSITY

Threads

A *thread* is an operating system abstraction of activity, task, multiple points of execution inside a single process

- Each thread executes sequentially
- There is an external scheduler that switches among threads
- Threads communicate by writing/reading to/from shared variables

PAYAP UNIVERSITY

Race conditions

Thread 1

```
A: if (x==0)
B: x=x+1
```

Thread 2

```
C: if (x==0)
D: x=10
```

What are the possible results of the execution, assuming $x = 0$ initially?

PAYAP UNIVERSITY

Race conditions

Can the following cause race conditions?

- `if (x==0) x=10`
- `x = x + 1`
- `x = 100`

PAYAP UNIVERSITY

Critical section

A *critical section* is a portion of a program that uses a shared resource.

- Critical sections must be **mutually exclusive**: only one thread can be in a critical section at a time.
- Mutual exclusion can be achieved by placing locks around the critical section. (`lock.acquire, lock.release`)

PAYAP UNIVERSITY

Concurrency in Java

Concurrency is tricky to deal with!

Thread scheduling is platform dependent

- What you observe on your laptop may be different from what you can see on my laptop
- DO NOT rely on timing for correctness
- In fact, the timing behavior often changes from version to version of the JDK, without Sun telling you!

Two ways to create threads in Java

`extends Thread`

Inherit from java.lang.**Thread**

Name the thread.
For your benefit only

You must override **run(),**
thread behavior goes in here

Instantiate a new thread…

…and start its execution

```
public class MyThread extends Thread {
  public MyThread(String name) {
    super(name);
  }
  public void run() {
    for(int x=0; x<20; x++)
      System.out.println(this.getName()+x);
  }
}

public class ThreadFun {
  public static void main(String[] args) {
    MyThread t1,t2;
    t1=new MyThread("Thread 1");
    t2=new MyThread("Thread 2");
    t1.start();
    t2.start();
  }
}
```

Two ways to create threads in Java

`implements Runnable`

Implement the **`Runnable`** interface

`run()` must be implemented

Use **`currentThread()`** to get a handle to the running thread

First create the runnable object

Then assign it to a new thread

Then start the thread

```
public class MyRunnable implements Runnable
{
   public void run() {
      for(int x=0; x<20; x++)
        System.out.println(
          Thread.currentThread().getName()+x);
   }
}

public class ThreadFun {
  public static void main(String[] args) {
    Thread t1, t2;
    Runnable r1, r2;
    r1 = new MyRunnable();
    r2 = new MyRunnable();
    t1 = new Thread(r1, "Thread 1");
    t2 = new Thread(r2, "Thread 2");
    t1.start();
    t2.start();
  }
}
```

Threads

- `new Thread(new MyRunnable(), name).start()`
- Affecting scheduler behavior
  - `yield();`
    - Indicates that the thread is willing to give up the processor
    - NOT guaranteed to behave this way
  - `sleep(n); // n in milliseconds, 1000ms=1s`
    - Cease execution for some time
    - Note, can be interrupted, so need an exception block
  - `setPriority(n);`
    - Control relatively how often a thread is selected for execution
    - `n = {MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY}`

PAYAP UNIVERSITY

Resource sharing

```
public class RGBColor {
    private int r;
    private int g;
    private int b;

    public void setColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r;
        this.g = g;
        this.b = b;
    }
```

- Consider creating two threads, red and blue: each tries to set the same RGBColor instance to its color
- Due to interleaving, the color may end up purple
- Formally, this is called a write/write conflict

PAYAP UNIVERSITY

Synchronized blocks

- Block synchronization takes an argument indicating which object to lock on

```
synchronized (myObject) {
    // Lock is held
    …
}
// Lock is released
```

- Prevents multiple threads from being inside blocks synchronized on the same object
- Used to protect part of a method
- This can be placed anywhere in your code: You must remember to explicitly lock the same object each time you need access protection

PAYAP UNIVERSITY

Synchronized methods

- Entire methods can be declared synchronized

```
synchronized void f() {...}
```

- This is the same as locking the object in which the method is declared

```
void f() { synchronized(this) {...} }
```

- If some methods of an object are synchronized and others not, only the synchronized methods are mutually exclusive. Others can execute in parallel

PAYAP UNIVERSITY

Acquiring and releasing locks

- Locks are acquired atomically
- Once a thread holds a lock, it can enter another block synchronized on that object (intrinsic locks are reentrant)
- Locks are released when the block is exited (Even if the exit is due to an exception)
- Non-synchronized methods can be executed in parallel to synchronized blocks

`synchronized` keyword

- The `synchronized` keyword is not considered part of a method's signature:
  - the synchronized modifier is not automatically inherited when subclasses override superclass methods
  - methods in interfaces cannot be declared as synchronized
- Synchronization in an inner class method is independent of its outer class
  - however, a non-static inner class method can lock its containing class, say `OuterClass`, via code blocks using:
    `synchronized(OuterClass.this) { /* body */ }`
- Static synchronization employs the lock possessed by the `Class` object associated with the class the static methods are declared in.
  - The static lock for class C can also be accessed inside instance methods via: `synchronized(C.class) { /* body */ }`
  - Static locks of superclass and subclass are different

