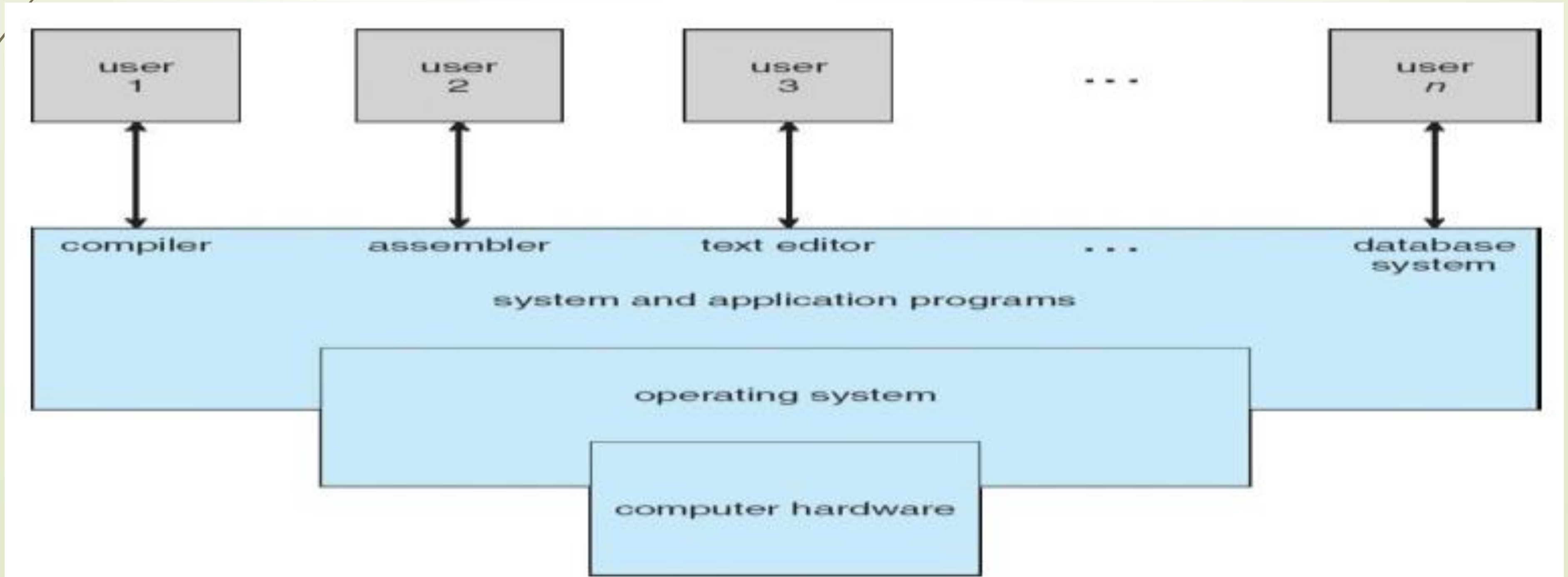
Operating System 2(Labs)

Introduction

- **Operating System**: A program that acts as an intermediary between a user of a computer and the computer hardware.
- Operating system goals:
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Four Components of a Computer System

