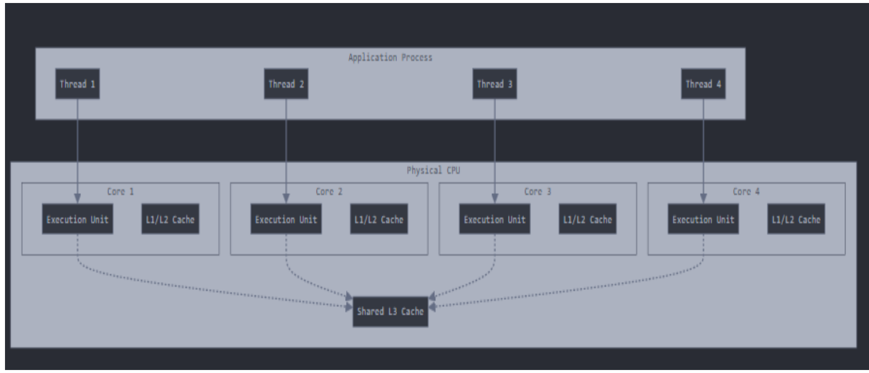
CS6308- Java Programming

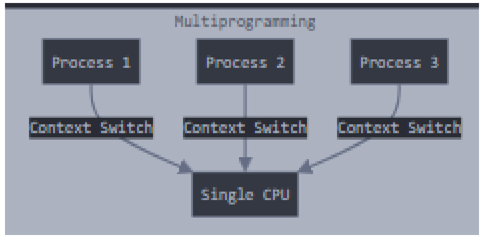
V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

CONCURRENT PROGRAMMING

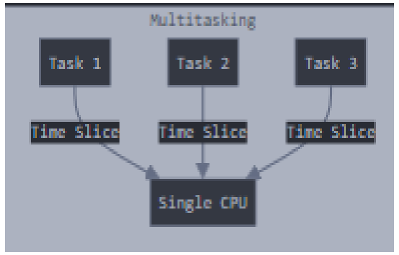
- **Concurrency:**
  - The idea of two program parts, or even two separate programs, executing *Concurrently or simultaneously*
  - Concurrency also occurs in all four programming paradigms-imperative, object oriented, functional, and logic.
  - multicore processors, such as Intel i9 and Apple M3 which feature multiple cores can run threads in parallel.
  - save enormous amounts of computing resources, both in space and in speed.
- **Concurrency types**
  - The single processor setting
    - **Multithreading** : a program runs on a single processor, but it can dynamically divide into concurrent *threads* of control from time to time.
  - The interprocess communication (IPC) setting.
    - ***client-server***: a program is viewed as a collection of cooperating processes that run over a network and share data.



CONCURRENT PROGRAMMING



- **Concurrency:**
  - A concurrent program is a program designed to have **two or more execution contexts executing in parallel**.
  - To model the parallelism in the real world: Air traffic control
    - Parallel execution of the program on more than one processor will be much more difficult to achieve
  - Concurrency also occurs in all four programming paradigms-imperative, object oriented, functional, and logic.
  - save enormous amounts of computing resources, both in space and in speed.
  - to provide multiple simultaneous services to the user.

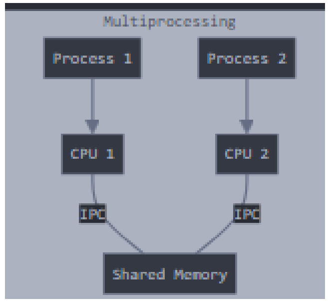


Multiprogramming

Multiple processes in executions on a **single processor**, Concept of Context Switching is used. increases CPU utilization. More than one process in execution. share memory

Multitasking

tasks multiplex their executions on a **single processor.** Multiple tasks within processes through time-sharing and Context Switching. increases CPU utilization, it also increases responsiveness. share memory



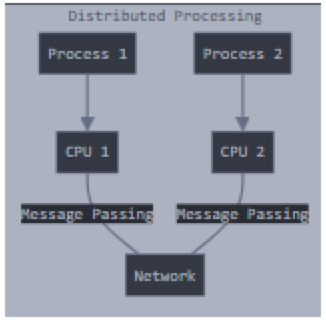
Multiprocessing

Multiple processes running truly parallel on a **multiprocessor** system.

IPC (Inter-Process Communication) communication . share memory

Distributed Processing

tasks multiplex their executions on **several independent processors** which do not share memory



- **Concurrency types: process vs thread**
- Both process and Thread are independent paths of execution but one process can have multiple Threads.

1. The single processor setting
   - **Multithreading** : a program runs on a single processor, but it can dynamically divide into concurrent *threads* of control from time to time.
     - **multiple threads are executing in a process at the same time**
     - multiple threads to be created within a process, executing independently but concurrently sharing process resources more easily(heap memory, file descriptors) but each Thread has its own Exception handler and own stack
     - provides concurrency within the context of a single process, i.e a separate flow of control that occurs within a process
   - Advantage
     - requires less processing overhead than multiprogramming because concurrent threads are able to share common resources more easily.
     - CPU switches between multiple threads of the same program is simple as it uses wait and notify for example in java
   - Disadvantage
     - Without the proper use of locking mechanisms, data inconsistency and dead-lock situations can arise. Thread starvation and resource contention issues arise if many threads are trying to access a shared resource
   - Example
     - A web server will utilize multiple threads to simultaneous process requests for data at the same time.
     - An image analysis algorithm will spawn multiple threads at a time and segment an image into quadrants to apply filtering to the image.
     - While typing, multiple threads are used to display your document, asynchronously check the spelling and grammar of your document
     - Multiple threads of execution are used to load content, display animations, play a video, and so on.
2. The interprocess communication (IPC) setting.
   - ***Multitasking /client-server***: a program is viewed as a collection of cooperating processes that run over a network and share data.
     - **multiple processes are executing at the same time**
     - Every process has its own memory space, executable code, and a unique process identifier (PID)
     - provides concurrency between processes.
     - logically concurrent execution of multiple programs ,a separate process for each program that can run in parallel
   - Disadvantage
     - CPU switches between multiple programs to complete their execution in real time, duplicate resources and share data as the result of more time-consuming interprocess communication.
     - context switching from one process to another is expensive ; Process are heavyweight and require their own address space
   - Example
     - Play MP3 music, edit documents in MS Word, surf the Google **Chrome** ,**Firefox** use **multi-threading** for their **tabs**, while **Chrome** use multi-processes ,