PAYAP UNIVERSITY

Key Rules

- Always lock during updates to object attributes

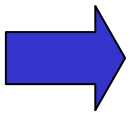
```
synchronized (point) {
  point.x = 5;
  point.y = 7;
}
```

- Always lock during access of possibly updated object

```
synchronized(point) {
  if (point.x > 0) {
          ...
}}
```

- You do not need to synchronize stateless parts of methods

```
synchronized void service() {
  state = ...; // update state
  operation();
 }
```



```
void service() {
  synchronized(this) {
    state = ...; // update state
  }
  operation();
}
```

- Avoid locking when invoking methods on other objects

PAYAP UNIVERSITY

Assignments

- Assigning variables is atomic
  - Except for `long` or `double`
- However, threads are allowed to hold the values of variables in local memory (e.g., a register)
  - This means one thread can change the value and another may not see it, or may see changes in surprising order
  - Declaring the variable `volatile` forces it to be read/written from/to memory upon each access

PAYAP UNIVERSITY

Car Park example

PAYAP UNIVERSITY

CarParkControl Monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() {
    …  --spaces; …
    }

  synchronized void depart() {
    … ++spaces; …
    }
}
```

mutual exclusion
synch methods

block if full?
(spaces==0)

block if empty?
(spaces==capacit

PAYAP UNIVERSITY

Object Synchronization

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's lock

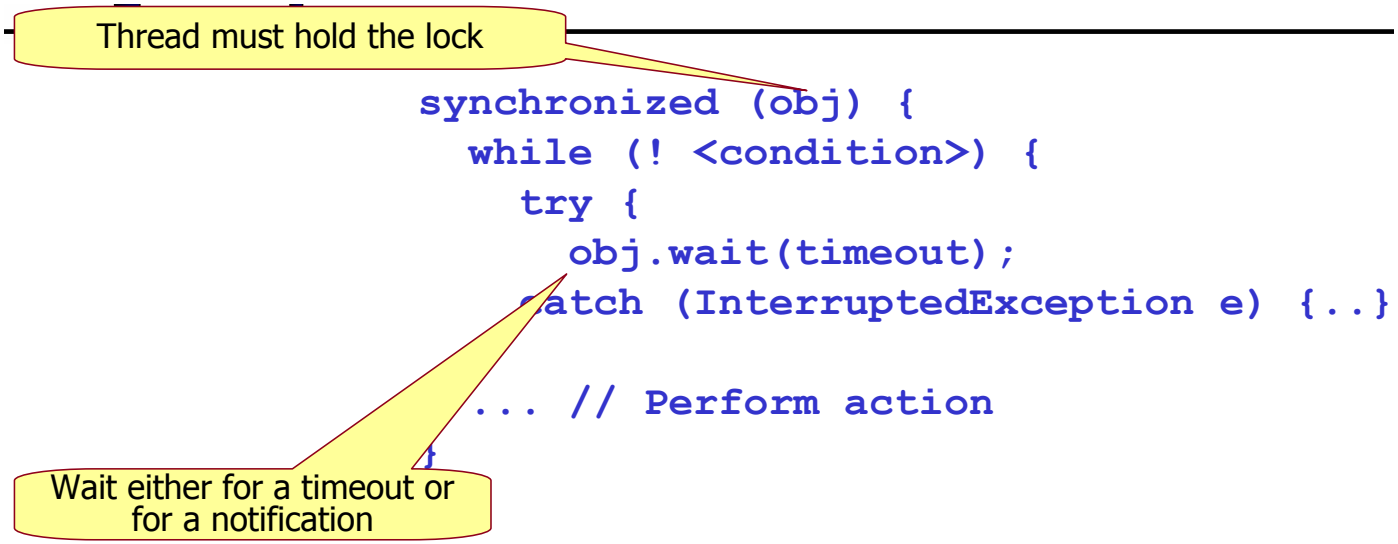
```
public final void notifyAll()
```

Wakes up all threads that are waiting on this object's lock

```
public final void wait()
    throws InterruptedException
```

Waits to be notified by another thread. The waiting thread releases this object's lock. When notified, the thread must wait to reacquire the lock before resuming execution

wait



When notification arrives, waiting thread is put back into ready queue with all other threads, and MUST check condition again before acting.

CarParkControl Monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notify();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notify();
  }
}
```

