Threads - Chapter 13

- Threads allow a program to "do more than one thing at a time."
- If the hardware has multiple processing units, then this can result in faster execution of a program.
- In distributed programming, the distinct tasks to be performed may happen on different physical computers (often called a client and a server).

1

Concurrent Programming Needs

- Concurrency - It must be possible to create multiple execution threads (or processes or tasks or whatever they are called).
- Synchronization - It must be possible for two threads to synchronize so that they do not interfere with each other.
- Communication - The threads must be able to share information.

2

The Implicit AWT Thread

- Every GUI program built using Swing and AWT has two threads.
- The main thread is the one executing main(). It often terminates after the GUI is built.
- The event thread detects GUI events and invokes the appropriate listener methods, and methods such as paint().

3

```
//AwtThread.java - doing two things at once
import java.awt.*;
import javax.swing.*;
import tio.*;
class AwtThread {
  public static void main(String[] args)
       throws InterruptedException
  {
    createGUI();
    int count = 0;
    while (true) {
      count++;
      // go to sleep for 1 second = 1000 milliseconds
      Thread.currentThread().sleep(1000);
      System.out.println("count is now " + count);
      System.out.flush(); // force output to print now
    }
  }
```

4

```
  static void createGUI() {
    JFrame frame = new JFrame("AwtThread");
    Container pane = frame.getContentPane();
    JButton quit = new JButton("Quit");
    quit.addActionListener(new GoodBye());
    pane.add(quit, BorderLayout.NORTH);
    JButton counter = new JButton("Click to count");
    counter.addActionListener(new ClickCounter());
    pane.add(counter, BorderLayout.SOUTH);
    frame.pack();
    frame.show();
  }
}
```

5

```
//ClickCounter.java - count button clicks
import java.awt.event.*;

class ClickCounter implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    count++;
    System.out.println("Total clicks is " + count);
  }
  int count = 0;
}
```

6

Creating Your Own Threads

```
//TwoThreads.java - create two simple threads
class TwoThreads{
  public static void main(String[] args) {
    SimpleThread t1 = new SimpleThread(1, 1000);
    SimpleThread t2 = new SimpleThread(2, 1300);
    t1.start();
    t2.start();
  }
}
```

7

```
//SimpleThread.java - periodically print a message
class SimpleThread extends Thread {
  private int id;
  private int delay;

  SimpleThread(int threadId, int threadDelay) {
    id = threadId;
    delay = threadDelay; // in milliseconds
  }
```

8

```
  public void run() {
    System.out.println("Thread" + id + " started.");
    System.out.flush(); //needed to see the effect
    for (int i = 0; i < 10; i++) {
      try {
        sleep(delay); // sleep delay milliseconds
      }
      catch (InterruptedException e) {
        System.out.println("sleep interrupted: "+e);
      }
      System.out.println("Thread" + id + ": i = " +i);
    }
    System.out.println("Thread" + id + " finished.");
  }
}
```

9

Communication Between Threads

- Threads within the same JVM can communicate by each having a reference to a shared object.
- If one changes a field in the object, the other can view the updated field.

10

```
//TwoThreads2.java - two threads sharing a counter
class TwoThreads2 {
  public static void main(String[] args) {
    Counter counter = new Counter(0, 1000000);
    Racer t1 = new Racer(1, counter);
    Racer t2 = new Racer(2, counter);
    t1.start();
    t2.start();
  }
}
```

11

```
//Racer.java - click the counter 100000 times
class Racer extends Thread {
  Racer(int id, Counter counter) {
    this.id = id;
    this.counter = counter;
  }
  public void run() {
    System.out.println("Thread" + id + " started.");
    for (int i = 0; i < 1000000; i++) {
      counter.click();
    }
    System.out.println("Thread" + id +
            " finished counter is " + counter.get());
  }
  private int id;
  private Counter counter;
}
```

12

Synchronizing

- If you run TwoThreads2, you will probably not get the "expected" final result of 2,000,000.
- Why not?

13

A Critical Section

- The statement
  `value = (value + 1) % modulus`
  is compiled into a sequence of bytecodes similar to:

```
load value
load 1
add
load modulus
mod
store value
```

- That sequence is a *critical section*, it must be completed without interference.

14

Context Switches and Races

- Consider this sequence of events:

Thread 1	Thread 2	value
Load value		0
	Load value	
	Store value	1
	Load value	
	Store value	2
	Load value	
Store value		1

15

synchronized

- A synchronized method is a critical section.
- ```
  //in class Counter
  public synchronized void click() {
    value = (value + 1) % modulus;
  }
  ```

16

synchronized methods

- Two threads *cannot* both be executing a synchronized method for a particular object, at the same time.
- Synchronization is on a **per object basis**.
- If a class has two synchronized methods. You cannot have one thread in one method and a different thread in the other method.
- If you have two objects, one thread can be in a method for one object, and another thread in the same (or different method) for the other object.

