2/17/2023

Client/Server Computing

Threads

2/17/2023 1

A single threaded program

class ABC

{

….

public void main(..)

{

…

..

}

}

2

begin

body

end

1

2/17/2023

A Multithreaded Program

Main Thread

start startstart

Thread A Thread B Thread C

Threads may switch or exchange data/results 3

Single and Multithreaded Processes threads are light-weight processes within a process

Single-threaded Process

Multiplethreaded Process

Threads of

Execution

Single instruction stream Multiple instruction stream Common

Address Space

4

2

2/17/2023

Multithreaded Server: For Serving Multiple Clients Concurrently



**Client 1 Process Server Process**

**Server**

**Threads**

■ Internet

**Client 2 Process**

5

Web/Internet Applications:

Serving Many Users Simultaneously

**PC client**

**Internet**

**Server**

**Local Area Network**

**PDA**

6

3

2/17/2023

What are Threads?

⌘A piece of code that run in concurrent with other threads.

⌘Each thread is a statically ordered sequence of instructions.

⌘Threads are being extensively used express concurrency on both single and multiprocessors machines.

⌘Programming a task having multiple threads of control – Multithreading or Multithreaded Programming.

7

Threads in Programming Languages 

⌘ Several programming languages have long provided constructs/abstractions for writing concurrent programs ⮹ Modula, Ada, etc.

⌘ Java does it like it does everything else, by providing a Thread class

⮹ You create a thread object

⮹ Then you can start the thread

⌘ Java provides several higher level abstractions

4

2/17/2023

Extending the Thread class

⌘ To create a thread, you can extend the thread class and override its “run()” method

**class MyThread extends Thread {**

**public void run() {**

**. . .**

**}**

**. . .**

**}**

**myThread t = new MyThread();**

Example



**public class MyThread extends Thread {**

**public void run() {**

**for (int i=0; i<10; i++) {**

**System.out.println(“Hello world #“+i);**

**}**

**}**

**. . .**

**}**

**myThread t = new MyThread();**

5

2/17/2023

Spawning a thread



⌘ To launch, or spawn, a thread, you just call the thread’s start() method

⌘ **WARNING**: Don’t call the run() method directly to launch a thread

⮹ If you call the run() method directly, then you just call some method of some object, and the method executes

⌧Fine, but probably not what you want

⮹ The start() method, which you should not override, does all the thread launching

⌧It launches a thread that starts its execution by calling the run() method

The Thread Object

⌘ A thread is not an object

⌘ A Thread **is** an object

void start()

⮹Creates a new thread and makes it runnable

void run()

⮹The new thread begins its life inside this method

6

2/17/2023

What happens

⌘ The previous program runs as a Java process ⮹that is, a thread running inside the JVM

⌘ When the start() method is called, the main thread creates a new thread

⌘ We now have two threads

⮹The “main”, “original” thread

⮹The newly created thread

⌘ Both threads are running

⮹The main thread doesn’t do anything

⮹The new thread prints messages to screen and exits ⌘ When both threads are finished, then the process terminates

What happens

**Host**

**JVM process**

**Your**

**program’s main thread**

**Newly created thread**

7

2/17/2023

Thread Creation Diagram



Object A

**Thread t = new BThread(); t.start();**

**doMoreStuff();**

Object BThread (extends Thread)

**BThread() {**

**}**

**void start() {**

**// create thread**

**}**

**void run() {**

**doSomething();**

**}**

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

8

2/17/2023

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

9

2/17/2023

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

10

2/17/2023

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

11

2/17/2023

Thread Creation Diagram



Object A

Thread t = new BThread(); t.start();

doMoreStuff();

Example

Object BThread (extends Thread)

BThread() {

}

void start() {

// create thread

}

void run() {

doSomething();

}

**public class MyThread extends Thread {**

**public void run() {**

**for (int i=0; i<5; i++) {**

**System.out.println(“Hello world #“+i);**

**}**

**}**

**}**

**public class MyProgram {**

**public MyProgram() {**

**MyThread t = new MyThread();**

**t.start();**

**}**

**public static void main(String args[]) {**

**MyProgram p = new MyProgram();**

**}**

**}**

12

2/17/2023

Example



⌘ The previous example wasn’t very interesting because the main thread did nothing

⮹ Admittedly, this example is not interesting because the program doesn’t do anything useful, but we’ll get there eventually

⌘ In fact, we could have achieved the same result with no thread at all

⌘ So, let’s have the main thread to something

Example

**public class myThread extends Thread { public void run() {**

**for (int i=0; i<5; i++)**

**System.out.println(“Hello world #“+i);**

**}**

**}**

**public class MyProgram {**

**public MyProgram() {**

**MyThread t = new MyThread();**

**t.start();**

**for (int i=0; i<5; i++)**

**System.out.println(“Beep ”+i);**

**}**

**public static void main(String args[]) {**

**MyProgram p = new MyProgram();**

**}**

**}**

13

What happens?