Synchronized, therefore holding lock of CarParkControl instance

Wait while no spaces available

Notify a car is available to leave

Wait while no cars inside

Notify any waiting threads that a space is available

Could use notifyAll(), but only one car can possibly enter

PAYAP UNIVERSITY

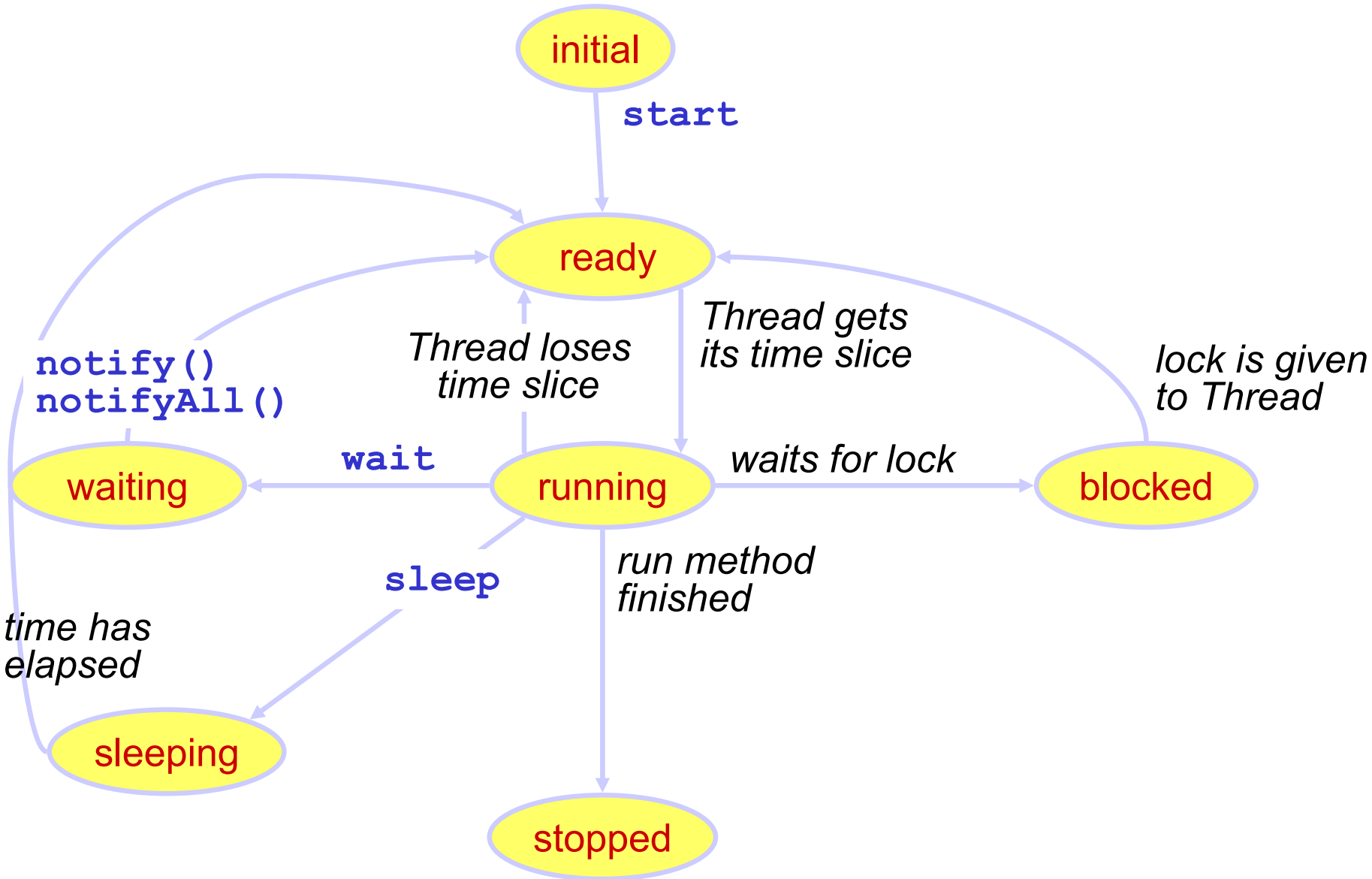
Single notifications

- You can reduce the context-switch overhead associated with notifications by using a single `notify` rather than `notifyAll`

- Single notifications can be used to improve performance when you are sure that at most one thread needs to be awoken. This applies when:
  - all possible waiting threads are necessarily waiting for conditions relying on the same notifications, usually the exact same condition
  - each notification will enable at most a single thread to continue. Thus, it would be useless to wake up others

PAYAP UNIVERSITY

Thread State Transitions



PAYAP UNIVERSITY

Recap

- Changes in the state of the monitor are signaled to waiting threads using `notify()` or `notifyAll()`
- The monitor is related to the object instance which is used to communicate among threads
- The lock must always be acquired before wait is invoked:

```
synchronized(lock) {
      ...
      lock.wait();
}
```

- Always re-check condition, after a wait:

```
synchronized(lock) {
  while(!cond) {
    lock.wait();
  }
}
```

YES

```
synchronized(lock) {
  if(!cond) {
    lock.wait();
  }
}
```

NO