History and Definitions

- **Uniprogramming**: *one thread at a time*
  - MS/DOS
- **Multiprogramming:** *more than one thread at a time*
  - MULTIX, UNIX, OS/2, Windows NT/2000/XP
- **Concurrent programming** was used in early operating systems to support parallelism in the underlying hardware and to support multiprogramming and time-sharing.
- A ***parallel program*** is a concurrent program in which several execution contexts, or threads, are active *simultaneously*
- A ***distributed program*** is a concurrent program that is designed to be executed simultaneously on a network of autonomous processors that do not share main memory, with each thread running on its own separate processor.

Multithreading

Outline

- Single thread vs Multi thread
- Thread vs Process
- Runnable vs Thread

Thread

- What is Thread?
  - A **thread** is an independent path of execution within a program.
- Why Thread?
  - a **thread** is the information needed to serve one individual user or a particular service request.
- Is Java single threaded?
  - Every **thread in Java** is created and controlled by the **java**.lang.**Thread** class.
  - Every Java application has minimum two threads
    - Main thread
    - Garbage collector thread
  - The Java Virtual Machine allows an application to have multiple threads of execution running concurrently

Single Thread

- Single task at a time.

```
Class Test{
    Public static void main(String[] args){
        Add aobj = new Add(5,3);
        Sub sobj =new Sub(5,3);
        //…
        }
```

Single thread

Task 1

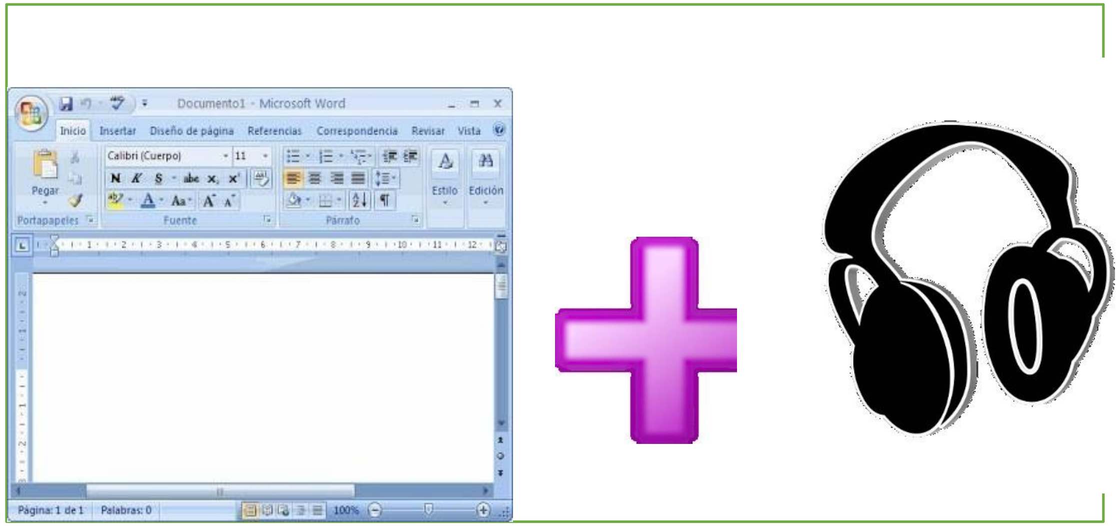
Task 2 will start after completion of Task1

Execution time is more.

Multithread (multitask)

Multiple task possible for human at a time !

Do computers perform Multitask at a time?



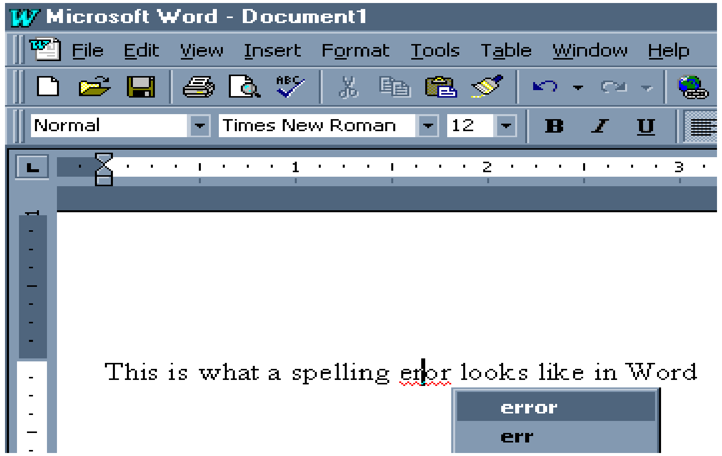
Use of multithreads

- Reduce excecution time
  - Doing tasks in parallel
- Asynchronous tasks
- Web applications-server/client
- Computer games-animations

Process vs Thread

Process-heavy weight process

Sequential Program Execution Stream



Each application is a process
Eg. Java application, Ms word , MS power point, etc

Thread –Light weight process

Sub process of a process are threads

- Type
- Spell check
- Grammar check

Threads are unit of process

Thread 1 Thread 2 Thread 3

Typing	Spell check	Grammar check	…

Thread Control and Communication

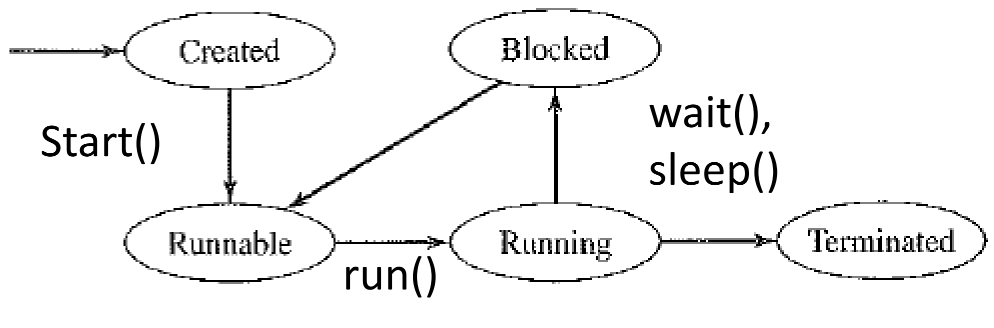
1 Created: the thread has been created, but is not yet ready to run.

