CS601: Principles of Software Development

Multithreading. Synchronization.

Olga A. Karpenko

Parts of this presentations are based on the materials of Prof. Engle.

Announcements

- Lab 2 is due on Friday night
- Go to Bhargavi's office hours 2:15-4:15
- Instructor available 3:20-3:35 and after 5pm
- Read chapter 26 in Deitel&Deitel
- Tests: separate tests files for each class

Creating Threads in Java

```
public class Task implements Runnable {
     public void run() {
          // work for the thread
     }
}

// In another class:
Thread t = new Thread(new Task());
t.start();
```

Class Thread

```
public class Thread {
    public Thread(Runnable R);
    public Thread(Runnable R, String name);
    public void start(); // begin thread execution
    public String getName();
    public void interrupt();
    public boolean isAlive();
    public void join();
    public void setDaemon(boolean on);
    public void setName(String name);
    public void setPriority(int level);
    public static Thread currentThread();
    public static void sleep(long milliseconds);
}
```

Creating Threads: Alternative

- Extend the Thread class and override the run() method

```
public class MyThread extends Thread {
    public void run() {
        // work for thread
    }
}
// in main:
MyThread t = new MyThread();
t.start();
```

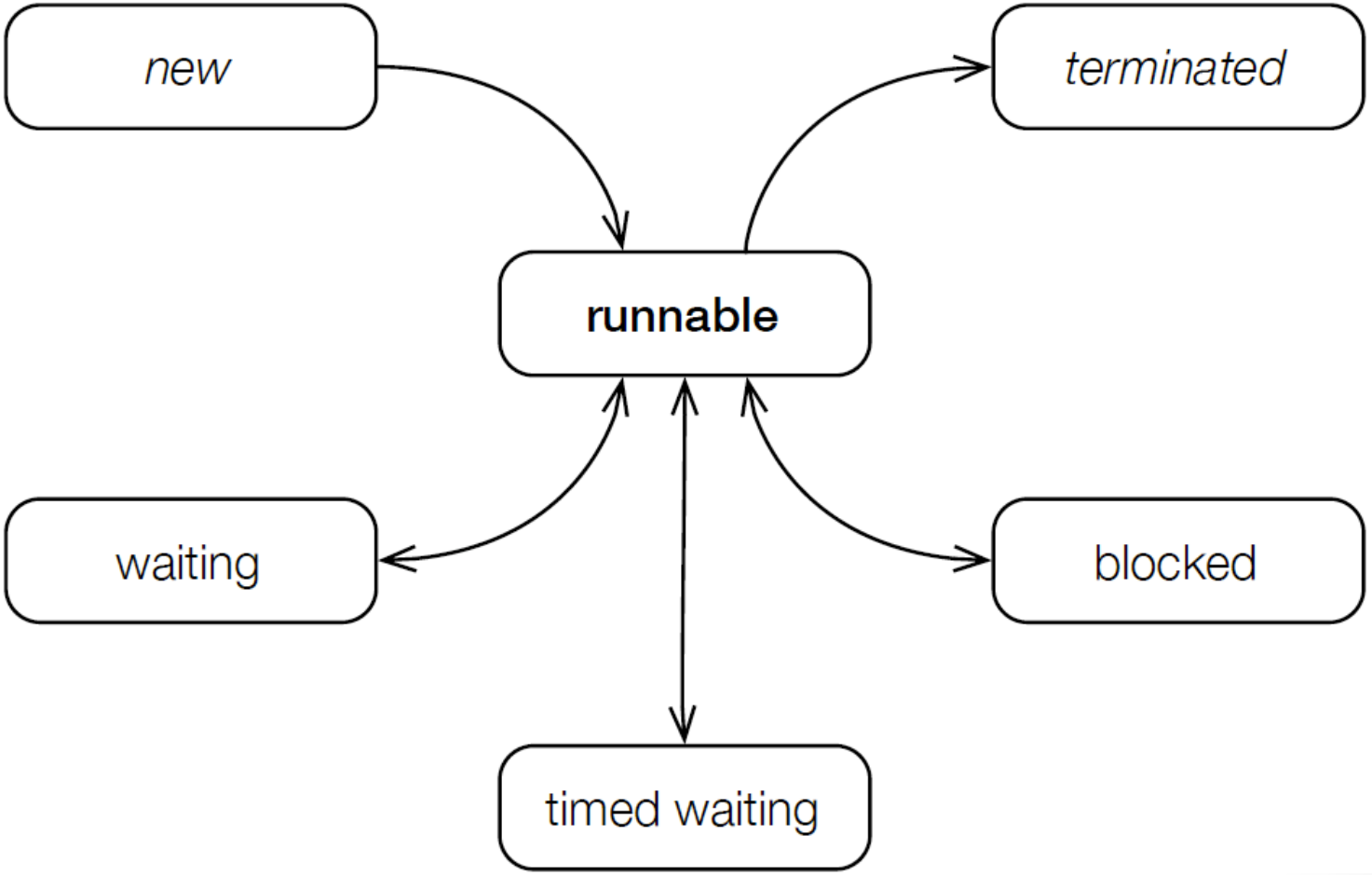
Creating Threads: Alternative

- Extend the Thread class and override the run() method
  - Not recommended
- The methods of the worker class and the Thread class get all tangled up