17

Locks

- Every ***object*** (not class) can be thought of as having a lock.
- When a thread invokes a synchronized method, it must first lock the lock for the object, having found the lock unlocked.
- If another thread trys to lock the locked lock, it is blocked until the lock is unlocked.
- We usually refer to "locking the lock" as *obtaining the lock*, and "unlocking the lock" as *releasing the lock*.

18
  ```

Signal-Wait Synchronization

- Mutual exclusion synchronization (critical sections) allows us to keep two threads from interfering with each other. But what about making one thread wait for something to get finished?
- For this we use signal-wait synchronization, or in Java wait-notify.
- For example, one thread is filling a buffer, and another thread is reading values out of the buffer. This is called the producer-consumer problem.

19

Producer-Consumer

- How do we make sure that the consumer doesn't try and consume items from an empty buffer, and that the producer doesn't try to produce items and place them in an already full buffer (that has yet to be consumed).

20

Notice the shared Buffer object in this example.

```
//ProducerConsumer.java - signal/wait synchronization
class ProducerConsumer {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();
    Producer prod = new Producer(buffer);
    Consumer cons = new Consumer(buffer);
    prod.start();
    cons.start();
  }
}
```

21

```
//Producer.java - "produce" the integers 0-9
class Producer extends Thread {
  Producer(Buffer buf) {
    buffer = buf;
  }
  public void run() {
    System.out.println("Producer started.");
    for (int i = 0; i <  10; i++) {
      // code to produce a value here
      System.out.println("Producer produced " + i);
      buffer.put(i); // let i be the produced value
    }
    System.out.println("Producer is finished.");
  }
  private Buffer buffer;
}
```

22

```
//Consumer.java - consume first 10 integers from buffer
class Consumer extends Thread {
  Consumer(Buffer buf) {
    buffer = buf;
  }
  public void run() {
    System.out.println("Consumer started.");
    for (int i = 0; i < 10; i++) {
      int value = buffer.get();
      System.out.println("Consumer received "+value);
      // code to "consume" the value here
    }
    System.out.println("Consumer is finished.");
  }
  private Buffer buffer;
}
```

23

```
//Buffer.java - a synchronized shared buffer
class Buffer {
  synchronized void put(int newValue) {
    while (!empty)
      try {
        //wait for previous value to be consumed
        wait();
      }
      catch(InterruptedException e) {
        System.out.println("wait interrupted: " + e);
      }
    value = newValue;
    empty = false;
    notify();
  }
```

24

```
  synchronized int get() {
    while (empty)
      try {
        wait(); // wait for buffer to fill
      }
      catch(InterruptedException e) {
        System.out.println("wait interrupted: "+e);
      }
    empty = true;
    notify();
    return value;
  }
  private boolean empty = true;
  private int value;
}
```

25

Monitors for Synchronizing

- Synchronization in Java is based on a technique called *monitor synchronization*.
- An object with synchronized methods *monitors* the execution of those methods, allowing only one thread at a time to execute.
- Some monitor variations allow threads to synchronize on *condition variables*. That is a thread waits until a condition is true.

26

Condition Variables

- Java doesn't have condition variables. In order to follow the monitor-with-condition-variables style of synchronization, the programmer must explicitly check for "the condition" after a thread returns from a wait().
- That is why we used

  `while (!empty) ...`

  instead of

  `if (!empty) ...`

27

Semaphores

- Another important synchronization method uses *semaphores*.
- Semaphores look a lot like Java's locks with wait() and notify().
- They are very different. Semaphores can count.

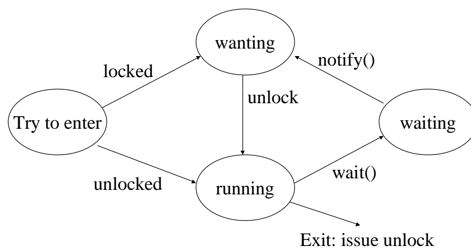
28

Semaphore’s signal and wait

- A semaphore has an internal counter.
- A wait decrements the internal counter and waits if the counter is less than 0.
- A signal increments the internal counter and wakes up a waiting thread if the counter is less than or equal to 0.
- Java's wait() and notify() have no counter. A wait() always waits, and a notify() goes unnoticed if no thread is waiting.

29

Synchronization States



30

Sockets

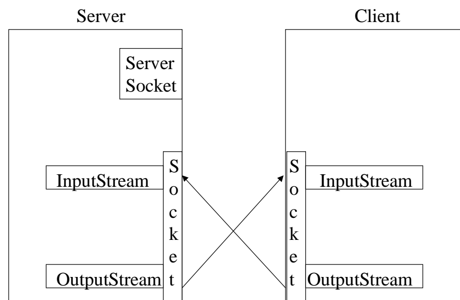
- Sockets are the basic mechanism used by two computers to communicate over the Internet.
- They are the fundamental building block for other distributed computing models.
- They are easy to use.

Socket Overview

- A server begins listening on a port.
- A client attempts to connect to a given port on a given computer (Internet host or IP address).
- If successful both the server and the client will have an InputStream and an OutputStream.

32

A Connected Server and Client



33

Creating the server socket