2Runnable: the thread is ready to run (sometimes this state is called *ready*). The thread awaits receiving a processor to run on.

3 Running: the thread is executing on a processor.

4 Blocked: the thread is either waiting to enter a section of its code that requires exclusive access to a shared resource (variable), or else has voluntarily given up its processor.

5 Terminated: the thread has stopped and cannot be restarted.



States of a Thread

Thread transfers back and forth between the *Blocked* and *Running* states :
For example, a thread may be sending several documents to a printer queue, but may need to wait until after the successful printing of one document before it sends a later one.

Thread Control and Communication

- Concurrent programs require **interthread communication** or interaction.
- **Communication** occurs for the following reasons:

  1 A thread sometimes requires exclusive access to a shared resource, like a printer queue, a terminal window, or a record in a data file.

  2 A thread sometimes needs to exchange data with another thread.

- In both cases the two communicating threads must **synchronize** their execution to avoid conflict when **acquiring resources**, or to make contact when **exchanging data.**
- A thread can communicate with other threads through:

  1 **Shared variables**: this is the primary mechanism used by Java, and it can also be used by Ada.

  2 **Message passing**: this is the primary mechanism used by Ada.

  3 **Parameters:** this is used by Ada in conjunction with message passing.

- Threads normally cooperate with one another to solve a problem.
- However, it is highly desirable to keep communication between threads to a minimum;

Races and Deadlocks

- Two fundamental problems that can occur while executing two different threads asynchronously are ***race conditions* and *deadlocks***.
- **Definition:** A ***race* condition** (sometimes called a *critical race)* occurs when the resulting value of a variable, when two different threads of a program are writing to it, will differ depending on which thread writes to it first.
- The result is determined by which of the threads wins a "race" between them, since it depends on the order in which the individual operations are interleaved over time.

c = c + 1:

JVM target code
1 load *c*
2 add 1
3 store c

- In fact, as the number of threads trying to execute this code increases, the resulting number of distinct values computed for c can vary between 1 and the number of threads!
- Two thread a and B, Then the resulting value of c depends critically on whether A or B completes step 3 before the other one begins step 1. If so, then the resulting value of c is 2; otherwise the resulting value of c is 1.

Races and Deadlocks

- A thread wishing to acquire a *shared resource,* such as a file or a shared variable (like c in the above example), must first acquire access to the resource.
- When the resource is no longer required, the thread must relinquish access to the resource so that other threads can access it.
  - If a thread is unable to acquire a resource, its execution is normally suspended until the resource becomes available.
- Resource acquisition must be administered so that no thread is unduly delayed or denied access to a resource that it needs.
- An occurrence of the latter is often called ***lockout* or *starvation*.**

Races and Deadlocks

- Errors that occur in a concurrent program may appear as ***transient errors***.
- These are errors that *may or may not occur*, depending on the execution paths of the various threads.
- *Finding a transient error* can be extremely difficult because the sequence of events that caused the occurrence of the fault may not be known or reproducible.
- Unlike sequential programming, *rerunning the same program* on the same data may not reproduce the fault.
- Inserting *debugging* output itself may alter the behavior of the concurrent program so as to prevent the fault from reoccurring.
- Thus, designing a concurrent program is the ability to express it in a form that guarantees the **absence of Critical races**.

Races and Deadlocks

- **The code inside a thread that accesses a shared variable or other resource is termed a *critical section.***
- **For a thread to safely execute a critical section, it needs to have access to a locking mechanism; such a mechanism must allow a lock to be tested or set as** a single atomic instruction**.**
- **Locking mechanisms are used to ensure that only a single thread is executing a critical section (hence accessing a shared variable) at a time;**
- This can eliminate critical race conditions and one such locking mechanism is called a *semaphore*.

Races and Deadlocks

- The second fundamental problem that can occur while executing two different threads asynchronously is called a ***deadlock.***
- **Definition:** A *deadlock* occurs when a thread is waiting for an event that will never happen.
- A deadlock normally involves several threads, each waiting for access to a resource held by another thread.
- A classical example of a deadlock is a traffic jam at an intersection where each car entering is blocked by another
- Four necessary conditions must occur for a deadlock to exist [Coffman *et al.*, 1971]:

  1 Threads must claim exclusive rights to resources.

  2 Threads must hold some resources while waiting for others; that is, they acquire resources piecemeal rather than all at once.

  3 Resources may not be removed from waiting threads (no preemption).

  4A circular chain of threads exists in which each thread holds one or more resources required by the next thread in the chain.

- **A thread is said to be *indefinitely postponed* if it is delayed awaiting an event that may never occur: Unfairness**
- **Neglecting fairness in designing a concurrent system may lead to indefinite postponement, thereby rendering the system unusable**

Multithreads:
By extending Thread class

Multiple task at a time.

```
Class Test{
public static void main(String[] args)
{
        Add aobj = new Add(5,3);
        Sub sobj =new Sub(5,3);
        //…
        aobj.start();
        sobj.start();}
        }
```

Start method Invokes run method: **creates new thread**

```
Class Add extends Thread{
    int num1;
    int num2
    Add(int a, int b){
        num1 = a;  num2 =b;
    }
    public void run(){
    System.out.println("sum="+ num1 +num2);
    }
}
```

Subclass extends superclass unless the programmer intends on modifying or enhancing the fundamental behavior of the superclass.