⌘ Now we have the main thread printing to the screen **and** the new thread printing to the screen

⌘ Question: what will the output be?

⌘ Answer: Impossible to tell for sure

⮹ If you know the implementation of the JVM on your particular machine, then you can probably tell

⮹ But if you write this code to be run anywhere, then you can’t expect to know what happens

⌘ Let’s look at what happens on my laptop

Example Execution



2/17/2023 14

Is it really concurrent?

⌘ One may wonder whether the execution is really concurrent ⮹ At least falsely concurrent

⌘ This can be verified by having threads run for longer ⌘ If the new thread prints “.” and the main thread prints “#”

Example Execution #1



2/17/2023 15

2/17/2023

Example Execution #2



Non-deterministic Execution!



⌘ The previous example shows one difficulty with thread programming, and especially debugging: it may be difficult to tell what the execution will look like

⌘ Somebody decides when a thread runs

⮹ You run for a while

⮹ Now you run for a while

⮹ . . .

⌘ This decision process is called thread scheduling

16

With False Concurrency

■ On my old Single Core laptop from 2 years ago



On a Different O/S

⌘ On a dual proc, hyperthreaded Linux box! ⮹Less interleaving than on my Mac OSX dual-core laptop ⮹Windows boxes typically also show less concurrency

2/17/2023 17

2/17/2023

Thread Scheduling

⌘ Threads are scheduled like processes

⌘ Thread states

⮹Running

⮹Waiting, Sleeping, Suspended, Blocked ⮹Ready

⮹Dead

⌘ When you invoke start() the Thread is marked ready and placed in the thread queue

Thread States

The start() method places a

thread in the ready state

Ready

Waiting

The scheduler selects a thread

and places it in the running state

Running

A thread that is waiting for I/O, was suspended, is sleeping, blocked, or otherwise is unable to do any more work is placed in the waiting state

18

2/17/2023

isAlive()

⌘ The method isAlive() determines if a thread is considered to be alive

⮹A thread is alive if it has been

started and has not yet died.

isAlive()

public class WorkerThread extends Thread { private int result = 0;

public void run() {

// Perform a complicated time consuming calculation // and store the answer in the variable result }

public static void main(String args[]) {

WorkerThread t = new WorkerThread();

t.start();

while ( t.isAlive() ); // What is wrong with this?

System.out.println( result );

}

}

19

2/17/2023

sleep()

⌘ Puts the currently executing thread to sleep for the specified number of milliseconds

⮹sleep(int milliseconds)

⮹sleep(int millisecs, int nanosecs)

⌘ Sleep can throw an InterruptedException

⌘ The sleep method is static and can be accessed through the Thread class name

sleep()

public class WorkerThread extends Thread {

private int result = 0;

public void run() {

// Perform a complicated time consuming calculation // and store the answer in the variable result

}

public static void main(String args[]) {

WorkerThread t = new WorkerThread();

t.start();

while ( t.isAlive() )

try {

sleep( 100 );

} catch ( InterruptedException ex ) {}

System.out.println( result );

}

}

20

2/17/2023

Timer

import java.util.Date;

class Timer extends Thread {

public void run() {

while ( true ) {

System.out.println( new Date() );

try {

Thread.sleep(1000);

}

catch ( InterruptedException e ) {}

}

}

public static void main( String args[] ) {

Timer t = new Timer();

t.start();

System.out.println( "Main done" );

}

}

yield()



⌘ A call to the yield() method causes the currently executing thread to go to the ready state (this is done by the thread itself)

⌘ With the yield() method, a thread will pause and give other RUNNABLE threads the opportunity to execute for a while

42

21

2/17/2023

yield()

public class WorkerThread extends Thread {

private int result = 0;

public void run() {

// Perform a complicated time consuming calculation // and store the answer in the variable result }

public static void main(String args[]) {

WorkerThread t = new WorkerThread();

t.start();

while ( t.isAlive() )

yield();

System.out.println( result );

