CSE2005 (Object Oriented Programming Systems)

Module 5
Concurrent Programming

Dr. Arundhati Das

Need for Multithreading

- Have you faced the following situations:
    - Your browser cannot skip to the next web page because it is downloading a file?
    - You cannot enter text into your current document until your word process completes the task of saving the document to disk

- These situations occur when the software you are using like the browser, or word processor consists of a single thread.

What is Multitasking?

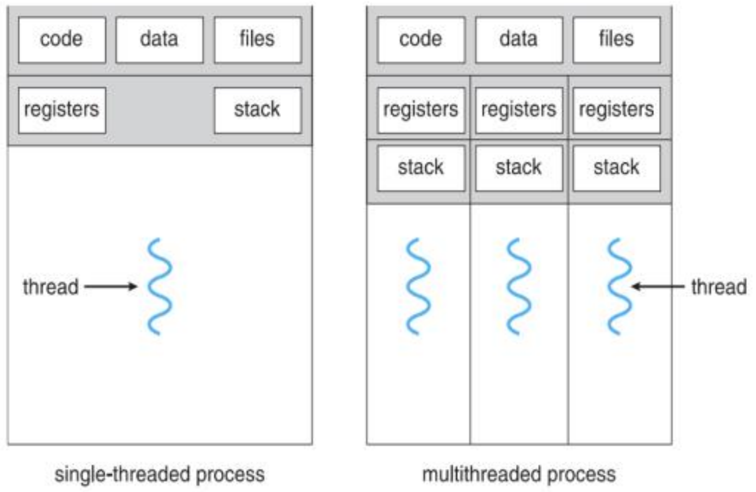
- Multitasking is synonymous with process-based multitasking, whereas multithreading is synonymous with thread-based multitasking
- All modern operating systems support multitasking
- A process is an executing instance of a program
- Process-based multitasking is the feature by which the operating system to run two or more programs concurrently

What is Multithreading?

- In multithreading, the thread is the smallest unit of code that can be dispatched by the thread scheduler
- A single program can perform two tasks using two threads
- Only one thread will be executing at any given point of time given a single-processor architecture

Single and Multithreaded Processes

As each thread has its own independent resource for process execution, multiple processes can be executed parallely by increasing number of threads.



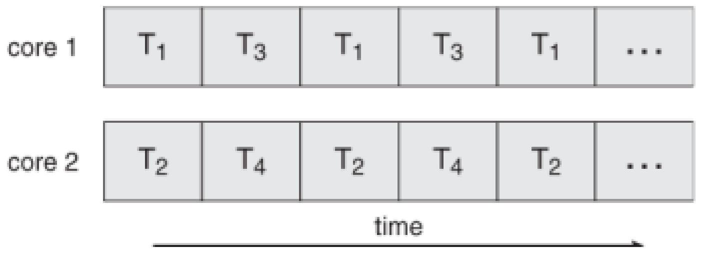


Fig. : Parallel execution with multiple threads executing independently in multiple cores

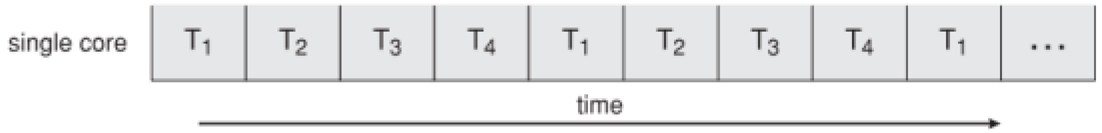


Fig.: Multi-tasking or concurrent execution

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multitasking Vs. Multithreading

- when compared to multitasking processes
  - Each process requires its own separate address space
  - Context switching from one process to another is a CPU-intensive task needing more time
  - Inter-process communication between processes is again expensive as the communication mechanism has to span separate address spaces

- These are the reasons why processes are referred to as heavyweight tasks

Multitasking Vs. Multithreading

- Multitasking threads cost less in terms of processor overhead because of the following reasons
  - Multiple threads in a program share the same address space, and cooperatively share the same heavyweight process
  - Context switching from one thread to another is less CPU-intensive
  - Inter-thread communication, on the other hand, is less expensive as threads in a program communicate within the same address space

- Threads are therefore called lightweight processes

What is Multithreading

- A multithreaded application performs two or more activities concurrently

- Accomplished by having each activity performed by a separate thread

- Threads are the lightest tasks within a program, and they share memory space and resources with each other

Java’s Multithreading Model

- Unlike many other computer languages, Java provides built-in support for multithreaded programming.
  A multithreaded program contains two or more parts that can run concurrently.
- *In Java*
  - *All the libraries and classes are designed with multithreading in mind*
  - *This enables the entire system to be asynchronous*
- In Java:
  - the **java.lang.Thread** class is used to create thread-based code
  - imported into all Java applications by default

Thread Priorities

- A thread priority:
  - decides how that thread should be treated with respect to other threads
  - is set when created
  - is used to decide when to switch from one running thread to another

- This is called a **context switch**.

- Higher priority threads are guaranteed to be scheduled before lower priority threads

Deciding On a Context Switch

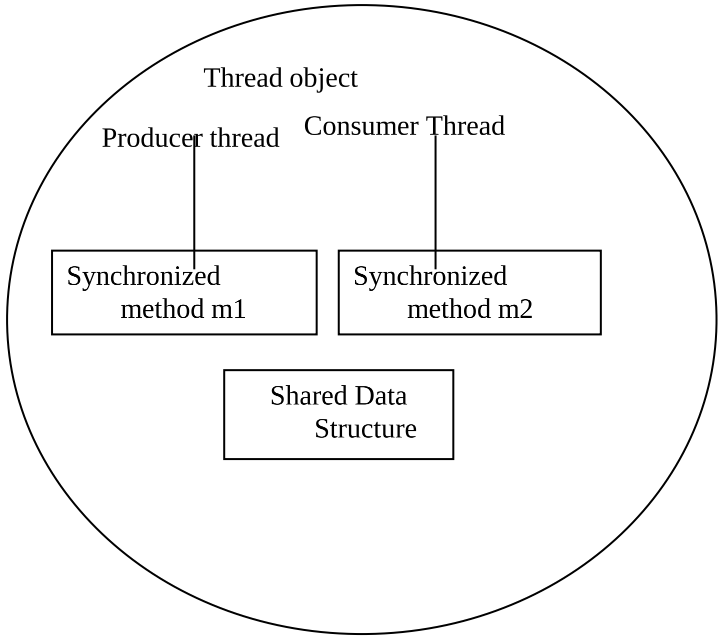
- A thread can voluntarily relinquish control:
  - by explicitly yielding, sleeping, or blocking on pending Input/Output
- All threads are examined and the highest-priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread:
  - a lower-priority thread that does not yield the processor is superseded, or preempted by a higher-priority thread
  - Whenever a higher priority thread wants to run, it does
  - This is called preemptive multitasking
- **When two threads with the same priority are competing for CPU time, threads are time-sliced in round-robin fashion**

Synchronization

- It is normal for threads to be sharing objects and data
- Different threads shouldn't try to access and change the same data at the same time
- Threads must therefore be synchronized
- For example, imagine a Java application where:
  - one thread (the producer) writes data to a data structure,
  - while a second thread (the consumer) reads data from the data structure

Synchronization

this example use concurrent threads that share a common resource: a data structure.