```
ServerSocket sock=null;
try {
   sock = new ServerSocket(portnum);
} catch (IOException e) { }
System.out.println("Now listening at port " + portnum);
Socket clientSocket = null;
try {
   clientSocket = sock.accept();
} catch (IOException e) { }
InputStream in = clientSocket.getInputStream();
OutputStream out = clientSocket.getOutputStream();
System.out.println("Connection established.");
```

Creating the client socket

```
Socket sock = new Socket(hostname,portnum);
InputStream input = sock.getInputStream();
OutputStream output = new sock.getOutputStream();
```

```
//MiniServer.java - server that echos what it receives
import java.io.*;
import java.net.*;

class MiniServer{
  public static void main (String args[])
        throws java.io.IOException
  {
    if (args.length != 1) {
      System.out.println("Usage: " +
          "java MiniServer portnumber");
      System.exit(1);
    }
    int portnum = Integer.parseInt(args[0]);
```

36

```
    ServerSocket sock = null;
    try {
      sock = new ServerSocket(portnum);
    }
    catch (IOException e) {
      System.out.println("Could not listen on port: "
                         + portnum + ", " + e);
      System.exit(1);
    }
    System.out.println("Now listening at port " +
                       portnum);
```

37

```
    Socket clientSocket = null;
    try {
      clientSocket = sock.accept();
    }
    catch (IOException e) {
      System.out.println("Accept failed: " +
                               portnum + ", " + e);
      System.exit(1);
    }
    // A client has tried to connect
```

38

```
    BufferedReader input = new BufferedReader(
        new InputStreamReader(
          clientSocket.getInputStream()));
    PrintWriter output =
      new PrintWriter(clientSocket.getOutputStream());
    System.out.println("Connection established.");

    int i = 0;
    String line = input.readLine();
    while (line!=null) {
      System.out.println(line);
      i++;
      output.println("line " + i + ":" + line);
      output.flush();
      line = input.readLine();
    }
  }
}
```

39

```
//MiniClient.java - simple client for MiniServer
import java.io.*;
import java.net.*;
import tio.*;

class MiniClient{
  public static void main (String args[])
        throws java.io.IOException
  {
    if (args.length != 2) {
      System.out.println("Usage: " +
          "java MiniClient hostname portnumber");
      System.exit(0);
    }
    int portnum = Integer.valueOf(args[1]).intValue();
    Socket sock = new Socket(args[0], portnum);
```

40

```
    BufferedReader input = new BufferedReader(
      new InputStreamReader(sock.getInputStream()));
    PrintWriter output =
      new PrintWriter(sock.getOutputStream());
    System.out.println("Connection established.");
    System.out.println("type some characters then" +
        " return:");
    String line = Console.in.readLine();
    while (line != null) {
      output.println(line);
      output.flush();
      line = input.readLine();
      System.out.println("got back:" + line);
      System.out.println("type some characters: " );
      line = Console.in.readLine();
    }
  }
}
```

41

A Multithreaded Server

- The ServerSocket is only used to create the Socket.
- The call sock.accept() returns a Socket that is uesd to connect to the client.
- The server can accept more connections by issuing another sock.accept() call.
- We need a thread to manage each connection.

42

```
//MultiServer.java - a multithreaded server
import java.io.*;
import java.net.*;

class MultiServer {
  public static void main (String args[])
        throws java.io.IOException
  {
    int portnum = Integer.parseInt(args[0]);
    ServerSocket sock = null;
    try {
      sock = new ServerSocket(portnum);
    }
    catch (IOException e) {
      System.out.println("Could not listen on port: "
                          + portnum + ", " + e);
      System.exit(1);
    }
```

43

```
    System.out.println(
            "Now listening at port " + portnum);
    Socket clientSocket = null;
    while (true) {
      try {
        clientSocket = sock.accept();
      }
      catch (IOException e) {
        System.out.println("Accept failed: "
                           + portnum + ", " + e);
        System.exit(1);
      }
      WorkerThread worker =
               new WorkerThread(clientSocket);
      worker.start();
    }
  }
}
```

44