Example

- `SolvingForMax` – find a maximum in the large matrix
  - multithreaded version
  - each thread will find a max in one row

Thread States



Thread States

- First created -> new state
- Ready to start working -> runnable
- Finished working -> terminated
- Voluntarily decides to wait (usually for an event) -> waiting
- Wants to wait for a set amount of time -> timed waiting
- Involuntarily blocked (usually from accessing a locked resource) -> blocked

Transitions Between States

- By invoking methods in Thread and Object
- Thread Class
  - –start(), join(…), sleep(…), interrupt()
- Object Class
  - – notify(), notifyAll(), wait(…)
- Other (external) events
  - Scheduler, I/O, etc …

Scheduling

- Scheduler
  - Determines which runnable threads to run
  - Can be based on thread priority
  - Part of OS or Java Virtual Machine (JVM)

Thread Safety

- An object is thread safe if it maintains a consistent state even when accessed concurrently
- Examples: Thread Safe
  - Immutable objects, e.g. constants
  - Some mutable objects, e.g. StringBuffer
- Examples: Not Thread Safe
  - Some mutable objects, e.g. arrays, ArrayList

Synchronization

Motivation

- One thread increments x
- One thread decrements x
- Race condition if no synchronization
- See `NoSynchronization.java`

Race Condition

- When 2+ threads try to change shared data at the same time , and the result depends on the sequence of execution

Motivation

#	Thread 1: x++;	Thread 2: x--;
1	read value of x	read value of x
2	calculate x + 1	calculate x − 1
3	assign x to calculated result	assign x to calculated result

The image is courtesy of Prof. Engle.

Motivation

#	Thread 1: x++;	Thread 2: x--;
1	read	
2	calculate	
3	assign	
4		read
5		calculate
6		assign

The image is courtesy of Prof. Engle.

Motivation

#	Thread 1: x++;	Thread 2: x--;
1	read x = 1	
2	calculate 1 + 1 = 2	
3	assign x = 2	
4		read x = 2
5		calculate 2 – 1 = 1
6		assign x = 1
	final value x = 1	

The image is courtesy of Prof. Engle.

Motivation

#	Thread 1: x++;	Thread 2: x--;
1	read	
2		read
3	calculate	
4		calculate
5	assign	
6		assign

The image is courtesy of Prof. Engle.

Motivation

#	Thread 1: x++;	Thread 2: x--;
1	read x = 1	
2		read x = 1
3	calculate 1 + 1 = 2	
4		calculate 1 – 1 = 0
5	assign x = 2	
6		assign x = 0
	final value x = 0	

The image is courtesy of Prof. Engle.

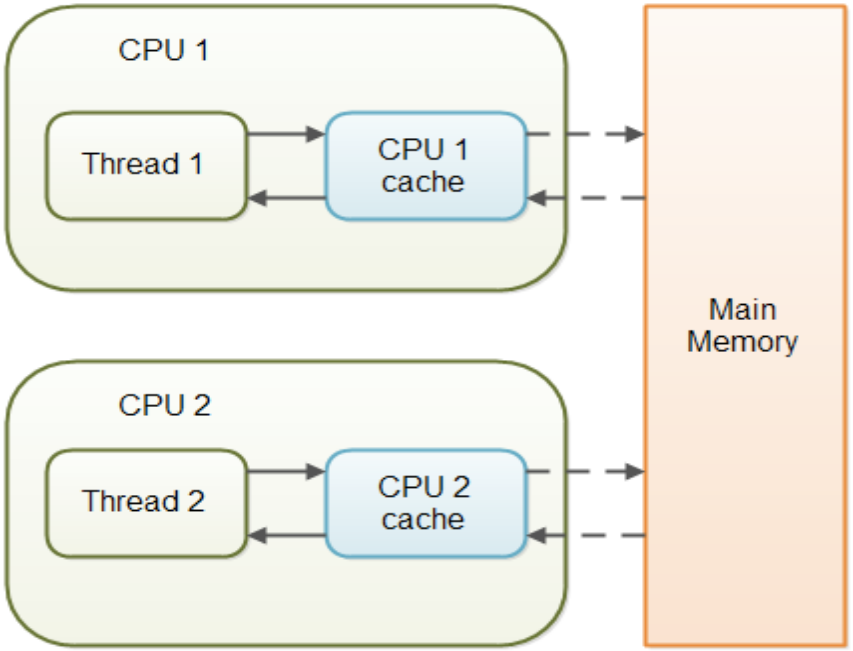
Problems

- Atomicity
  - Operations x++ and x-- are not *atomic*
- Visibility
  - Shared data modified between read & use