Synchronization

- The current thread operating on the shared data structure: must be granted mutually exclusive access to the data
- the current thread gets an exclusive lock on the shared data structure, or a **mutex**

A **mutex** is a concurrency control mechanism used to ensure the integrity of a shared data structure

Synchronization

Mutex is not assured if the methods in the thread object accessed by competing threads are ordinary methods

It might lead to a race condition when the competing threads will race each other to complete their operation

A **race condition** can be prevented by defining the methods accessed by the competing threads as **synchronized**

Synchronization

Synchronized methods are an elegant variation on an time-tested model of interprocess-syncrhonization: the **monitor**.

The monitor is a thread control mechanism

When a thread enters a monitor **(synchronized method)**, all other threads must wait until that thread exits the monitor

The monitor acts as a concurrency control mechanism

Thread Messaging

In Java, you need not depend on the OS to establish communication between threads

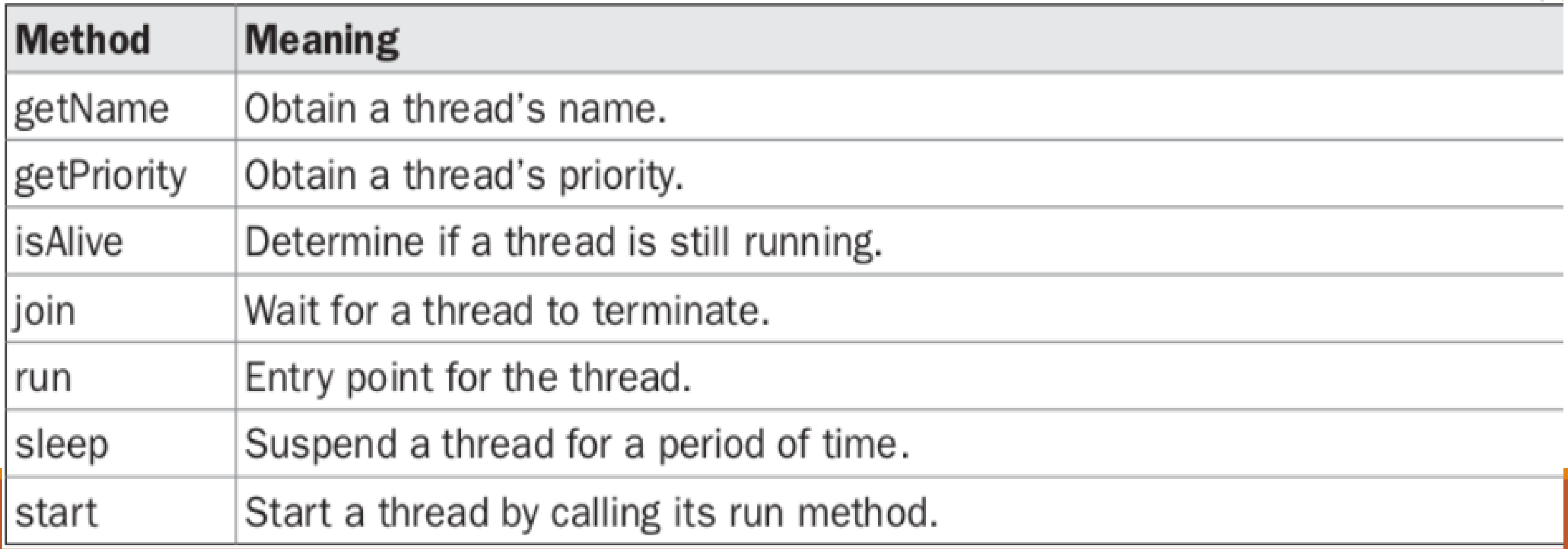
All objects have predefined methods, which can be called to provide inter-thread communication.

The Thread Class and the Runnable Interface

Java’s multithreading feature is built into the **Thread** class, its methods, and its companion interface, Runnable.

To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads.



Method	Meaning
getName	Obtain a thread’s name.
getPriority	Obtain a thread’s priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The main Thread

When a Java program starts executing

- the main thread begins running
- the main thread is immediately created when **main()** commences execution

The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.
- To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread.
- Its general form is shown here:

```
static Thread currentThread( )
```

Obtaining Thread-Specific Information

Example 1:

```
public class ThreadDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Thread t=Thread.currentThread();
		System.out.println("the current thread:"+t);
		t.setName("oop thread");
		System.out.println("the new name:"+t);
		try
		{
			Thread.sleep(2000);
		}
		catch(InterruptedException e)
		{
			System.out.println("you are interrupting sleep");
		}	}
}
```

Example 2:

```
class CurrentThreadDemo {public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
        try {
                for(int n = 5; n > 0; n--)   {
                System.out.println(n);
                Thread.sleep(1000);
                                                    }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted");
                }
                        }
                            }
```

Example 3:

```
class MyThread1 extends Thread
{
        public void run()
        {
            int i=100;
            while(i>0)
                {
                    System.out.println("Thread 1 is running");
                    i--;
                }
        }
}
```

```
class MyThread2 extends Thread
{
        public void run()
        {
            int i=100;
            while(i>0)
                {
                        System.out.println("Thread 2 is running");
                        i--;
                }
        }
}
```

```
Public MultiThreads
{
        public static void main(String args[])
                {
                        MyThread1 t1=new MyThread1();
                         MyThread2 t2=new MyThread2();
                        t1.start();
                        t2.start(); //start() will call run() internally
                }
}
```

Creating Threads

A thread can be created by instantiating an object of type **Thread**.

Java defines two ways in which this can be accomplished:

- implementing the **Runnable** interface
- extending the **Thread** class

Creating Threads – Implementing Runnable

- Create a class that implements the **Runnable** interface
- **Runnable** abstracts a unit of executable code for the thread
- A thread can be constructed on any object that implements the **Runnable** interface
- To implement **Runnable**, a class need to implement only a single method called **run( )**

Creating Threads – Implementing Runnable

After you define a class that implements **Runnable**, you will instantiate an object of type thread from within an object of that class. This thread will end when **run( )** returns, or terminates.