```
//WorkerThread.java - handle one connection
import java.io.*;
import java.net.*;

class WorkerThread extends Thread {
  WorkerThread(Socket socket)
  {
    clientSocket = socket;
    workerNumber++;
    number = workerNumber;
  }
  public void run()
  {
    try {
      BufferedReader input = new BufferedReader(
          new InputStreamReader(
                    clientSocket.getInputStream()));
      PrintWriter output = new PrintWriter(
                    clientSocket.getOutputStream());
```

45

```
      System.out.println("Connection " +
                         number + " established.");

      String line = input.readLine();
      while (line != null) {
        System.out.println(line);
        output.println("worker " + number + ":"+line);
        output.flush();
        line = input.readLine();
      }
    }
    catch (IOException e) {
      System.out.println( e );
    }
    System.out.println("worker " + number+" exiting");
  }
  private Socket clientSocket;
  private static int workerNumber = 0;
  private int number;
}
```

46

A Synchronized Statement

- wait() and notify() are inherited from Object.
- It is a runtime error to invoke wait() or notify for an object, when the current thread does not hold the lock for the object.
- Entering a synchronized method obtains the lock.
- You can also obtain the lock for any object using a synchronized statement.

47

`synchronized` (*Expression*) *Block*

Expression must evaluate to a reference (i.e. anything but a primitive type).

```
String s = "Wait for this?";
…
synchronized ( s ) {
    s.wait();
}
```

48

```java
//Racer2.java - assume click() is not synchronized
class Racer2 extends Thread {
  Racer2(int id, Counter counter) {
    this.id = id;
    this.counter = counter;
  }
  public void run() {
    System.out.println("Thread" + id + " started.");
    for (int i = 0; i <  1000000; i++) {
      synchronized(counter) {
        counter.click();
      }
    }
    System.out.println("Thread" + id +
            " finished counter is " + counter.get());
  }
  private int id;
  private Counter counter;
}
```

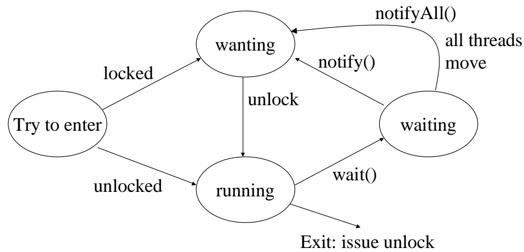
49

notifyAll()

- obj.notify() wakes up at most one thread that is waiting on obj.
- If there are many threads waiting, one is selected *at random*.
- A call to obj.notifyAll() wakes up all threads that are waiting on obj. They all move from "waiting" to "wanting" in the state diagram.

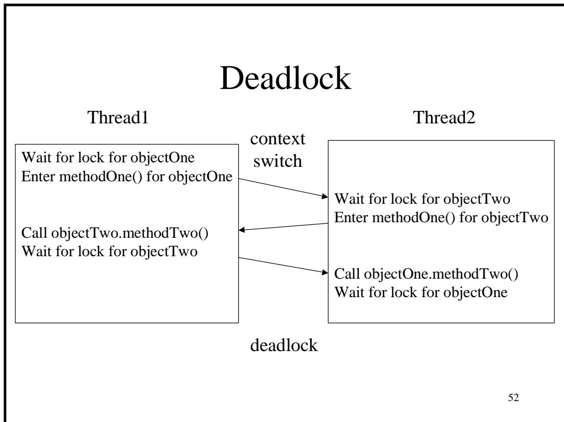
50

Synchronization States



51

Deadlock



52

Resource Ordering

- One way to avoid deadlock is to always obtain locks in *the same order* for all threads.
- In the previous example all threads should first obtain the lock for objectOne, and then for objectTwo. Thread2 gets them in the wrong order.

53

- Suppose the call looked like
  `objectA.methodOne(objectB);`
  where for thread1, objectA is objectOne.
- We could obtain the locks with

```
synchronized(objectA) {
  synchronized(objectB) {
    objectA.methodOne(objectB);
  }
}
```

- This still doesn't work for both threads because thread2 has objectA referring to objectTwo.

54

Adding a resource number to each object, we can avoid deadlock and still have common code.

```
Resource resOne, resTwo;
if (objectA.getResNumber() < objectB.getResNumber) {
  resONe = objectA;
  resTwo = objectB;
}
else {
  resOne = objectB;
  resTwo = objectA;
}
synchronized (resOne) {
  synchronized (resTwo) {
    objectA.methodOne(objectB);
  }
}
```

55

Encapsulating Synchronization

- Recall that encapsulation means preventing one module from knowing details of another.
- Using synchronized methods, exposes the synchronization of the class.
- Any method can obtain the lock for any object it can reference.
- We can use synchronized statements to hide the object we use for locking.

56

```
class SomeClass {
  synchronized void critical() {
      //method body goes here
  }
}



class SomeClass {
  void critical() {
    synchronized (lock) {
      //method body goes here
    }
  }
  private Object lock = new Object();
}
```

57