Runnable interface should be used if you are only planning to override the run() method and no other Thread methods.

Multithreads:
By implementing Runnable Interface

```
Class Test{
public static void main(String[] args)
{
        Add aobj = new Add(5,3);
        Sub sobj =new Sub(5,3);
        //...
        Thread t1=new Thread( aobj );
        Thread t2=new Thread(sobj);
        t1.start();
        t2.start();
        }
        }
```

Pass runnable ref object to thread

Start method

Invokes run method: **creates new thread**

```
Class Add implements Runnable{
        int num1;
        int num2
        Add(int a, int b){
                num1 = a;  num2 =b;
        }
        public void run(){
        System.out.println("sum="+ num1 +num2);
        }
}
```

Runnable interface should be used only if run() method is overridden but not other Thread methods are enhanced.

Thread constructors

Constructor Summary
Thread()   Allocates a new Thread object.
Thread(Runnable target)   Allocates a new Thread object.
Thread(Runnable target, String name)   Allocates a new Thread object.
Thread(String name)   Allocates a new Thread object.
Thread(ThreadGroup group, Runnable target)   Allocates a new Thread object.
Thread(ThreadGroup group, Runnable target, String name)   Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
Thread(ThreadGroup group, Runnable target, String name, long stackSize)   Allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified *stack size*.

```java
//multitask using Runnable interface
class Add implements Runnable{
		int num1,num2;
		Add(int a, int b){
				num1=a; num2=b;
		}
		public void run(){
		for(int i=1; i<5; i++)
		 System.out.println("sum=" + (num1+i) + "+" +
(num2+i) +"="+ ((num1+i) + (num2+i)));
		}
}

class Sub implements Runnable{
		int num1,num2;
		Sub(int a, int b){
				num1=a; num2=b;
		}
		public void run(){
		   for(int i=1; i<5; i++)
		      System.out.println("difference=" + (num1+i) + "-"
+ (num2+i)+"="+ ((num1+i) - (num2+i)));
		}
}
```

```java
class Test{
public static void main(String[] args){
		Add aobj=new Add(5,3);
		Sub sobj=new Sub(5,3);
		//--->invoke Add and Sub class run() method
		//-->reference object of Add and Sub class is passed
		Thread t1=new Thread(aobj);
		Thread t2=new Thread(sobj);
		t1.start();
		t2.start();
}
}
```

Command Prompt

```
C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java Test
sum=6+4=10
difference=6-4=2
sum=7+5=12
difference=7-5=2
sum=8+6=14
difference=8-6=2
sum=9+7=16
difference=9-7=2

C:\Program Files\Java\jdk1.8.0_131\bin>
```

```java
//multitask using Thread class
class Add extends Thread{
        int num1,num2;
        Add(int a, int b){
                num1=a; num2=b;
        }
        public void run(){
        for(int i=1; i<5; i++)
                System.out.println("sum=" + (num1+i) + "+"
+ (num2+i) +"="+ ((num1+i) + (num2+i)));
        }
}

class Sub extends Thread{
        int num1,num2;
        Sub(int a, int b){
                num1=a; num2=b;
        }
        public void run(){
                for(int i=1; i<5; i++)
                System.out.println("difference=" +
(num1+i) + "-" + (num2+i)+"="+ ((num1+i) - (num2+i)));
        }
}
```

```java
class TestDemo{
public static void main(String[] args){
        Add aobj=new Add(5,3);
        Sub sobj=new Sub(5,3);
        aobj.start();
        sobj.start();
}
}
```

Command Prompt

```
C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/TestDemo.java

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestDemo
difference=6-4=2
difference=7-5=2
difference=8-6=2
difference=9-7=2
sum=6+4=10
sum=7+5=12
sum=8+6=14
sum=9+7=16

C:\Program Files\Java\jdk1.8.0_131\bin>_
```

```java
class Add extends Thread{
            int num1,num2;
            Add(int a, int b){
                        num1=a; num2=b; }
            public void run(){
                for(int i=1; i<5; i++)
                        System.out.println("sum=" + (num1+i) + "+"
                        + (num2+i) +"="+ ((num1+i) + (num2+i)));
             try{
                        Thread.sleep(1000);   }
             catch(Exception e){}
             }
}
class Sub extends Thread{
            int num1,num2;
            Sub(int a, int b){
                        num1=a; num2=b; }
            public void run(){
               for(int i=1; i<5; i++)
                  System.out.println("difference=" + (num1+i) + "-"
                        + (num2+i)+"="+ ((num1+i) - (num2+i)));
               try{     Thread.sleep(1000);  }
             catch(Exception e){}
            } }
```

```java
class TestDemo{
public static void main(String[] args){
            Add aobj=new Add(5,3);
            Sub sobj=new Sub(5,3);
            aobj.start();
            sobj.start();
}
}
```

Command Prompt

```
C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/Test.java

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java Test
sum=6+4=10
difference=6-4=2
sum=7+5=12
difference=7-5=2
sum=8+6=14
difference=8-6=2
sum=9+7=16
difference=9-7=2

C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/TestDemo.java

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestDemo
sum=6+4=10
difference=6-4=2
sum=7+5=12
difference=7-5=2
sum=8+6=14
sum=9+7=16
difference=8-6=2
difference=9-7=2
```