This is mandatory because a thread object confers multithreaded functionality to the object from which it is created.

Therefore, at the moment of thread creation, the thread object must know the reference of the object to which it has to confer multithreaded functionality. This point is borne out by one of the constructors of the **Thread** class.

Creating Threads – Thread class

The **Thread** class defines several constructors one of which is:

- **Thread(Runnable threadOb, String threadName)**

In this constructor, **threadOb:**

- is an instance of a class implementing the **Runnable** interface
- ensures that the thread is associated with the **run( )** method of the object implementing **Runnable**

This defines where execution of the thread will begin

The name of the new thread is specified by **threadName**.

Creating Threads – Implementing Runnable

```
class DemoThread implements Runnable
{
Thread t;
DemoThread()
{
t = new Thread(this, "Demo Thread");
System.out.println("Child Thread: " + t);
t.start();
}

public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
}
```

[refer the eclipse code that I showed in class]

Creating Threads – Implementing Runnable

```
catch (InterruptedException e)
{
System.out.println("Child Thread Interrupted");
}
System.out.println("Exiting Child Thread");
}
}
class ThreadImpl
{
public static void main(String args[])
{
new DemoThread();

try
{
for(int i=5; i>0; i--)
{
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
}
```

Child thread

Main thread

Creating Threads – Implementing Runnable

```
catch (InterruptedException e)
{
System.out.println("Main Thread Interrupted");
}
System.out.println("Main Thread Exiting");
}
}
```

Extending Thread

An alternative way to create threads:

- Is to create a new class that extends **Thread**
- Create an instance of that class
- The extending class must override the **run( )** method
- run( ) is the entry point for the new thread
- The **start( )** method must be invoked on the thread
- **start( )** initiates a call to the thread’s **run( )** method

The earlier program is rewritten by extending the **Thread** class.

Extending Thread

```
class DemoThread extends Thread
{
DemoThread()
{
super("Demo Thread");
System.out.println("Child Thread:" + this);
start();
}

public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
}
```

[refer the eclipse code that I showed in class]

Extending Thread

```
cSystem.out.println("Child Thread Interrupted");
}
System.out.println ("Exiting Child Thread");
}
}
atch (InterruptedException e)
{

class ThreadImpl
{
public static void main(String args[])
{
new DemoThread();
try
{
for(int i=5; i>0; i--)
{
```

Extending Thread

```
System.out.println("Child Thread: " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Main Thread      Interrupted");
}
System.out.println("Main Thread Exiting");
}
}
```

Implementing Runnable or Extending Thread

A modeling heuristic pertaining to hierarchies says that classes should be extended only when they are enhanced or modified in some way

So, if the sole aim is to define an entry point for the thread:

- (by overriding the **run( )** method)
- and not override any of the **Thread** class' other methods

it is recommended to implement the **Runnable** interface

Creating Multiple Threads

You can launch as many threads as your program needs

The following example is a program spawning multiple threads:

```
class DemoThread implements Runnable
{
String name;
Thread t;

DemoThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New Thread: " + t);
t.start();
}
```

Creating Multiple Threads

```
public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(1000); }
}
catch (InterruptedException e)
{
System.out.println(name + "Interrupted");
}
System.out.println(name +"Exiting");
}
}
class MultiThreadImpl
{
public static void main(String args[])
{
```

Creating Multiple Threads

```
new DemoThread("One");
new DemoThread("Two");
new DemoThread("Three");
try
{
Thread.sleep(10000);
}
catch (InterruptedException e)
{
System.out.println("Main Thread Interrupted");
}
System.out.println("Main Thread Exiting");
}
}
```

Control Thread Execution - The **isAlive( )** & **join( )** methods

The main thread must always be the last thread to finish

Using **sleep( )** with a sufficiently long delay in **main( )** to allow the child threads to execute and terminate prior to the main thread is not foolproof

There is no way for the main thread to know that the child threads have indeed terminated within the time set by sleep( )

Control Thread Execution - The **isAlive( )** & **join( )** methods

Two ways exist by which you can determine whether a thread has finished:

The **isAlive( )** method will return true if the thread upon which it is called is still running; else it will return false.

The **join( )** method waits until the thread on which it is called terminates.

Join method in Java allows one thread to wait until another thread completes its execution. In simpler words, it means it waits for the other thread to die. It has a void type and throws InterruptedException.

Control Thread Execution - The IsAlive() & Join( ) methods

```
class DemoThread implements Runnable
{
String name;
Thread t;
DemoThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New Thread: " + t);
t.start();
}

public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(1000); }
}
```

Control Thread Execution - The IsAlive() & Join( ) methods

```
catch (InterruptedException e)
{
System.out.println(name + "Interrupted");
}
System.out.println(name +"Exiting");
}
}
class MultiThreadImpl
{
public static void main(String args[])
{
DemoThread t1 = new DemoThread("One");
DemoThread t2 = new DemoThread("Two");
DemoThread t3 = new DemoThread("Three");
System.out.println("Thread One is alive: " +   t1.t.isAlive());
System.out.println("Thread Two is alive: " +  t2.t.isAlive());
System.out.println("Thread Three is alive: "  + t1.t.isAlive());
try
{
System.out.println("Waiting for child  threads to finish");
t1.t.join();
t2.t.join();
t3.t.join();
}
```

Control Thread Execution - The IsAlive() & Join( ) methods

```
catch (InterruptedException e)
{
System.out.println("Main thread interrupted");
}
System.out.println("Thread One is alive: " +  t1.t.isAlive());
System.out.println("Thread One is alive: " +  t1.t.isAlive());
System.out.println("Thread One is alive: " +   t1.t.isAlive());
System.out.println("Main thread exiting");
}
}
```

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run

Higher-priority threads get more CPU time than lower-priority threads

A higher priority thread can also preempt a lower priority thread

Actually, threads of equal priority should evenly split the CPU time

Thread Priorities

For safety:

- Threads that share the same priority should yield the CPU once in a while
- This ensures that all threads have a chance to run in a non-preemptive operating system
- In a non-preemptive OS, most threads get a chance to run
- threads invariably encounter some blocking situation, such as sleeping for a specified time, or waiting for I/O

When this happens, the blocked thread is suspended, and other threads can run.

Thread Priorities

Every thread has a priority.

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

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.

The methods for accessing and setting priority are as follows:

- public final int getPriority( );
- public final void setPriority (int level);

The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively.