}

43

}

Joining Threads

⌘ Calling isAlive() to determine when a thread has terminated is probably not the best way to

accomplish this

⌘ What would be better is to have a method that once invoked would wait until a specified thread has terminated

⌘ join() does exactly that

⮹join()

⮹join(long timeout)

⮹join(long timeout, int nanos)

⌘ Like sleep(), join() is static and can throw an InterruptedException

44

22

2/17/2023

join()

public class WorkerThread extends Thread { private int result = 0;

public void run() {

// Perform a complicated time consuming calculation // and store the answer in the variable result }

public static void main(String args[]) {

WorkerThread t = new WorkerThread();

t.start();

try {

t.join();

} catch ( InterruptedException ex ) {}

System.out.println( result );

45

}

}

Race Conditions

⌘Two threads are simultaneously modifying a single object

⌘Both threads “race” to store their value

⌘In the end, the last one there “wins the race” ⌘(Actually, both lose)

23

2/17/2023

Race Condition Example

**class Account {**

**int balance;**

**public void deposit(int val)**

**{**

**int newBal;**

**newBal = balance + val;**

**balance = newBal;**

**}**

**}**

Thread Synchronization

⌘ Language keyword: synchronized

⌘ Takes out a monitor lock on an object

⮹Exclusive lock for that thread

⌘ If lock is currently unavailable, thread will block

24

2/17/2023

Race Condition Example

**class Account {**

**int balance;**

**public synchronized void deposit(int val) {**

**int newBal;**

**newBal = balance + val;**

**balance = newBal;**

**}**

**}**

Monitors

⌘Each object has a “monitor” that is a token used to determine which application thread has control of a particular object instance

⌘In execution of a synchronized method (or block), access to the object monitor must be gained before the execution

⌘Access to the object monitor is queued

25

2/17/2023

Monitor (cont.)

⌘Entering a monitor is also referred to as locking the monitor, or acquiring ownership of the monitor

⌘If a thread A tries to acquire ownership of a monitor and a different thread has already entered the monitor, the current thread (A) must wait until the other thread leaves the monitor

Critical Section

⌘The synchronized methods define critical sections

⌘Execution of critical sections is mutually exclusive.

26

2/17/2023

Example

public class BankAccount { private float balance;

public synchronized void deposit(float amount) { balance += amount;

}

public synchronized void withdraw(float amount) { balance -= amount;

}

}

Critical Sections

t3 t2 t1

deposit()



Bank Account

27

2/17/2023

One other problem

⌘ Sometimes, a synchronized method or block needs to wait for something

⌘ So, no other thread can run it since the current thread is in the synchronized method and

waiting.

⌘ Luckily, there is a feature of synchronized

methods which takes care of this situation.

⌘ A thread can call the object’s wait() method in order to wait inside of a synchronized method. This causes the current (calling) thread to block and the object’s lock to be released.

55

Wait and Notify

⌘Allows two threads to cooperate

⌘Based on a single shared lock object

28

2/17/2023

wait list

⌘Once a thread has called wait(), it enters a wait list. This is a list of threads waiting on a certain object.

⌘To remove the thread from the wait list, some other thread must call the object’s notify() or notifyAll() method.

⌘Notify() removes one arbitrary thread from the waiting list; notifyAll() removes

them all.

wait/notify

57



⌘Once a thread has been removed from the wait list, the scheduler will eventually run it again. At that point, it will attempt to lock the object (remember it gave up the lock when it called wait()).

⌘Once the thread gets the lock, it will

reenter the object and continue where it left off after the call to wait().

⌘This is known as reentry.

58

29

2/17/2023

wait/notify



⌘ It is VERY important that some other thread calls notify() or notifyAll().

⌘ When a thread calls wait(), there is no way of unblocking it unless another thread calls notify().

⌘ The waiting threads are not automatically

reactivated when no other thread has the object locked.

⌘ If all thread but one are blocked and that thread calls wait(), the program will simply wait forever as there is no thread left to call notify().

59

wait/notify



⌘ It is somewhat confusing as to when a program should call notify(), as there is no control as to which thread will become unblocked.

⌘ Generally, call notifyAll() whenever the state of the object changes in such a way that might be advantageous to other threads.

⌘ For example, if the account balance of an

account changes, multiple threads may be

blocked based on this event occurring, so it is a good idea to wake them all up.

60

30

2/17/2023

wait/notify

⌘ Note that calling notifyAll() does not immediately activate all of the waiting threads.

⌘ It simply unblocks the waiting threads and

removes them from the wait list.