Write a complete subclass of *Thread* to represent a thread that writes out the numbers from 1 to 10. Then write some code that would create and start a thread belonging to that class.

```
public class CountingThread extends Thread {
public static run() {
for (int i = 1; i <= 10; i++)
System.out.println(i);
}
 }
```

```
CountingThread counter; // Declare a variable to represent a thread.
// Create the thread object.
counter = new CountingThread();
// Start the thread running.
or

counter.start();

//create thread object and start thread
new CountingThread().start();
```

```
class Person implements Runnable{
public static String winner=null;
public void race()
{
  for(int meters=0; meters<=10; meters++){
            if(Person.winner==null){
                  if(meters==10){
                        Person.winner=Thread.currentThread().getName();
                        System.out.println("Winner ="+ Person.winner + "meters
                        run=" + meters);
                        break;}
               else
                        System.out.println(Thread.currentThread().getName()+
                        "meters run=" + meters);
            }
   }
}
public void run()
  { this.race();
   }
}
```

```
class TestRace{
public static void main(String[] args){
            Person p1=new Person();
            Person p2=new Person();
            Person p3=new Person();
            Person p4=new Person();
            Thread t1=new Thread(p1,"KAMAL");
            Thread t2=new Thread(p2,"RAJINI");
            Thread t3=new Thread(p3,"AJITH");
            Thread t4=new Thread(p4,"VIJAY");
            t1.start();
            t2.start();
            t3.start();
            t4.start();}}
```

```
C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/TestRace.java

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestRace
VIJAYmeters run=0
VIJAYmeters run=1
VIJAYmeters run=2
AJITHmeters run=0
AJITHmeters run=1
AJITHmeters run=2
RAJINImeters run=0
RAJINImeters run=1
RAJINImeters run=2
KAMALmeters run=0
KAMALmeters run=1
KAMALmeters run=2
KAMALmeters run=3
RAJINImeters run=3
AJITHmeters run=3
AJITHmeters run=4
AJITHmeters run=5
AJITHmeters run=6
AJITHmeters run=7
VIJAYmeters run=3
AJITHmeters run=8
AJITHmeters run=9
RAJINImeters run=4
KAMALmeters run=4
Winner =AJITHmeters run=10
VIJAYmeters run=4
```

```
KAMALmeters run=4
Winner =AJITHmeters run=10
VIJAYmeters run=4

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestRace
RAJINImeters run=0
RAJINImeters run=1
KAMALmeters run=0
KAMALmeters run=1
KAMALmeters run=2
KAMALmeters run=3
KAMALmeters run=4
KAMALmeters run=5
AJITHmeters run=0
VIJAYmeters run=0
VIJAYmeters run=1
VIJAYmeters run=2
VIJAYmeters run=3
AJITHmeters run=1
KAMALmeters run=6
KAMALmeters run=7
KAMALmeters run=8
KAMALmeters run=9
RAJINImeters run=2
Winner =KAMALmeters run=10
AJITHmeters run=2
VIJAYmeters run=4
```

```
class Person implements Runnable{
public static String winner=null;
public synchronized void race()
{
  for(int meters=0; meters<=10; meters++){
              if(Person.winner==null){
                    if(meters==10){
                              Person.winner=Thread.currentThread().getName();
                              System.out.println("Winner ="+ Person.winner + "meters
                              run=" + meters);
                              break;}
                else
                              System.out.println(Thread.currentThread().getName()+
                              "meters run=" + meters);
              }
   }
}
public void run()
  { this.race();
   }
}
```

```
class TestRace{
public static void main(String[] args){
               Person p1=new Person();
               Person p2=new Person();
               Person p3=new Person();
               Person p4=new Person();
           Thread t1=new Thread(p1,"Elon musk");
           Thread t2=new Thread(p2,"Jeff Bezos");
           Thread t3=new Thread(p3," Zuckerberg");
           Thread t4=new Thread(p4,"Bill gates");
               t1.start();
               t2.start();
               t3.start();
               t4.start();}}
```

```
Elon musk meters run=0
Elon musk meters run=1
Bill gates meters run=0
Elon musk meters run=2
Zuckerberg meters run=0
Bill gates meters run=1
Zuckerberg meters run=1
Zuckerberg meters run=2
Zuckerberg meters run=3
Zuckerberg meters run=4
Zuckerberg meters run=5
Zuckerberg meters run=6
Zuckerberg meters run=7
Jeff Bezos meters run=0
Jeff Bezos meters run=1
Elon musk meters run=3
Elon musk meters run=4
Elon musk meters run=5
Elon musk meters run=6
Elon musk meters run=7
Jeff Bezos meters run=2
Jeff Bezos meters run=3
Zuckerberg meters run=8
Bill gates meters run=2
Zuckerberg meters run=9
Jeff Bezos meters run=4
Elon musk meters run=8
Bill gates meters run=3
Winner =Zuckerbergmeters        run=10
```

```
Elon musk meters run=0
Elon musk meters run=1
Elon musk meters run=2
Elon musk meters run=3
Elon musk meters run=4
Elon musk meters run=5
Elon musk meters run=6
Elon musk meters run=7
Elon musk meters run=8
Elon musk meters run=9
Zuckerberg meters run=0
Bill gates meters run=0
Jeff Bezos meters run=0
Winner =Elon muskmeters         run=10
```

SYNCHRONIZATION STRATEGIES

- Two principal devices have been developed that support programming for concurrency:
  - semaphores and monitors.

P(s) - Try to enter
When a thread (person) wants to use the resource, it calls P on the semaphore.

Two Key Operations: P and V
In the context of semaphores, we have two main actions to manage access to the shared resource: P (think of it as "wait" or "try to enter") and V (think of it as "signal" or "leave").

SYNCHRONIZATION STRATEGIES: Semaphores

V(s) - Leave
When a thread (person) is done with the resource, it calls V on the semaphore to signal that it’s finished.

If no one is waiting in line, it simply increases the counter by 1 (freeing up space for the next person who wants to use it).
If there are threads waiting (people in line), it allows one of them to enter the resource, instead of just increasing the counter.

