CS F213
Object Oriented Programming

BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.

Multithreaded Programming

Ch.11 of R1.

The Complete Reference- Java, 7th Edition, Herbert Schildt, Tata McGraw Hill Publishing.

And Ch.11 in T2 Jva Complete refererence 11 th Edition.

And also refer to Class notes.

Content

1. Introduction to multithreading
2. Java thread model
3. Thread class and Runnable interface
4. Creating a thread and multiple threads
5. Synchronization
6. Interthread communication
7. Examples

Introduction

Operating System

A set of programs that act as layer on top of underlying HW to shield its complexities.

Provides interface to different systems , programs.

Manages resources:

- ❑ CPU time
- ❑ Memory
- ❑ IO
- ❑ Files etc.

Multiprocessing

Process is an abstraction of a running program.

Several programs can be kept in the memory.

If one job goes on IO then we can start another job.

Hence, CPU utilization is optimized.

This increases throughput.

Switching happens so fast that user will get an illusion that they are running parallel.

Process Control Block (PCB) or Process Table.

A process includes- executing program, registers, PC, variables etc.

The rapid switching between processes is known as multiprogramming or multiprocessing.

True parallelism comes through multiple HW units.

PCB stores info like- State, PC, Stack pointer, mem allocation, files, scheduling info.

Each process has an address space and single thread of control.

Scheduling Algorithms

Could be Preemptive and Non-preemptive

- Non preemptive

(terminates or blocks/waits on its own)

- Preemptive scenario (OS interrupts it)

- First –Come-First-Serve (FCFSC) NP
- Shortest Job First (SJF) NP
- Round Robin(RR) P
- Priority based NP/P

Threads

Modern SW applications need to perform multiple tasks at same time.

Ex: GUI display, Open files, take inputs, do some background processing etc.

In traditional way multiple processes of the same applications are created.

But that causes lot of overhead.

To avoid this overhead, a single process can be segmented into multiple sequences of execution called threads.

Each such sequence shares some resources with its parent process.

Each thread of a process is recognized by thread ID, and comprises program counter, stack, register set, and remaining resources are shared with the process.

While switching only the concerned constructs need to be shared.

Process: is a program under execution

Thread: is part of a program which has independent path of execution

Task : is an executable unit which has independent path

Multitasking

Process-based

Thread-based

Per process items	Per thread items
Address space	PC
Global variables	Registers
Open files	Stack
Child processes	State
signals	
Accounting info	

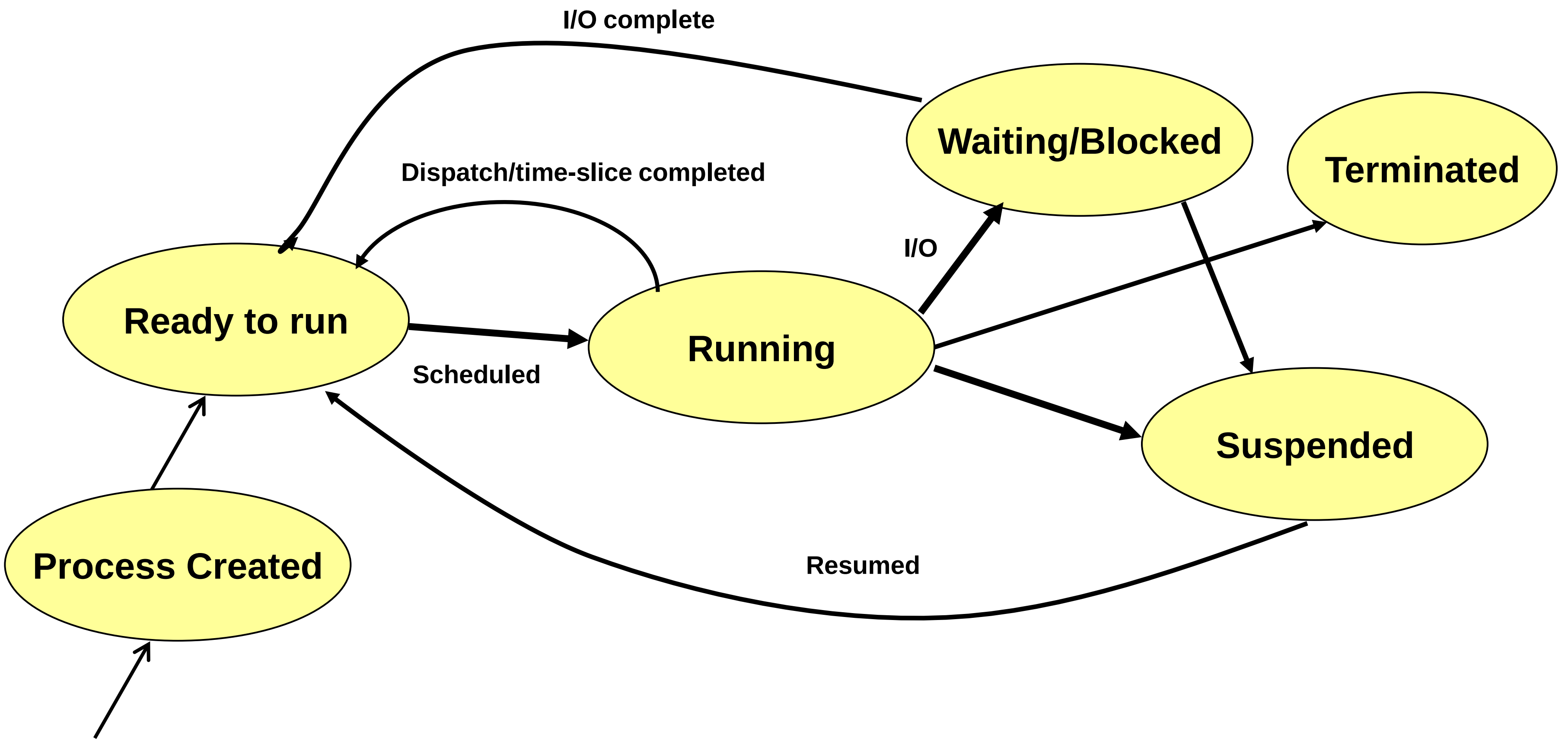
Multiple processes share memory, printers, disk etc	Multiple threads share address space, open files etc.

