Operating Systems
BCSC 0004

Processes Synchronization

BCSC0004: OPERATING SYSTEMS

Objective: *This course aims to introducing the concept of computer organization. In particular, it focuses on basic hardware architectural issues that affect the nature and performance of software.*

Credits:03 **L-T-P-J:3-0-0-0**

Module No.	Content	Teaching Hours
I	**Introduction:** Operating System and its Classification - Batch, Interactive, Multiprogramming, Time sharing, Real Time System, Multiprocessor Systems, Multithreaded Systems, System Protection, System Calls, Reentrant Kernels, Operating System Structure- Layered structure, Monolithic and Microkernel Systems, Operating System Components, Operating System Functions and Services. **Processes:** Process Concept, Process States, Process State Transition Diagram, Process Control Block (PCB), Process Scheduling Concepts, Threads and their management. **CPU Scheduling:** Scheduling Concepts, Performance Criteria, Scheduling Algorithms, Multiprocessor Scheduling. **Process Synchronization:** Principle of Concurrency, Implementation of concurrency through fork/join and parbegin/parend, Inter Process Communication models and Schemes, Producer / Consumer Problem, Critical Section Problem, Dekker's solution, Peterson's solution, Semaphores, Synchronization Hardware. **Classical Problem in Concurrency:** Dining Philosopher Problem, Readers Writers Problem.	20
II	**Deadlock:** System model, Deadlock characterization, Prevention, Avoidance and detection, Recovery from deadlock, Combined Approach. **Memory Management:** Multiprogramming with fixed partitions, Multiprogramming with variable partitions, Paging, Segmentation, Paged segmentation. **Virtual memory concepts:** Demand paging, Performance of demand paging, Page replacement algorithms, Thrashing, Locality of reference. **I/O Management and Disk Scheduling:** I/O devices, I/O subsystems, I/O buffering, Disk storage and disk scheduling. **File System:** File concept, File organization and access mechanism, File directories, File allocation methods, Free space management.	20

Text Books:
- Silberschatz, Galvin and Gagne, "Operating Systems Concepts",9th Edition, Wiley, 2012.

Reference Books:
- SibsankarHalder and Alex a Aravind," Operating Systems", 6th Edition, Pearson Education, 2009.
- Harvey M Dietel, "An Introduction to Operating System", 2nd Edition, Pearson Education, 2002.
- D M Dhamdhere, "Operating Systems: A Concept Based Approach", 2nd Edition, 2006.
- M. J. Bach, "Design of the Unix Operating System", PHI, 1986.

Outcome: After completion of course, the student will be able to:
- CO1: Understand the classification of operating system environment.
- CO2: Understand the basic of process management.
- CO3: Apply the concept of CPU process scheduling for the given scenarios.
- CO4: Illustrate theprocess synchronization and concurrency process in operating system.
- CO5: Analyze the occurrence of deadlock in operating system.
- CO6: Describe and analyze the memory management and its allocation policies.
- CO7: Understand the concepts of disk scheduling.

Process Synchronization

- **Two types Process**
  - **Independent process**: Execution of the process does not affect the execution of other.
  - **Cooperative process:** Execution of one process affects the execution of other process.
- **Process Synchronization:** It helps to maintain shared data consistency & cooperating process execution. When process execute, it should be ensure that, those concurrent access to shared data does not create inconsistency.

Race condition

- In an O.S., the processes that are working together shared some common storage (main memory file) that each process can read and write, when two or more processes are reading or writing some **shared data** and the final result depending on **who run precisely when**, are called race conditions.
- Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the **particular order** in which the access takes place, is **called a race condition.**
- Race condition are also possible in O.S. if the ready queue is implemented as a linked list and if the ready queue is manipulated during the handling of other interrupt before the first one completes. If the interrupts are not disable then the linked list could become corrupted.

How to avoid race condition?

- The key to preempting trouble involving shared storage is to find some way to prohibit more than one process from reading and writing shared data simultaneously.
- That part of the program where the shared memory is accessed is called the critical section.
- To avoid race conditions and flowed results one must identify codes in critical section in each thread.

Critical section:

When more than one processes access a same code segment that segment is known as critical section. Critical section contains shared variables or resources which are needed to be synchronized to maintain consistency of data variable.

The portion in any program that accesses a shared resource is called as critical section (or) critical region.

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (1);
```

Figure General structure of a typical process P_i.

A solution to the critical section problem must satisfy the following 3 requirements:

1.mutual exclusion:

Only one process can execute their critical section at any time.

2. Progress:

When no process is executing a critical section for a data, one of the processes wishing to enter a critical section for data will be granted entry.

3. Bounded wait:

No process should wait for a resource for infinite amount of time.

Peterson’s solution:

Peterson solution is one of the solutions to critical section problem involving two processes. This solution states that when one process is executing its critical section then the other process executes the rest of the code and vice versa.

Peterson solution requires two shared data items:

1) **turn**: indicates whose turn it is to enter into the critical section. If turn == i ,then process i is allowed into their critical section.

2) **flag:** indicates when a process wants to enter into critical section. when process i wants to enter their critical section, it sets flag[i] to true.