⌘ They still must compete for entry into the object after the current thread has exited its

synchronized method.

61

The wait() Method

⌘**wait()** is also similar to **yield()**

⮹Both take the current thread off the

execution stack and force it to be rescheduled ⌘ However, **wait()** is not automatically put back into the scheduler queue

⮹**notify()** must be called in order to get a thread back into the scheduler’s queue

31

2/17/2023

Deadlocks



⌘ A deadlock is a situation where two threads are stuck, each waiting for the other to do something first. A deadlock could also involve a ring of

threads, waiting for each other, or simply one thread, waiting for itself.

⌘ Java provides no mechanisms for dealing with Deadlock. The programmers have to write code that will not enter this state, which can be hard to do.

63

The Runnable Interface



⌘ What if you want to create a thread that extends some other class?

⮹e.g., a multi-threaded applet is at the same time a Thread and an Applet

⌘ Java does not allow for double inheritance

⌘ Which is why it has the concept of interfaces

⌘ So another way to create a thread is to have runnable objects

⌘ It’s actually the most common approach

⮹Allows to add inheritance in a slightly easier way after the fact ⌘ Let’s see this on an example

32

2/17/2023

Runnable Example

**public class RunnableExample {**

**class MyTask implements Runnable {**

**public void run() {**

**for (int i=0; i<50; i++)**

**System.out.print("#");**

**}**

**}**

**public RunnableExample() {**

**Thread t = new Thread(new MyTask());**

**t.start();**

**for (int i=0; i<50; i++)**

**System.out.println(".");**

**}**

**public static void main(String args[]) {**

**RunnableExample p = new RunnableExample();**

**}**

**}**

2nd method: Threads by implementing Runnable interface

class MyThread implements Runnable

{

.....

public void run()

{

// thread body of execution

}

}

⌘ Creating Object:

MyThread myObject = new MyThread();

⌘ Creating Thread Object:

Thread thr1 = new Thread( myObject );

⌘ Start Execution:

thr1.start();

66

33

2/17/2023

**An example**

****class MyThread implements Runnable {

public void run() {

System.out.println(" this thread is running ... ");

}

} // end class MyThread

class ThreadEx2 {

public static void main(String [] args ) {

**Thread t = new Thread(new MyThread());**

// due to implementing the Runnable interface

// I can call start(), and this will call run().

**t.start();**

} // end main()

} // end class ThreadEx2

67

Example

**public class MyThread extends Thread {**

**public void run() {**

**for (int i=0; i<5; i++) {**

**System.out.println(“Hello world #“+i);**

**}**

**}**

**}**

**public class MyProgram {**

**public MyProgram() {**

**MyThread t = new MyThread();**

**t.start();**

**}**

**public static void main(String args[]) {**

**MyProgram p = new MyProgram();**

**}**

**}**

68

34

2/17/2023

Thread Priority

⌘Every thread has a priority

⌘When a thread is created, it inherits the priority of the thread that created it

⌘The priority values range from 1 to 10, in increasing priority

Thread Priority (cont.)

⌘The priority can be adjusted subsequently using the **setPriority()** method

⌘The priority of a thread may be obtained using **getPriority()**

⌘

⌘Priority constants are defined:

⮹MIN\_PRIORITY=1

⮹MAX\_PRIORITY=10

⮹NORM\_PRIORITY=5

35

2/17/2023

Some Notes

⌘Thread implementation in Java is actually based on operating system support

⌘Some Windows operating systems support only 7 priority levels, so different levels in Java may actually be mapped to the same operating system level

Blocking Threads

⌘When reading from a stream, if input is not available, the thread will block

⌘Thread is suspended (“blocked”) until I/O is available

⌘Allows other threads to automatically activate ⌘When I/O available, thread wakes back up again ⮹Becomes “runnable”

⮹Not to be confused with the Runnable interface

36

2/17/2023

Thread Scheduling



⌘ In general, the runnable thread with the highest priority is active (running)

⌘ Java is priority-preemptive

⮹If a high-priority thread wakes up, and a low-priority thread is running

⮹Then the high-priority thread gets to run immediately

⌘ Allows on-demand processing

⮹Efficient use of CPU

Thread Starvation

⌘If a high priority thread never blocks

⌘Then all other threads will starve ⌘Must be clever about thread priority

37

2/17/2023

Conclusion



⌘ Two ways to create threads

⮹extends Thread

⮹implements Runnable

⌘ Thread Scheduling is complex, not fully deterministic, and should not be counted on to guarantee program correctness

76

38