The normal priority is 5. To return a thread to default priority, specify **NORM_PRIORITY** which is 5. For your convenience, **java.lang.Thread** defines three **final** constants that can be used to set priorities. **MAX_PRIORITY** = 10;

MIN_PRIORITY = 1;

NORM_PRIORITY = 5;

Thread Priorities

When multiple threads are ready to be executed, the runtime system chooses the highest priority runnable thread to run

Only when that thread stops, yields, or becomes not runnable for some reason will a lower priority thread start executing

If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion

Thread Priorities

```
class Counter implements Runnable
{
int count = 0;
Thread t;
private volatile boolean running = true;
public Counter(int p_level)
{
t = new Thread(this);
t.setPriority(p_level);
}

public void run()
{
while (running)
{
count++;
}
}
```

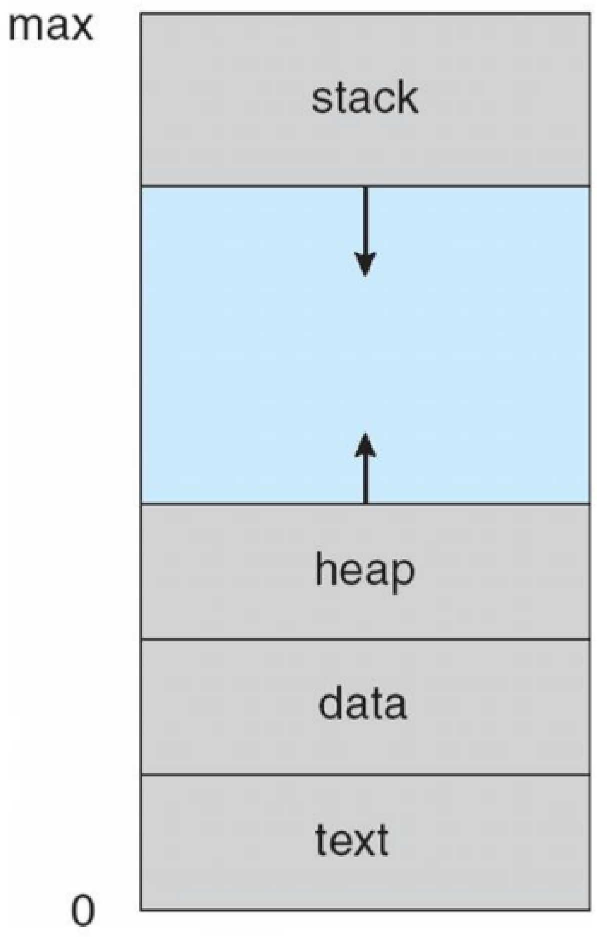
Thread Priorities

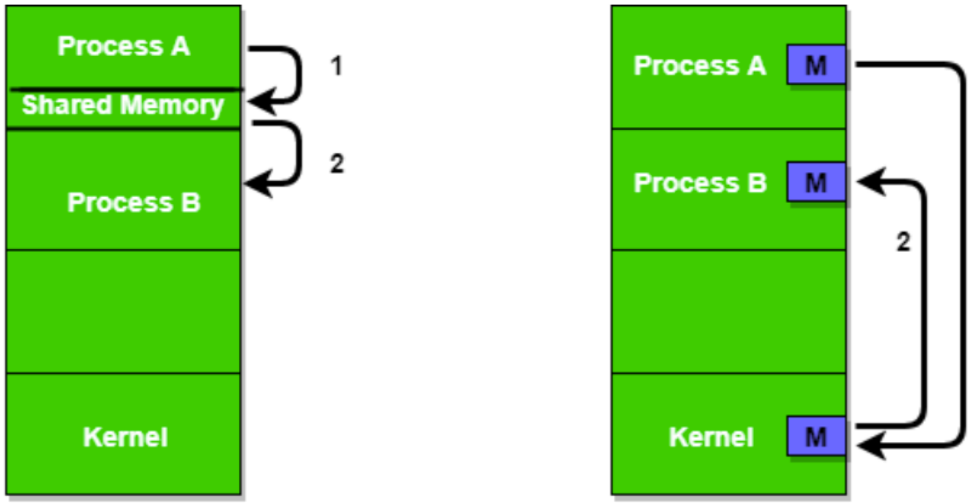
```
public void stop()
{
running = false;
}

public void start()
{
t.start();
}
}

class HighLowPriorityTest
{
public static void main(String args[])
{
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
Counter high = new Counter(Thread.NORM_PRIORITY+2);
Counter low = new  Counter(Thread.NORM_PRIORITY-2);
low.start();
high.start();
```

Thread Priorities

```
try
{
Thread.sleep(10000);
}
catch(InterruptedException e)
{
System.out.println("Main Thread  Interrupted");
}
low.stop();
high.stop();
try
{
low.t.join();
high.t.join();
}
catch(InterruptedException e)
{
System.out.println("interrupted Exception caught");
}
System.out.println("Low Priority Thread’s Iterations: " + low.count);
System.out.println("High Priority Thread’s Iterations: " + high.count);
}
}
```