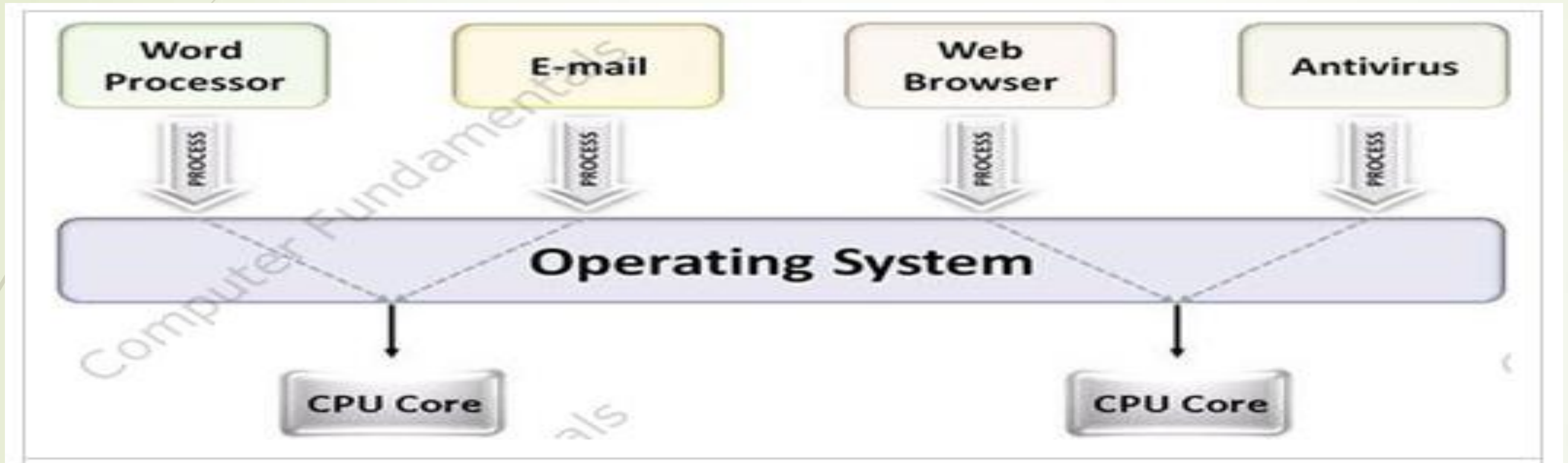


Basic Terms

- Peripherals: is an auxiliary device used to put information into and get information out of a computer.
- Program : is a set of an executable instructions.
- Process: is a program in execution.
- Task : a usually assigned piece of work often to be finished within a certain time(may mean a process, a thread).
- Thread : A thread is a path of execution within a process. A process can contain multiple threads.
- Critical section: is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.
- Job : is a unit of work that has been submitted by user(often means a set of processes).

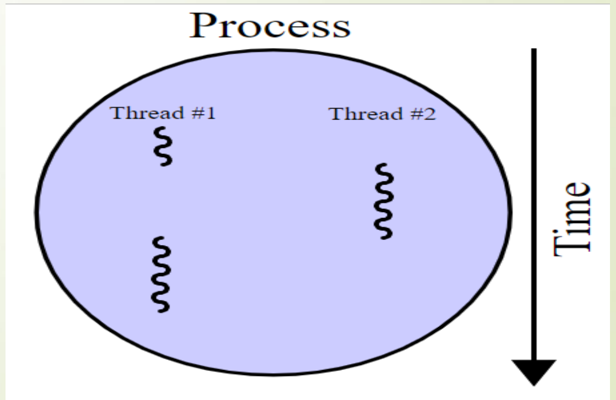
Multitasking and Multithreading

- Multitasking :is a term frequently used to describe the activity of performing multiple tasks (Process)during a specified time period.



Multitasking of an Operating System

- Multithreading :is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system.



Synchronization of process:

 On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

Seminars:

- Real Time Operating System
- Fedora Operating System
- Batch Operating System
- Bada OS
- BlackBerry OS
- iPhone OS / iOS
- Symbian OS
- Windows Mobile OS
- Harmony OS
- Palm OS
- WebOS (Palm/HP)

Project1

Semaphore Implementation

- In computer science, a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system
- Semaphore is an entity devised by Edsger W. Dijkstra, to solve Process Synchronization problem in OS.
- It uses signaling mechanism to allow access to shared resource, Semaphore can have two different operations which are wait and signal. In some books wait signals are denoted by P(s) and signal by V(s).
- Wait p(s) or wait(s)

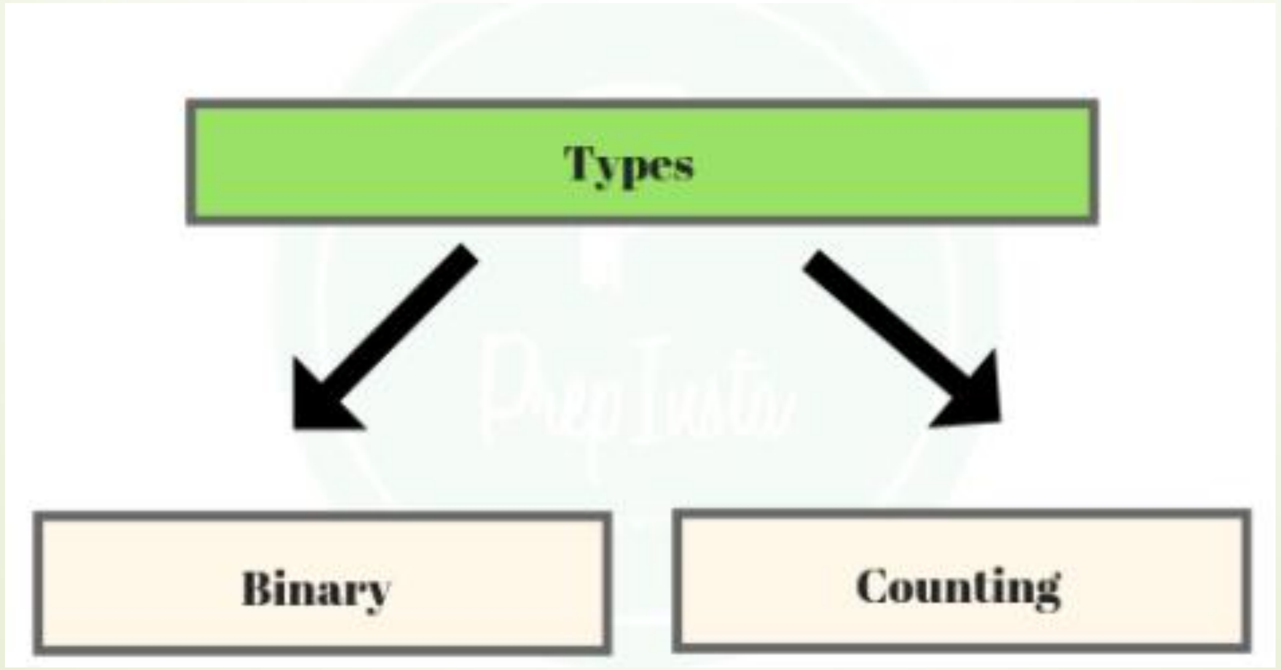
  Wait decrements the value of semaphore by 1
- Signal v(s) or signal(s)