- Semaphores were originally defined by Dijkstra [1968a].
- Basically, a semaphore is an integer variable and an associated thread queueing mechanism.
- Two atomic operations, traditionally called *P* and *V*, are defined for a semaphores:
  - *P* (s)- >if *s* > 0 then assign s = s - 1; otherwise block (enqueue) the thread that called *P*.
  - *V* (*s*)- >if a thread *T* is blocked on the semaphore s, then wake up *T;* otherwise assign s = s + 1.
- The operations *P* and *V* are atomic in the sense that they cannot be interrupted once they are initiated.
  - If the semaphore only takes on the values 0 and 1, it is called a *binary* semaphore. Otherwise, it is called a *counting semaphore*.

If the resource is available (meaning the semaphore count is greater than 0), the thread can "enter." It reduces the counter by 1 and starts using the resource.
If the resource isn’t available (the counter is 0), the thread has to wait. It is blocked and put in line until the resource becomes available again.

Semaphores : producer-consumer

- A classic example occurs in the case of producer-consumer cooperation, also called as *cooperative synchronization.*
- Where a **single producer task deposits information** into a shared, single-entry buffer for a **single consumer task to retrieve**.
- The **producer** thread waits (via a P) for the buffer to be empty, deposits information, then signals (via a *V)* that the buffer is full.
- The **consumer** thread waits (via a P) for the buffer to be full, then removes the information from the buffer, and signals (via a *V)* that the buffer is empty.
- semaphore is an elegant, low-level mechanism for synchronization control, difficult to build a large, multitasking system like OS.

Monitors

- *Monitors* [Hoare, 1974] provide an alternative device for managing concurrency and avoiding deadlock.
- Monitors provide the basis for synchronization in Java.
  - Provide an automatic locking mechanism on concurrent operations so that at most one thread can be executing an operation at one time.
  - Its purpose is to encapsulate a shared variable with primitive operations (signal and wait) on that variable

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class Person implements Runnable {
   public static String winner = null;
   private final String name;
   public Person(String name) {
      this.name = name;
   }
   public void race() {
      for (int meters = 0; meters <= 10; meters++) {
         if (winner == null) {
            if (meters == 10) {
               synchronized (Person.class) {
                  if (winner == null) {
                     winner = name; // Use the instance name
                     System.out.println("Winner is " + winner);
                  }
               }
               break;
            } else {
               System.out.println(name + " meters run = " +
meters);          }          }       }    }
   public void run() {
      race();
   }}
```

```java
public class TestRace {
   public static void main(String[] args) {
      ExecutorService executor =
Executors.newFixedThreadPool(4);

      // Create and submit Person instances as tasks
      executor.submit(new Person(" Elon musk "));
      executor.submit(new Person(" Jeff Bezos "));
      executor.submit(new Person(" Zuckerberg "));
      executor.submit(new Person(“Sundar Pichai"));

      executor.shutdown();
   }
}
```

```
Sundar Pichai meters run = 7
Zuckerberg meters run = 6
Sundar Pichai meters run = 8
Sundar Pichai meters run = 9
Jeff Bezos meters run = 9
Elon Musk meters run = 3
Zuckerberg meters run = 7
Winner is Sundar Pichai

Process finished with exit code 0
```

```java
public class RaceCondition{
  public static void main(String[] args) {
    Counter counter = new Counter();
    Thread t1 = new Thread(counter, "Thread-1");
    Thread t2 = new Thread(counter, "Thread-2");
    Thread t3 = new Thread(counter, "Thread-3");
    t1.start();
    t2.start();
    t3.start();
  }
}
```

Output

```
Value  After increment Thread-2  3
Value after decrementing Thread-2  2
Value After increment Thread-1 2
Value after decrementing Thread-1  1
Value After increment Thread-3 1
Value after decrementing Thread-3  0
```

```java
class Counter implements Runnable{
  private int c = 0;
  public void increment() {
    try {
                Thread.sleep(10);
    } catch (InterruptedException e) { e.printStackTrace();    }
   c++;
  }
  public void decrement() {
    c--;   }

  public int getValue() {
    return c;   }

  @Override
  public void run() {
    this.increment();      //incrementing
    System.out.println("Value After increment "
    + Thread.currentThread().getName() + " " + this.getValue());
       this.decrement(); //decrementing
    System.out.println("Value after decrementing "
    + Thread.currentThread().getName() + " " + this.getValue());
  } }
```

```
public class RaceCondition{
  public static void main(String[] args) {
    Counter counter = new Counter();
    Thread t1 = new Thread(counter, "Thread-1");
    Thread t2 = new Thread(counter, "Thread-2");
    Thread t3 = new Thread(counter, "Thread-3");
    t1.start();
    t2.start();
    t3.start();
  }
}
```

Output

Value After increment Thread-2 1
Value after decrementing Thread-2 0
Value After increment Thread-3 1
Value after decrementing Thread-3 0
Value After increment Thread-1 1
Value after decrementing Thread-1 0

```
class Counter implements Runnable{
  private int c = 0;
  public void increment() {
    try {
                Thread.sleep(10);
    } catch (InterruptedException e) { e.printStackTrace();    }
   c++;
  }
  public void decrement() {
    c--;   }

  public int getValue() {
    return c;   }

  @Override
  public void run() {
       synchronized(this){
    this.increment();      //incrementing
    System.out.println("Value After increment "
    + Thread.currentThread().getName() + " " + this.getValue());
        this.decrement(); //decrementing
    System.out.println("Value after decrementing "
    + Thread.currentThread().getName() + " " + this.getValue());
  } } }
```

```java
class BankAccount {
    private int balance = 1000;
     public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() + " withdrew " + amount + ". New balance: " +
balance);
        } else {
            System.out.println(Thread.currentThread().getName() + " cannot withdraw " + amount + ". Insufficient
balance.");
        }
    }
}
 class AccountUser extends Thread {
    private BankAccount account;
    private int withdrawAmount;
     public AccountUser(String name, BankAccount account, int withdrawAmount) {
        super(name);
        this.account = account;
        this.withdrawAmount = withdrawAmount;
    }
```