Process Vs. Thread

1. Address space
2. It is desirable to have multiple threads of control within a single program
3. Context switching
4. Inter Process Communication

Threads are lightweight tasks.

- A multithreaded program contains two or more parts that can run concurrently.
- Each part is called a thread.
- In thread-based multitasking, thread is the smallest unit of dispatchable code.
- A single program can perform more than one task simultaneously.
- We can write programs which are efficient.



Process/Thread states

Thread priority

To determine how a thread should be treated with respect to other threads.

It is an integer.

It doesn’t specify execution speed of a thread.

Synchronization

Java implements synchronization with *monitors*.

A *monitor* is a box which allows only one thread at a time.

Messaging

How different threads talk to each other.

Ex: notifying a waiting thread.

Thread class and ***Runnable*** interface

Thread encapsulates a thread of execution

To create a new thread

Extend the *Thread* class

Implement the interface *Runnable*

The main thread:

When a Java program starts up one thread begins running immediately and is called the ***main thread***.

Other threads are spawned from this main thread.

This is the last thread to finish.

To get the reference of main thread:

Static Thread *currentThread()* is a method in *Thread* class.

Example

```
class CurrentThreadDemo{
{     public static void main(String args[])
      {         Thread t=Thread.currentThread();
                System.out.println(“ current thread  :”+t);
                t.setName(“My Thread”);
                System.out.println(“ After name change   :”+t);
                try{
                                for(int i=5;  i>0;  i--)
                                {   System.out.println(i);
                                    Thread.sleep(1000);
                                }
                } catch(InterruptedException ie)
                {System.out.println(“ Main thread interrupted  :”);}
      }
}
```

```
current thread : Thread[main, 5, main]

After name change : Thread[My Thread, 5,
main]

5

4

3

2

1
```

final void setName(String name)

final String getName()

static void sleep(long millsec) throws InterruptedException

final boolean isAlive()

final int getPriority()

public void run()

void start()

final void join() throws InterruptedException

Java Interface

A Java interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all of the methods defined in the interface, thereby agreeing to certain behavior.

Interfaces are provided in Java for the following reason.

Java supports only single inheritance. To allow multiple inheritance, an interface can be implemented to achieve the same purpose.

An interface is a named collection of method definitions (without implementations).

An interface can also include variables which are static and final and must be initialized. Implementing class can not change them.

Meaning they are constants.

All methods and variables are implicitly public.

Defining an Interface

Defining an interface is similar to creating a new class. An interface definition has two components: the interface declaration and the interface body.

```
public interface MyInterface {
    interfaceBody
}
```

if an implementing class provides partial implementation it should be declared abstract.

One interface can inherit another interface by using extends.

Creating a new thread

By implement the interface ***Runnable***

The ***Runnable*** interface has only one method that is *run()* which defines the code that constitutes the new thread

Thread(Runnabl runobj, String name)

One class can implement any number of interfaces thus simulating multiple inheritance.

We can create a thread By extending the class Thread.

Still we need to override the method run() which defines the code that constitutes the new thread

class *Thread* implements *Runnable*

Thread(Runnable runobj, String name)

Thread(String name)

```
class NewThread implements Runnable
{      Thread t;
       NewThread(){
       t= new Thread(this, “Demo Thread”);
       System.out.println(“ Child Thread    :”+t);
       t.strat(); }
       public void run(){
                 try{
                                 for(int i=5;i>0;i--)
                                 {   System.out.println(“ Child Thread  : ”+i);
                                     Thread.sleep(500);
                                 }
                 } catch(InterruptedException ie)
                 {System.out.println(“ Child  interrupted  :”);}
        System.out.println(“ Exiting child  :”);}
       }
}
```

```
class ThreadDemo
{    public static void main(String args[])
     {   new NewThread();
              try{
             for(int i=5;  i>0;  i--)
             {   System.out.println(“ Main Thread  : ”+i);
                  Thread.sleep(1000);
             }
             } catch(InterruptedException ie)
             {System.out.println(“ main thread  interrupted  :”);}
      System.out.println(“ Main Thread Exiting  :”);}
     }
}
```

```
Child Thread : Thread[Demo thread, 5, main]
Main Thread : 5
Child Thread : 5
Child Thread : 4
Main Thread : 4
Child Thread : 3
Child Thread : 2
Main Thread : 3
Main Thread : 2
Child Thread : 1
Exiting child :
Main Thread : 1
Main Thread Exiting
```

Synchronization

When multiple threads access shared resources, it is required to make sure that only one thread will use it at a given instance of time.