  Signal increments the value of semaphore by 1

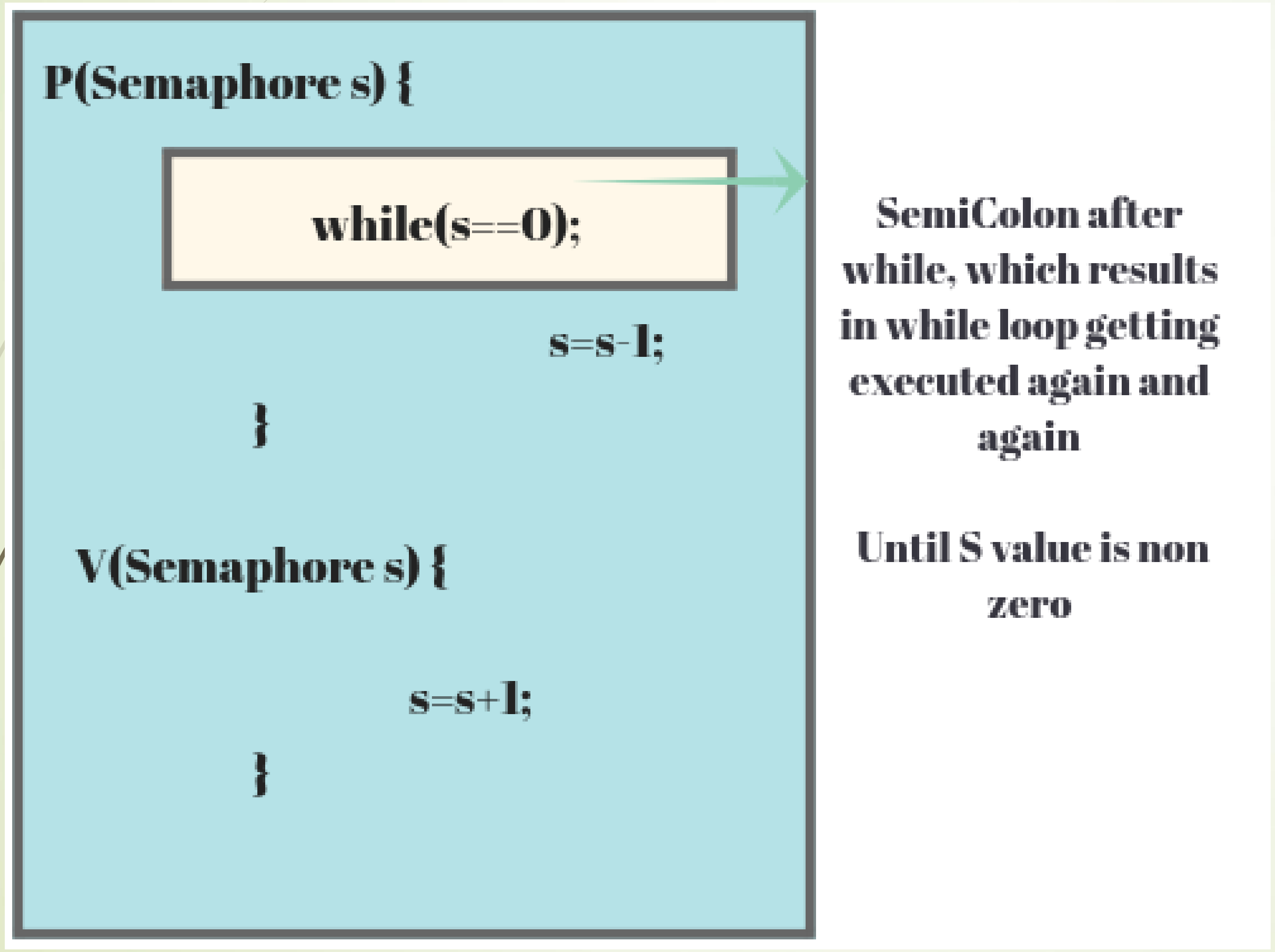
Semaphore Types

- There are two types of Semaphores –

1. Binary Semaphore – Only True/False or 0/1 values
2. Counting Semaphore – Non-negative value



Semaphore in OS



```
P(Semaphore s) {
        while(s==0);
                s=s-1;
     }

 V(Semaphore s) {

            s=s+1;
     }
```