```java
 public void run() {
        account.withdraw(withdrawAmount);
    }
}

public class SharedResourceExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        AccountUser user1 = new AccountUser("User 1", account, 800);
        AccountUser user2 = new AccountUser("User 2", account, 400);

        user1.start();
        user2.start();
    }
}
```

java.util.concurrent

- Defines the built-in approaches to synchronization and interthread communication.
- These include
  - Synchronizers
  - Executors
  - Concurrent collections
  - The Fork/Join Framework

Synchronizers

- Synchronizers offer high-level ways of synchronizing the interactions between multiple threads.
- The synchronizer classes defined by java.util.concurrent are

Class	Explanation
Semaphore	Implements the classic semaphore.
CountDownLatch	Waits until a specified number of events have occurred.
CyclicBarrier	Enables a group of thread to wait at a predefined execution point.
Exchanger	Exchange data between threads.
Phaser	Synchronized threads that advance through multiple phases of an operation.

Semaphore

- To access the resource, a thread must be granted a permit from the semaphore.
- A semaphore controls access to a shared resource through the use of a counter.
  - If the counter is greater than zero, then access is allowed. If it is zero, then access is denied.

Semaphore

- **To use a semaphore:**
  - Thread that wants access to the shared resource tries to acquire a permit.
    - If the semaphore’s count is greater than zero, then the thread acquires a permit
    - Then the semaphore’s count is decremented.
    - Otherwise, the thread will be blocked until a permit can be acquired.
  - Thread no longer needs access to the shared resource, it releases the permit
    - Then the semaphore’s count is incremented.
    - If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java’s Semaphore class

- Semaphore has the two constructors :

```
Semaphore(int num)
Semaphore(int num, boolean how)
```

- num specifies the initial permit count.
  - i.e., num specifies the number of threads that can access a shared resource at any one time.
- If num is one, then only one thread can access the resource at any one time. By default, waiting threads are granted a permit in an undefined order.
- Setting how to true ensure that waiting threads are granted a permit in the order in which they requested access.

Java’s Semaphore class

- To acquire a permit, call the acquire( ) method, which has these two forms:

  void acquire( ) throws InterruptedException

  void acquire(int num) throws InterruptedException

- The first form acquires one permit.
- The second form acquires num permits.
- If the permit cannot be granted at the time of the call, then the invoking thread suspends until the permit is available.

•It has two main operations:
•It's essentially a counter that can be incremented or decremented.
•acquire() (also called wait() or P()): Decrements the counter
•release() (also called signal() or V()): Increments the counter
•If a thread tries to acquire when the counter is 0, it blocks until another thread releases the semaphore.

Java’s Semaphore class

- To release a permit, call release( ), which has these two forms:

```
void release( )
void release(int num)
```

- The first form releases one permit.
- The second form releases the number of permits specified by num.
- To use a semaphore to control access to a resource, each thread that wants to use that resource must first call acquire( ) before accessing the resource.
- When the thread is done with the resource, it must call release( ).

```java
import java.util.concurrent.Semaphore;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class ParkingLot {
  private static final int PARKING_SPACES = 5;
  private static final int TOTAL_CARS = 10;
  private static final Semaphore semaphore = new Semaphore(PARKING_SPACES, true);
  static class Car implements Runnable {
    private final int carId;
    public Car(int carId) {
      this.carId = carId;
    }
    @Override
    public void run() {
      try {
        parkCar();
        Thread.sleep(10000); // Simulate parking duration
        leaveParkingLot();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
    private void parkCar() throws InterruptedException {
      System.out.println("Car " + carId + " is waiting to enter the parking lot.");
      semaphore.acquire();
      System.out.println("Car " + carId + " has parked. Available spaces: " + semaphore.availablePermits());
    }
    private void leaveParkingLot() {
      semaphore.release();
      System.out.println("Car " + carId + " has left. Available spaces: " + semaphore.availablePermits());
    }  }
```

```java
public static void main(String[] args) throws InterruptedException {
ExecutorService executorService =
Executors.newFixedThreadPool(TOTAL_CARS);

    for (int i = 1; i <= TOTAL_CARS; i++) {
      executorService.execute(new Car(i));
      Thread.sleep(1000); // Simulate car arrival
    }

    executorService.shutdown();
}
}
```

```
Car 1 is waiting to enter the parking lot.
Car 1 has parked. Available spaces: 4
Car 2 is waiting to enter the parking lot.
Car 2 has parked. Available spaces: 3
Car 3 is waiting to enter the parking lot.
Car 3 has parked. Available spaces: 2
Car 4 is waiting to enter the parking lot.
Car 4 has parked. Available spaces: 1
Car 5 is waiting to enter the parking lot.
Car 5 has parked. Available spaces: 0
Car 6 is waiting to enter the parking lot.
Car 7 is waiting to enter the parking lot.
Car 6 has parked. Available spaces: 0
Car 1 has left. Available spaces: 1
Car 8 is waiting to enter the parking lot.
Car 9 is waiting to enter the parking lot.
Car 10 is waiting to enter the parking lot.
Car 6 has left. Available spaces: 1
Car 7 has parked. Available spaces: 0
Car 2 has left. Available spaces: 1
Car 8 has parked. Available spaces: 0
Car 8 has left. Available spaces: 1
Car 9 has parked. Available spaces: 0
Car 4 has left. Available spaces: 1
Car 10 has parked. Available spaces: 0
Car 7 has left. Available spaces: 1
Car 5 has left. Available spaces: 2
Car 3 has left. Available spaces: 3
Car 10 has left. Available spaces: 4
Car 9 has left. Available spaces: 5