The process by which this is achieved is called **synchronization**.

Java supports this at language level.

Key to this is a notion of ***monitor***.

A monitor represents a lock.

If a thread acquires the lock it is said to have entered the monitor. Then all other threads trying to acquire the lock will have to wait till the first one comes out.

Every java object will have its implicit monitor.

To enter the monitor of an objects just call a method which is modified with the keyword ***synchronized***.

Using synchronized blocks

We use this for synchronized access of methods of a class designed by somebody else.

```
public void run(){
        synchronized(obj)
        {
                obj.meth();
        }
   }
```

Interthread communication

final void wait() throws InterruptedException

tells the calling thread to give up monitor and go to sleep until some other thread enters the same and calls notify()

final void notify()

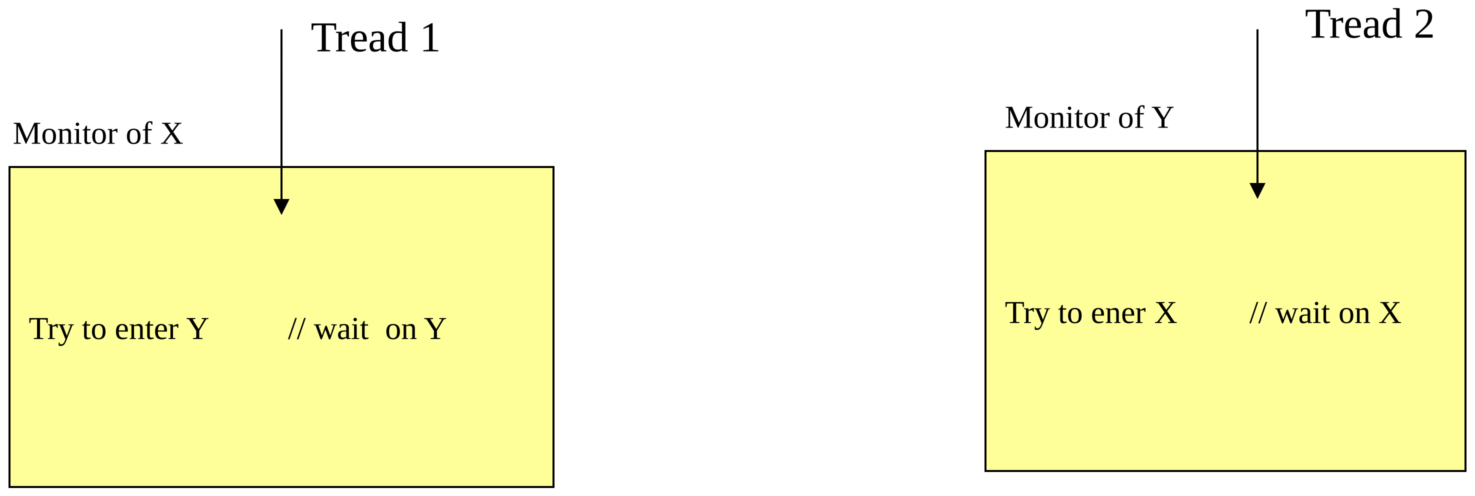
wakes up the first thread that called wait() on the same object

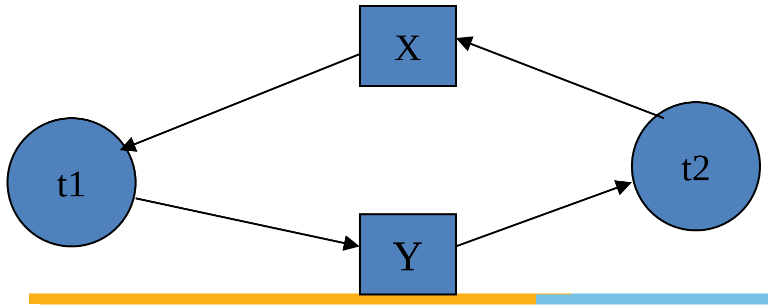
final void notifyAll()

wakes up all threads that have called wait() on the same object

Deadlocks

Occurs when two threads have a circular dependency on a pair of synchronized objects.





final void suspend()

final void resume()

final void stop()

When to use multithreding

When your program has more subsystems which need to execute concurrently.

Don’t use too many threads, it incurs lot of overhead in context switching.

Summary

1. Introduction to multithreading
2. Java thread model
3. Thread class and Runnable interface
4. Creating a thread and multiple threads
5. Synchronization
6. Interthread communication
7. Examples