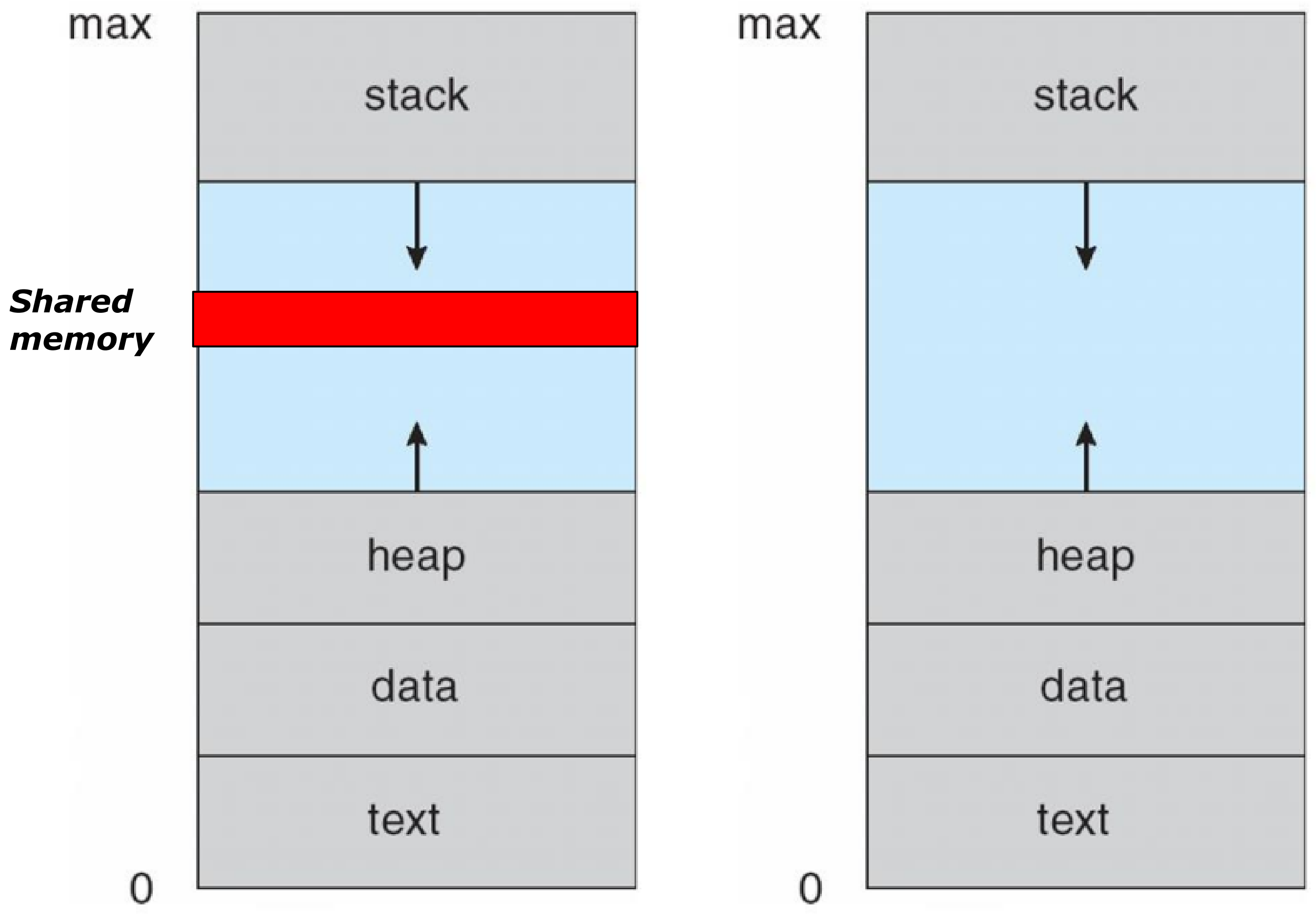
Through Inter Process/Thread Communication (IPC) models Cooperative processes/thread communicate.

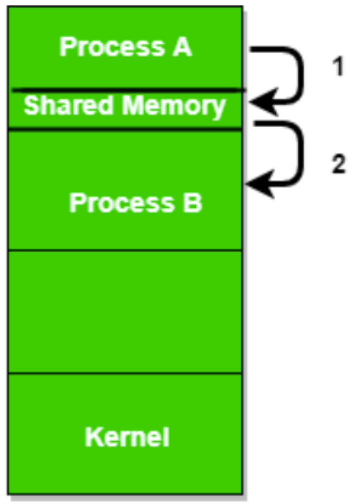




- ***Shared memory***
- ***Message passing***

Through Inter Process/Thread Communication (IPC) models Cooperative processes communicate.





Process/Thread A

Process/Thread B

- ***Shared memory***

Process Synchronization

- The Critical Section (CS): CS refers to the segment of code where the processes access shared resources, such as common variables, files and perform write operations on them.
    - When one process enters a CS, no other process is allowed to execute in its CS.
    - When two processes access and manipulate the shared resources concurrently, and the resulting execution outcome depends on the order in which processes access the resources, this is called ***"race condition".***
    - ***"Race condition"*** *leads to* inconsistent states of data. Therefore, we need a synchronization protocol that allows processes to co-operate while manipulating shared resources, which essentially is the CS problem.

Race Condition (Example 1)

```
P(){
    Read(a)
    a=a+1
    Write (a)
}
```

If a=10 and p1 , p2 executes serially then final value of a will be 12.

If process P1 context switch after Read(a) and then P2 executes (concurrent fashion), inconsistent result (a=11) may occur.

Race Condition (Example 2)

```
            int z = 10;
P1{                      |    p2(){
    int x = z            |       int y = z
    x = x+1              |        y = y-1
    z = x                |       z=y
}                        |    }
```

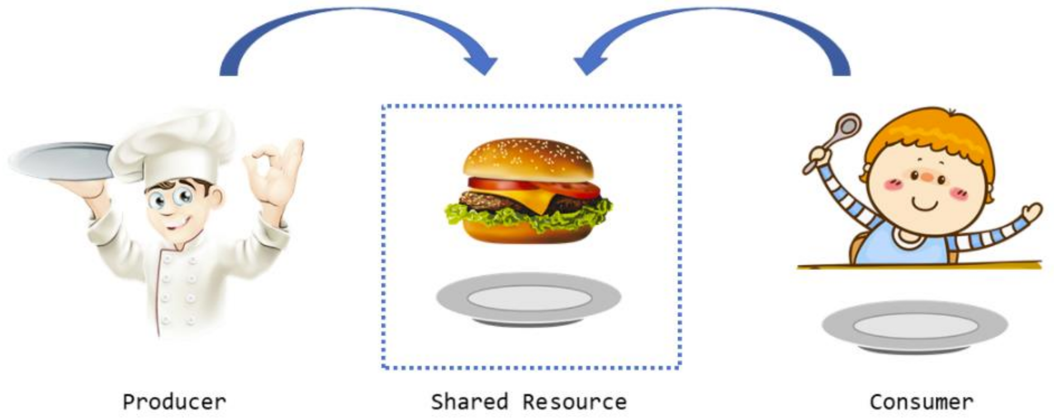
If process P1 and P2 executes serially then result is 10.

Race Condition (Example 2)

```
int z = 10;

P1{                         p2(){
    int x = z                   int y = z
    x = x+1                      y = y-1
    //context switch             //context switch
    z = x                       z=y
}                           }
```

If process P1 and P2 executes above code in concurrent fashion (p1 context switch after first instruction or p2 context switch after first instruction) inconsistent results (11/9) may come.