Process finished with exit code 0
```

```java
import java.util.concurrent.Semaphore;
class IncThread implements Runnable{
    String name;
    Semaphore sem;
    IncThread(Semaphore sem, String name){
        this.name =name;
        this.sem=sem;      }
    public void run(){
        System.out.println(name+":"+"is starting");
        System.out.println(name +":"+ "waiting for permit");
        try{ //first acquire permit
            sem.acquire();
            System.out.println(name +":"+ "acquired permit");
            //now access shared resource
            for(int i=1; i<=5; i++) {
                Shared.count++;
                System.out.println(name+":"+Shared.count);
                Thread.sleep( time: 10);
            }
            //release permit
        sem.release();
        System.out.println(name+":"+"released permit");
    }
    catch(InterruptedException e1){
        System.out.println(e1);      }
    }
}
```

```java
class DecThread implements Runnable{
    String name;
    Semaphore sem;
    DecThread(Semaphore sem, String name){
            this.name =name;
            this.sem=sem;
            }
    public void run(){
        System.out.println(name+":"+"is starting");
        System.out.println(name+":"+"waiting for permit");
        try{
        sem.acquire();
        System.out.println(name+":"+"acquired permit");
            for(int i=1; i<=5; i++){
                Shared.count--;
                System.out.println(name+":"+Shared.count);
                Thread.sleep( time: 10);    }
        sem.release();
        System.out.println(name+":"+"released permit");
    }
    catch(InterruptedException e){
        System.out.println(e);       }
    } }
//A shared resource
class Shared{
    static int count=1; }
public class Synch {
        public static void main(String args[]){
            Semaphore sem=new Semaphore( permits: 1);
            IncThread inc=new IncThread(sem,  name: "A");
            DecThread dec=new DecThread(sem, name: "B");
            Thread t1=new Thread(inc);Thread t2=new Thread(dec);
            t1.start();t2.start();
        }
}
```

Output

```
B:is starting
A:is starting
A:waiting for permit
B:waiting for permit
A:acquired permit
A:2
A:3
A:4
A:5
A:6
B:acquired permit
A:released permit
B:5
B:4
B:3
B:2
B:1
B:released permit
```

Java’s Semaphore class

- In both IncThread and DecThread, notice the call to sleep( ) within run( ).
- It is used to “prove” that accesses to Shared.count are synchronized by the semaphore.
- In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run.
- However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete.
- Thus, Shared.count is incremented five times by IncThread and decremented five times by DecThread.
- The increments and decrements are not intermixed

```java
// An implementation of a producer and consumer
// that use semaphores to control synchronization.

import java.util.concurrent.Semaphore;

class Q {
  int n;

  // Start with consumer semaphore unavailable.
  static Semaphore semCon = new Semaphore(0);
  static Semaphore semProd = new Semaphore(1);

  void get() {
    try {
      semCon.acquire();
    } catch(InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    System.out.println("Got: " + n);
    semProd.release();
  }

  void put(int n) {
    try {
      semProd.acquire();
    } catch(InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    this.n = n;
    System.out.println("Put: " + n);
    semCon.release();
  }
}
```

```java
class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
  }

  public void run() {
    for(int i=0; i < 20; i++) q.put(i);
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
  }

  public void run() {
    for(int i=0; i < 20; i++) q.get();
  }
}
```

```java
class ProdCon {
  public static void main(String args[]) {
    Q q = new Q();
    new Thread(new Consumer(q), "Consumer").start();
    new Thread(new Producer(q), "Producer").start();
  }
}
```

A portion of the output is shown here:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
.
```

Java’s Semaphore class

- The sequencing of put( ) and get( ) calls is handled by two semaphores: semProd and semCon.
- Before put( ) can produce a value, it must acquire a permit from semProd.
- After it has set the value, it releases semCon.
- Before get( ) can consume a value, it must acquire a permit from semCon.
- After it consumes the value, it releases semProd.
- This “give and take” mechanism ensures that each call to put( ) must be followed by a call to get( ).
- Notice that semCon is initialized with no available permits. This ensures that put( ) executes first.
- The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

CountDownLatch

- Sometimes a thread need to wait until one or more events have occurred.
- A CountDownLatch:
  - A CountDownLatch is initially created with a count of the number of events that must occur before the latch is released.
  - Each time an event happens, the count is decremented.
  - When the count reaches zero, the latch opens.

- CountDownLatch has the following constructor:

  CountDownLatch(int num)

  - Here, num specifies the number of events that must occur in order for the latch to open.
- To wait on the latch, a thread calls await( ), which has the forms shown here:

  void await( ) throws InterruptedException
  boolean await(long wait, TimeUnit tu) throws InterruptedException

  - The first form waits until the count associated with the invoking CountDownLatch reaches zero.
  - The second form waits only for the period of time specified by wait.
  - It returns false if the time limit is reached and true if the countdown reaches zero.
- To signal an event, call the countDown( ) method, shown next:

  void countDown( )

  - Each call to countDown( ) decrements the count associated with the invoking object.

The following program demonstrates CountDownLatch. It creates a latch that requires five events to occur before it opens.

```
// An example of CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
  public static void main(String args[]) {
    CountDownLatch cdl = new CountDownLatch(5);

    System.out.println("Starting");

    new Thread(new MyThread(cdl)).start();

    try {
      cdl.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    System.out.println("Done");
  }
}
```

```
class MyThread implements Runnable {
  CountDownLatch latch;

  MyThread(CountDownLatch c) {
    latch = c;
  }

  public void run() {
    for(int i = 0; i<5; i++) {
      System.out.println(i);
      latch.countDown(); // decrement count
    }
  }
}
```

The output produced by the program is shown here:

```
Starting
0
1
2

3
4
Done
```

- **Inside main( ), a CountDownLatch called cdl is created with an initial count of five.**
- **Next, an instance of MyThread is created, which begins execution of a new thread.**
- **cdl is passed as a parameter to MyThread's constructor and stored in the latch instance variable.**
- **Then, the main thread calls await( ) on cdl, which causes execution of the main thread to pause until cdl's count has been decremented five times.**
- **Inside the run( ) method of MyThread, a loop is created that iterates five times.**
- **With each iteration, the countDown( ) method is called on latch, which refers to cdl in main( ).**
- **After the fifth iteration, the latch opens, which allows the main thread to resume.**