An Atomic Operation

- All operations succeed or all operations fail
- No incomplete results

Problems: Visibility



- In multithreading: Shared data may be modified between read & use

From: http://tutorials.jenkov.com/java-concurrency/volatile.html

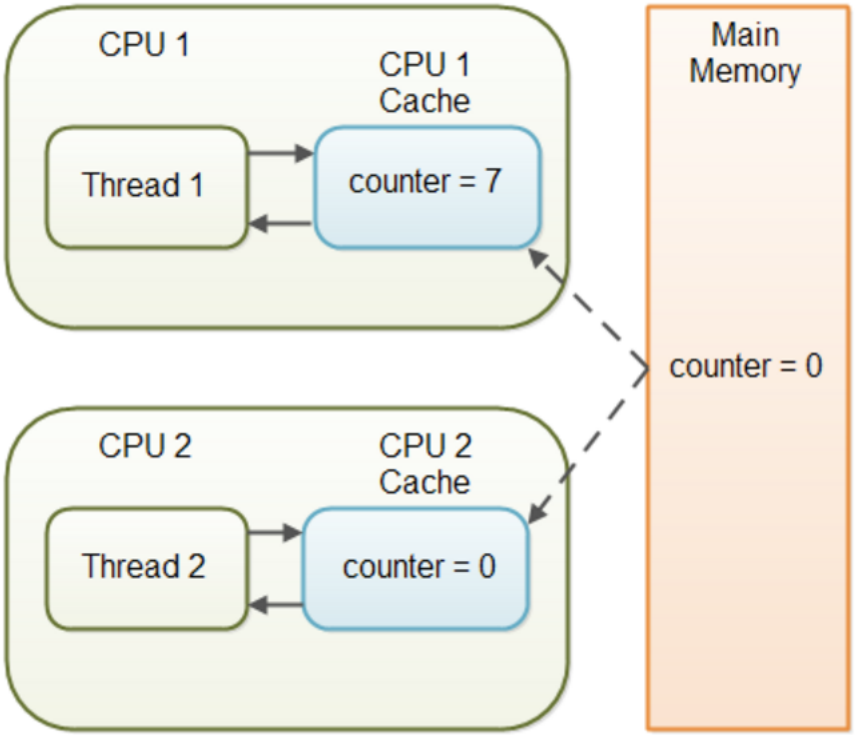
Example

```
public class SharedObject {

    public int counter = 0;

}
```

- Thread 1 modifies counter
- Threads 1 and 2 can both read counter

From: http://tutorials.jenkov.com/java-concurrency/volatile.html

Example: Visibility Issue



- Thread 1 changed `counter`
- `counter` has not been updated in the main memory
- Thread 2 reads the wrong value

Solution

- Must provide synchronization
- Synchronization in Java
  - Volatile variables
  - Synchronized code blocks and methods
  - Custom lock objects

Synchronization in Java

- Volatile variables
- Synchronized code blocks and methods
- Custom lock objects

Volatile

- All "writes" are written directly to the main memory
- All "reads" read from the main memory, not from cache
- Threads always read the latest value
- Guarantees visibility but not atomicity

From: http://tutorials.jenkov.com/java-concurrency/volatile.html

Volatile

```
public class SharedObject {

    public volatile int counter = 0;

}
```

From: http://tutorials.jenkov.com/java-concurrency/volatile.html

Volatile

- "Lightweight" synchronized
  - Simple
  - Never causes blocking
- The catch: can only be used in limited # of cases

Happens Before Guarantee

- Read and Write instructions on volatile variables can not be reordered
- Example:
  - Blue instructions will happen *before* statement with volatile
  - Green instructions will happen after

```
sharedObject.nonVolatile1 = 123;
sharedObject.nonVolatile2 = 456;

sharedObject.volatileVar  = true;

int someValue1 = sharedObject.nonVolatile3;
int someValue2 = sharedObject.nonVolatile4;
```

From: http://tutorials.jenkov.com/java-concurrency/volatile.html

Functionality

- Write operations can not depend on value
  - e.g. okay `volatileVar = true;`
  - e.g. not okay `volatileVar = volatileVar + 1`

Question: Why is this not ok?

```
volatileVar = volatileVar + 1
```

Functionality

- Write operations can not depend on value
  - e.g. okay `volatileVar = true;`
  - e.g. not okay `volatileVar = volatileVar + 1`

Question: Why is this not ok?

```
volatileVar = volatileVar + 1
```

Race condition in the time period between read and write – multiple threads might try to write

Functionality

- Variable cannot be used in invariants with other variables
  - e.g. Okay:

```
İf (volatileVariable == true)
```

  - e.g. not okay:

```
if (volatileVariable < otherVariable)
```

Use Pattern #1: Status Flag

- Read does not depend on other variables
- Write does not depend on the current value
- One state transition typically

Use Pattern #1: Status Flag

```
volatile boolean shutdownRequested;

// other code

public void shutdown() {
     shutdownRequested = true;
}

public void doWork() {
    while (!shutdownRequested) {
        // do stuff
    }
}
```

Use Pattern #1

- Example: `CalculatePrimes.java`
- Calculate as many primes as we can in ten seconds
  - One thread calculates prime numbers
  - Another one is the “timer”

Other Patterns

- http://www.ibm.com/developerworks/library/j-jtp06197/
- http://tutorials.jenkov.com/java-concurrency/volatile.html