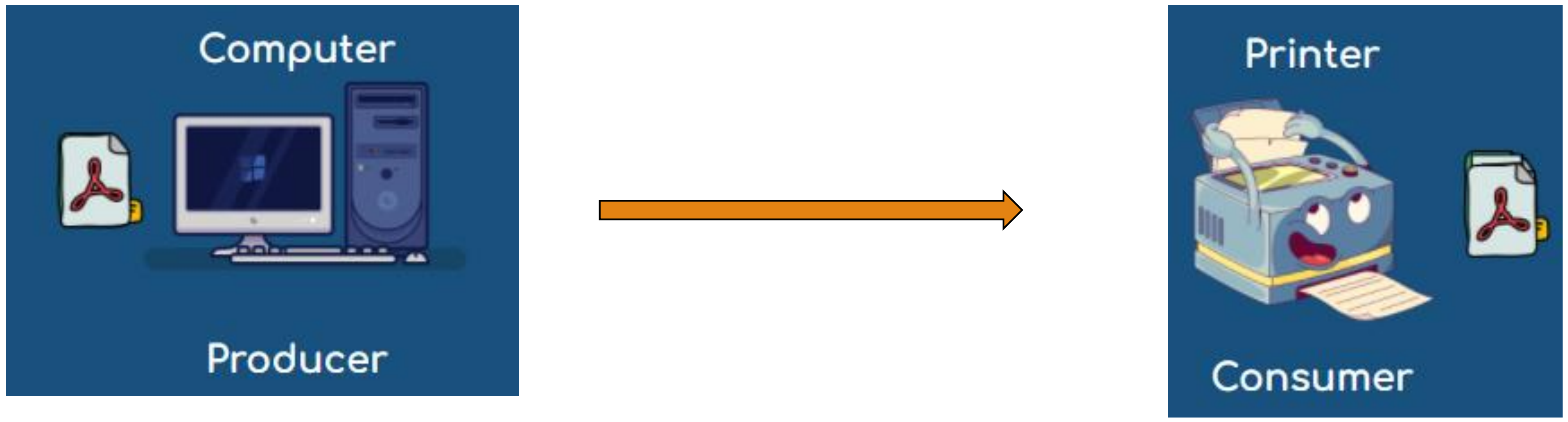
A producer Consumer Problem, Issues, Solutions



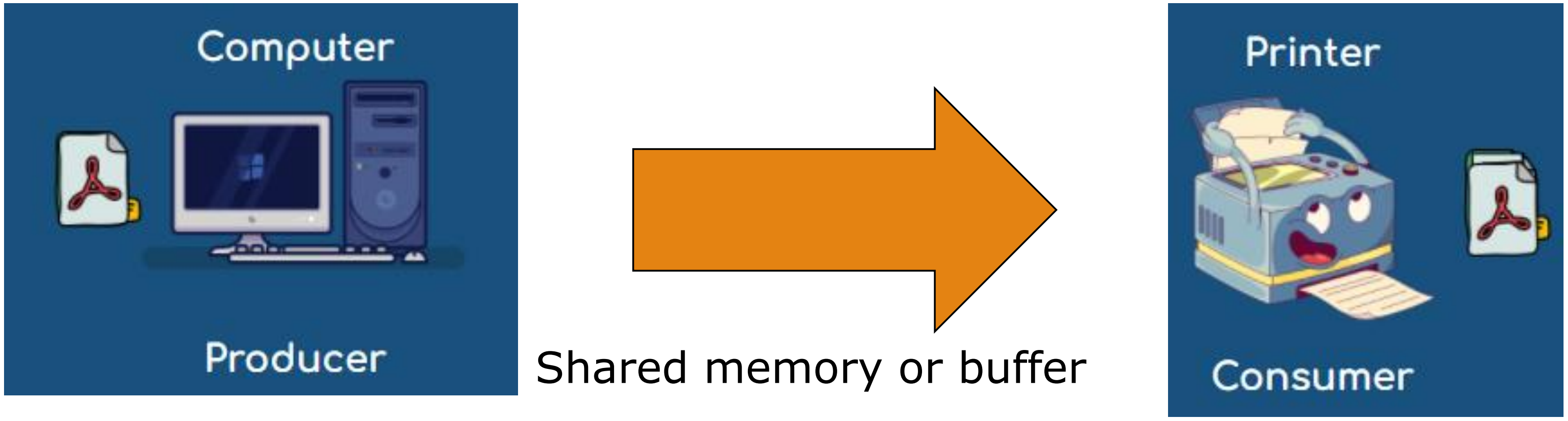
- It is a standard problem for multi-process synchronization
- Two processes: one is producer, another is consumer
- Assumption is, both try to run at the same time
- They are sharing some resource, i. e thet are co-operative processes
- A producer produces the item that is consumed by a consumer
- What if producer produces at a higher rate than a consumer can consume
- What if consumer consumes at a higher rate than a producer can produce

Race Condition (Producer Consumer)

- Both producer (computer) and consumer (printer) running concurrently
- Computer can send multiple print request
- Printer can print only one document at a time
- If computer sends request while printer is busy then request gets lost
- Computer must wait till printer gets free before sending print request



Race Condition (Producer Consumer)



•A possible solution for this problem would be use of shared memory (buffer)

•To allow producer and consumer concurrently a buffer is added that can be filled by producer and emptied by consumer

•The print requests from computer will be kept in the shared memory

•Consumer will pick one by one request and prints

•Producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced

Synchronization

Threads often need to share data

Need for a mechanism to ensure that the shared data will be used by only one thread at a time

This mechanism is called synchronization.

Key to synchronization is the concept of the **monitor** (also called a semaphore).

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them

To enter an object’s monitor, just call a **synchronized** method on the object

While a thread is inside a **synchronized** method, all other threads that try to call that synchronized method (or any other **synchronized** method on the same object) will have to wait

Using Synchronized Methods

```
class Callme
{
void call( String msg)
{
System.out.print( "[" + msg );
try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
{
System.out.println( "Interrupted");
}
System.out.println( "]" );
}
}
```

Using Synchronized Methods

```
class Caller implements Runnable
{
String msg;
Callme target;
Thread t;
public Caller (Callme targ, String s)
{
target = targ;
msg = s;
t = new Thread( this);
t.start( );
}

public void run( )
{
target.call( msg);
}
}
class Synch
{
```

Using Synchronized Methods

```
public static void main (String args[ ] )
{
Callme target = new Callme( );
Caller obj1 = new Caller( target, "Hello");
Caller obj2 = new Caller( target, "Synchronized");
Caller obj3= new Caller( target, "World");
try
{
obj1.t.join( );
obj2.t.join( );
obj3.t.join( );
}
catch (InterruptedException e)
{
  System.out.println( "Interrupted");
}
}
}
```

Using Synchronized Methods

To fix this problem:

- You must **serialize** access to **call( )**
- You must restrict its access to only one thread at a time.
- You need to precede **call( )**'s definition with the keyword **synchronized**

```
class Callme {
synchronized void call( String msg){…….. } }
class Callme {
   void call( String msg)
    { System.out.print( "[" + msg )
      try {
            Thread.sleep(1000);
           }catch (InterruptedException e){
        System.out.println( "Interrupted");  }
     System.out.println( "]" ) } }
class Caller implements Runnable {
   String msg;
```

The Synchronized Statement/Block

```
Callme target;
 Thread t;
public Caller (Callme targ, String s)
   { target = targ;
     msg = s;
     t = new Thread( this);
     t.start( );  }
public void run( )
{ synchronized(target) { //synchronized block
    target.call( msg); }}
class Synch { public static void main (String args[ ] )
  {Callme target = new Callme( );
    Caller obj1 = new Caller( target, “Hello”);
    Caller obj2 = new Caller( target, “Synchronized”);
    Caller obj1 = new Caller( target, “World”);
 try {ob1.t.join( );
   ob2.t.join( );
   ob3.t.join( ); }
catch (InterruptedException e) { System.out.println( “Interrupted”); }}}
```

•Synchronized block is used to lock an object for any shared resource.