SemiColon after while, which results in while loop getting executed again and again

Until S value is non zero

Binary Semaphore: Binary semaphore is used when there is only one shared resource. Binary semaphore exists in two states ie. Acquired(Take), Released(Give).

Counting Semaphore: To handle more than one shared resource of the same type, counting semaphore is used. Counting semaphore will be initialized with the count(N) and it will allocate the resource as long as count becomes zero after which the requesting task will enter blocked state.

Example For Binary Semaphore

- Let us try to understand the above code with an example –

1. Imagine that there are two processes A and B.
2. At the beginning the value of semaphore is initialized as 1.
3. Imagine process A wants to enter the critical section
    1. Before it can do that it checks the value of semaphore which is 1 thus, it can enter the CS and semaphore value is turned to 0
4. Now imagine that process B wants to enter too
    1. It checks the semaphore value which is 0 thus it can't enter and waits until the value is non zero – non negative value
5. Now, Process A finishes and signals semaphore which in turns changes semaphore value to 1
6. Thus, now process B can enter Critical section

Example For Counting Semaphore

- For Counting Semaphore we initialize the value of semaphore as the number of concurrent access of critical sections we want to allow.
- For example Let us assume that the value of semaphore is 3.
- Process 1 enters Critical section and semaphore value is changed to 2
- Process 2 also enters critical section and semaphore value is changed to 1
- Process 2 signals semaphore and comes out of critical section and Semaphore value is 2
- Note at this moment only 1 process that is process 1 is in critical section
- Process 3 and 4 also enter critical section simultaneously and semaphore value is 0
- At this moment there are three processes in Critical section which are process 1, 3, 4
- Now imagine that process 5 wants to enter the CS. It would not be able to enter as semaphore value is 0
- It can only enter once any of the process 1, 3, 4 signals out of the critical section.

Advantages of semaphore

- In semaphores there is no spinning, hence no waste of resources due to no busy waiting. That is because threads intending to access the critical section are queued
- Semaphores permit more than one thread to access the critical section, in contrast to alternative solution of synchronization like monitors, Hence, semaphores allow flexible resource management.
- the semaphore will prevent the possibility of "deadlock" or situation in which (2) pieces of code can think that they can have access to the resource.

Semaphore in C#

- When a thread enters into a critical section, it decreases the Int32(Semaphore) variable with 1 and when a thread exits from a critical section, it increases the Int32 variable with 1.
- When the Int32 variable is 0, no thread can enters into a critical section.
- Below is the syntax of C# semaphore initialization:

```
Semaphore semaphoreObject = new Semaphore(initialCount: 0, maximumCount: 5);
```

Maximumcount defines how many maximum threads can enter into a critical section. InitialCount set the value of Int32 variable.

For example if we set the maximum count of 3 and initial count of 0. That means 3 threads are already in the critical section. If we set the maximum count of 3 and initial count of 3, that means maximum 3 threads can enter into a critical section and there is no threads currently in the critical section.

WaitOne Method

- Threads can enter into the critical section by using WaitOne method. They called the WaitOne method on semaphore object.
- Below is the syntax of calling WaitOne method.

```
semaphoreObject.WaitOne();
```

Release Method

- When a thread exits from the critical section, it must call the Release method to increment the counter maintained by semaphore object. It allows waiting threads to enter into a critical section.

```
semaphoreObject.Release();
```

By default Release method only increment the counter by 1. That means only one thread exits from the critical section.

Semaphore example

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using System.Text;


namespace sem1
{
    class Program
    {
        static void Main(string[] args)
        {
            Semaphore ss = new Semaphore(0, 1);
            Task t1 = Task.Run(() =>
            {
                while (true)
                {
                    Console.WriteLine("task1 will get the semaphore and pend until it got released");
                    ss.WaitOne();
                }
            });
            Task t2 = Task.Run(() =>
            {
                while (true)
                {
                    Thread.Sleep(4000);
                    Console.WriteLine("task 2 will releasethe semaphore in 4000 mil sec");
                    ss.Release();
                }
            });
            Console.Read();
        }
    }
}
```