•Scope of synchronized block is smaller than the method.

```
synchronized (object refer
ence expression) {
  //code block
}
```

Inter-Thread Communication

Threads are often interdependent - one thread depends on another thread to complete an operation, or to service a request

The words ***wait*** and ***notify*** encapsulate the two central concepts to thread communication

- A thread waits for some condition or event to occur
- You notify a waiting threading that a condition or event has occurred

Java’s elegant inter-thread communication mechanism uses:

- **wait( )**
- **notify( ), and notifyAll( )**
- **After Java 5, we have** the same interprocess communication mechanism via the await(), signal(), and signalAll() methods. All three methods can be called only from within a synchronized context.

Inter-thread Communication

Wait( ), notify( ) and notifyAll( ) are:

- Declared as **final** in **Object**
- Hence, these methods are available to all classes
- These methods can only be called from a **synchronized** context

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

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

notifyAll( ) wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first.

Inter-thread Communication

The following sample program incorrectly implements a simple form of the producer/consumer problem.

It consists of four classes namely:

- **Q**, the queue that you are trying to synchronize
- **Producer**, the threaded object that is producing queue entries
- **Consumer**, the threaded object that is consuming queue entries
- **PC**, the class that creates the single Queue, Producer, and Consumer

Inter-thread Communication

```
class Q
{
int n;
synchronized int get( )
{
System.out.println( "Got: " + n);
return n;
}
synchronized void put( int n)
{
this.n = n;
System.out.println( "Put: " + n);
}
}
class Producer implements Runnable
{
Q q;
Producer ( Q q)
{
this.q = q;
new Thread( this, "Producer").start( );
}
```

Inter-thread Communication

```
public void run( )
{
int i = 0;
while (true)
{
q.put (i++);
}
}
}
class Consumer implements Runnable
{
Q q;
Consumer ( Q q)
{
this.q = q;
new Thread (this, "Consumer").start( );
}
public void run( )
{
while (true)
{
q.get( );
}
```

Inter-thread Communication

```
}
}
class PC
{
public static void main (String args [ ] )
{
Q q = new Q( );
new Producer (q);
new Consumer (q);
System.out.println("Press Control-C to stop");
}
}
```

Inter-thread Communication Using wait( ) & notify( )

```
class Q
{
int n;
boolean valueset = false;
synchronized int get( )
{
if (!valueset)
try
{
wait( );
}
catch (InterruptedException e)
{
System.out.println("InterruptedException caught");
}
System.out.println( "Got: " + n);
valueset = false;
notify( );
return n;
}
```

Inter-thread Communication

```
synchronized void put( int n)
{
if (valueset)
try
{
wait( );
}
catch (InterruptedException e)
{
System.out.println("InterruptedException caught");
}
this.n = n;
valueset = true;
System.out.println( "Put: " + n);
notify( );
}
}


class Producer implements Runnable
{
Q q;
Producer ( Q q)
```

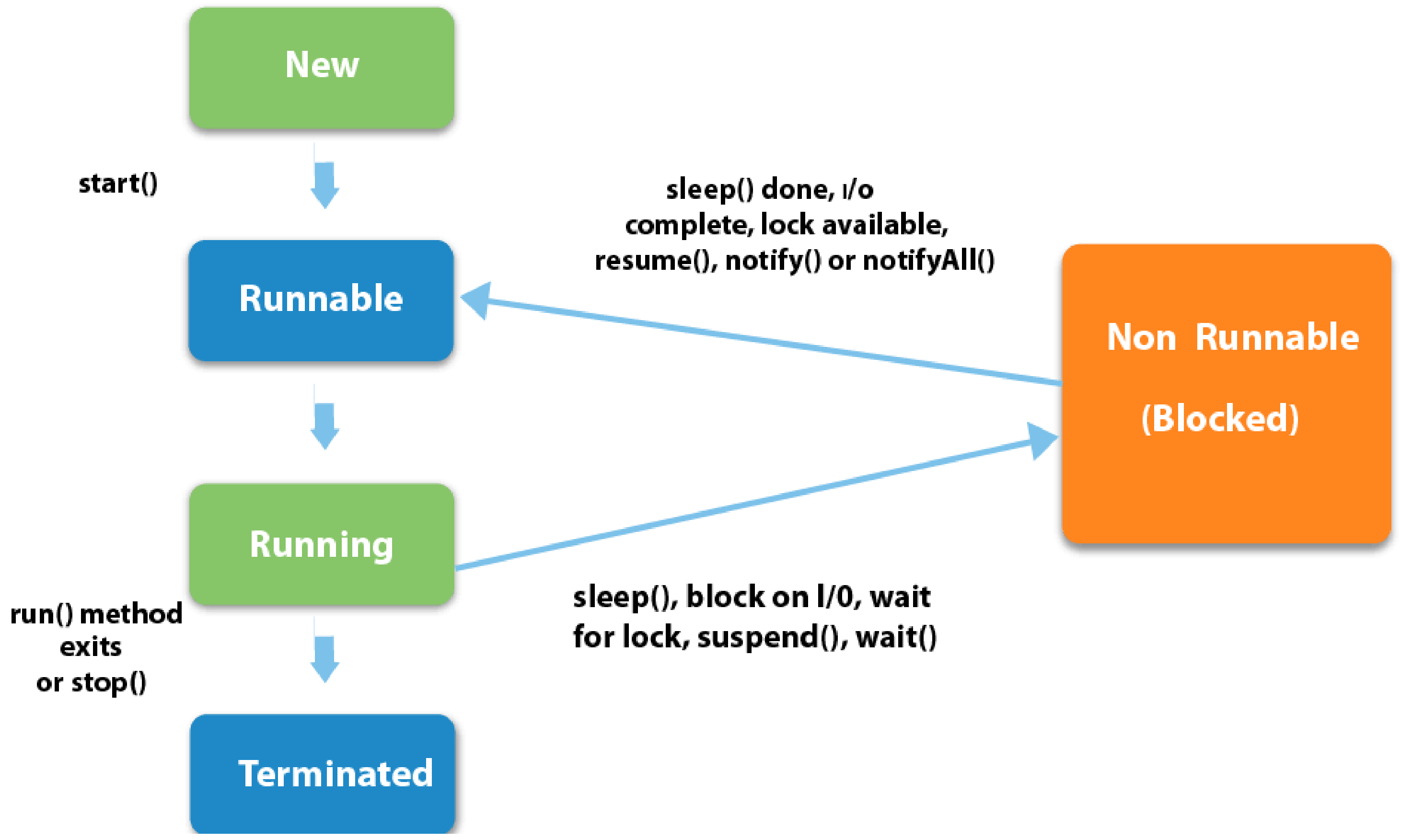
Inter-thread Communication

```
{
this.q = q;
new Thread( this, "Producer").start( );
}
public void run( )
{
int i = 0;
while (true)
{
q.put (i++);
}
}
}
class Consumer implements Runnable
{
Q q;
Consumer ( Q q)
{
this.q = q;
```

Inter-thread Communication

```
new Thread (this, "Consumer").start( );
}
public void run( )
{
while (true)
{
q.get( );
}
}
}
class PC
{
public static void main (String args [ ] )
{
Q q = new Q( );
new Producer (q);
new Consumer (q);
System.out.println("Press Control-C to stop");
}
}
```

Life cycle of a Thread (Thread States)


New
start()
Runnable
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable (Blocked)
Running
sleep(), block on I/0, wait for lock, suspend(), wait()
run() method exits or stop()
Terminated

Life cycle of a Thread (Thread States)

1. **New:**

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2. **Runnable:**

The thread is in runnable state after invocation of start() method, but the thread scheduler ha not selected it to be the running thread.

3. **Running:**

The thread is in running state if the thread scheduler has selected it.

4. **Non-Runnable (Blocked):**

This is the state when the thread is still alive, but is currently not eligible to run.

5. **Terminated:**

A thread is in terminated or dead state when its run() method exits.

Suspending, Resuming, and Stopping Threads:

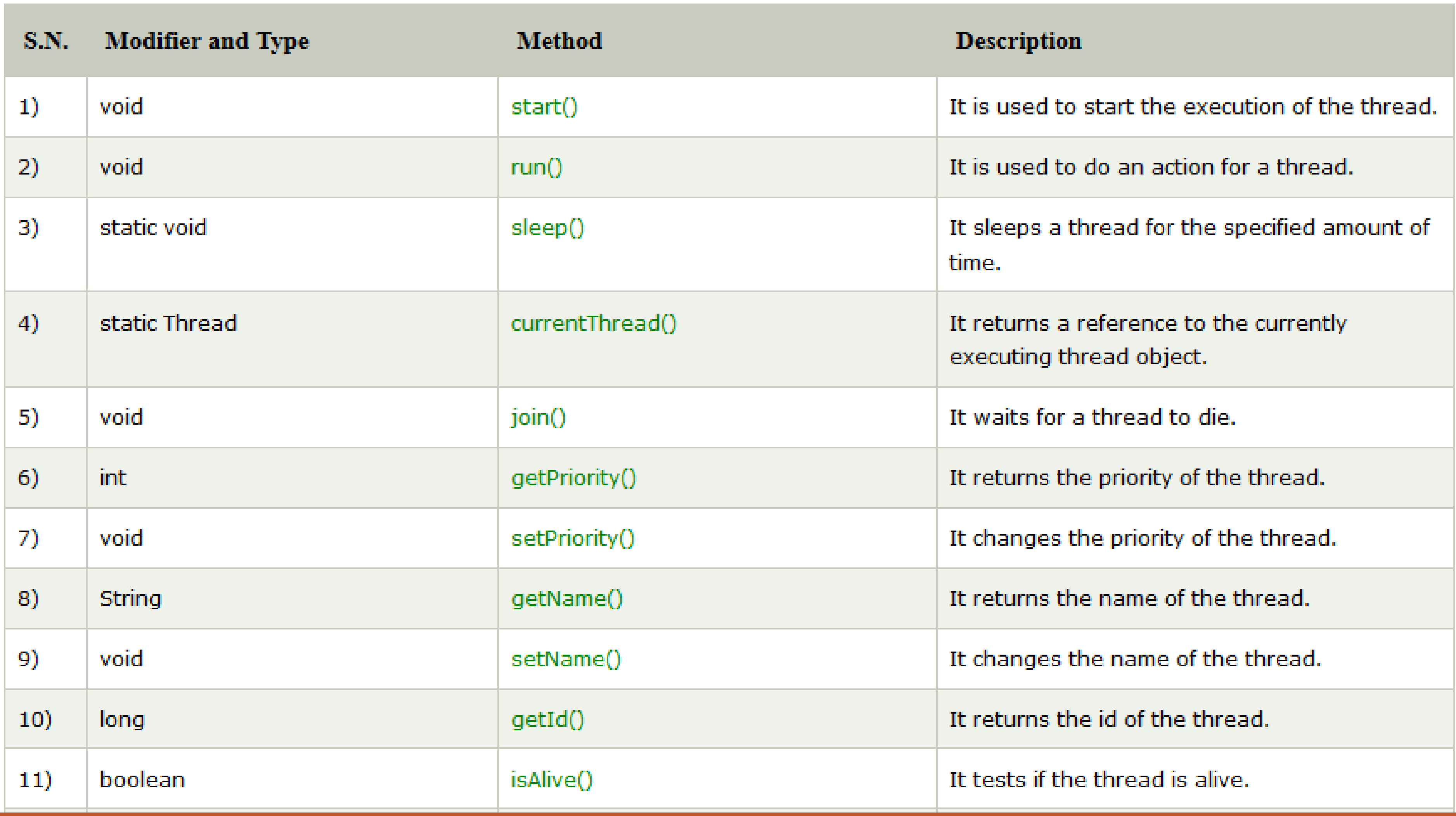
- This is accomplished by establishing a flag variable that indicates the execution state of the thread.

- As long as this flag is set to:
  - "running", the run() method must continue to let the thread execute.
  - If this variable is set to "suspend", the thread must Pause.
  - If it is set to "stop", the thread must terminate.
  - Refer to the code with file name SuspendResume1.java.

Interrupting a Thread:

- If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.
- If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interruptflag to true.
- The 3 methods provided by the Thread class for interrupting a thread:
  - public void interrupt()
  - public static boolean interrupted()
  - public boolean isInterrupted()

Commonly used methods of Thread class:



S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.