```
do {
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
critical section
flag[i] = FALSE; remainder section
} while (TRUE);
```

Peterson’s solution:

Process P0	Process P1
While(1)	While(1)
{	{
flag [0] = T	flag [1] = T
turn = 1	turn = 0
While(turn ==1 && flag[1]==T);	While(turn ==0 && flag[0]==T);
Critical section	Critical section
flag[0] = F	flag[1] = F
}	}

Semaphores

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows –

- **Wait** The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.
- **Signal** The

Process P_i	Wait (S)	Signal (S)

do
{
Entry Section
Critical Section
Exit Section
Remainder Section
} while(T);
	Wait(S)
	{
	While(s<=0);
	S = S-1;
	}
		Signal (S)
		{
		S = S+1
		}

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

- **Counting Semaphores** These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphores** The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

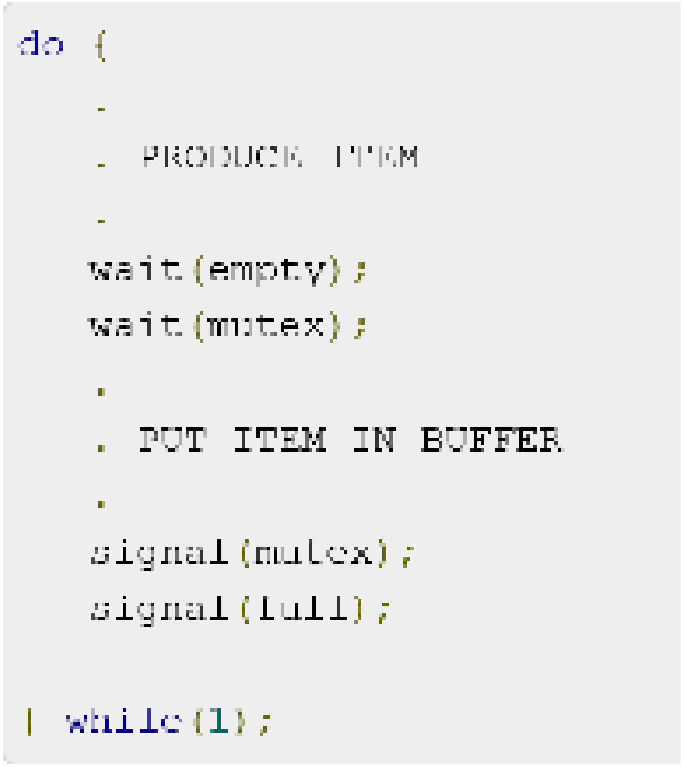
- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Producer Consumer Problem

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them. A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

Producer Process

The code that defines the producer process is given below –

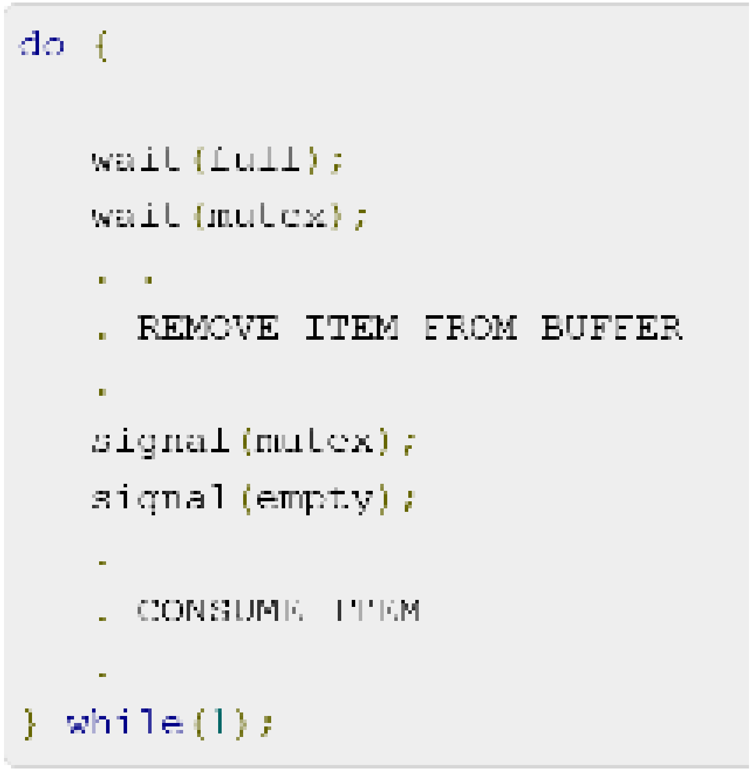


```
do {
    .
    . PRODUCE ITEM
    .
   wait(empty);
   wait(mutex);
    .
    . PUT ITEM IN BUFFER
    .
   signal(mutex);
   signal(full);

} while(1);
```

Consumer Process

The code that defines the consumer process is given below:



```
do {

   wait(full);
   wait(mutex);
    . .
    . REMOVE ITEM FROM BUFFER
    .
   signal(mutex);
   signal(empty);
    .
    . CONSUME ITEM
    .
} while(1);
```

Monitor

Monitor is one of the ways to achieve Process synchronization. Monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can’t access the internal variable of monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

Components of Monitor

There are four main components of the monitor:

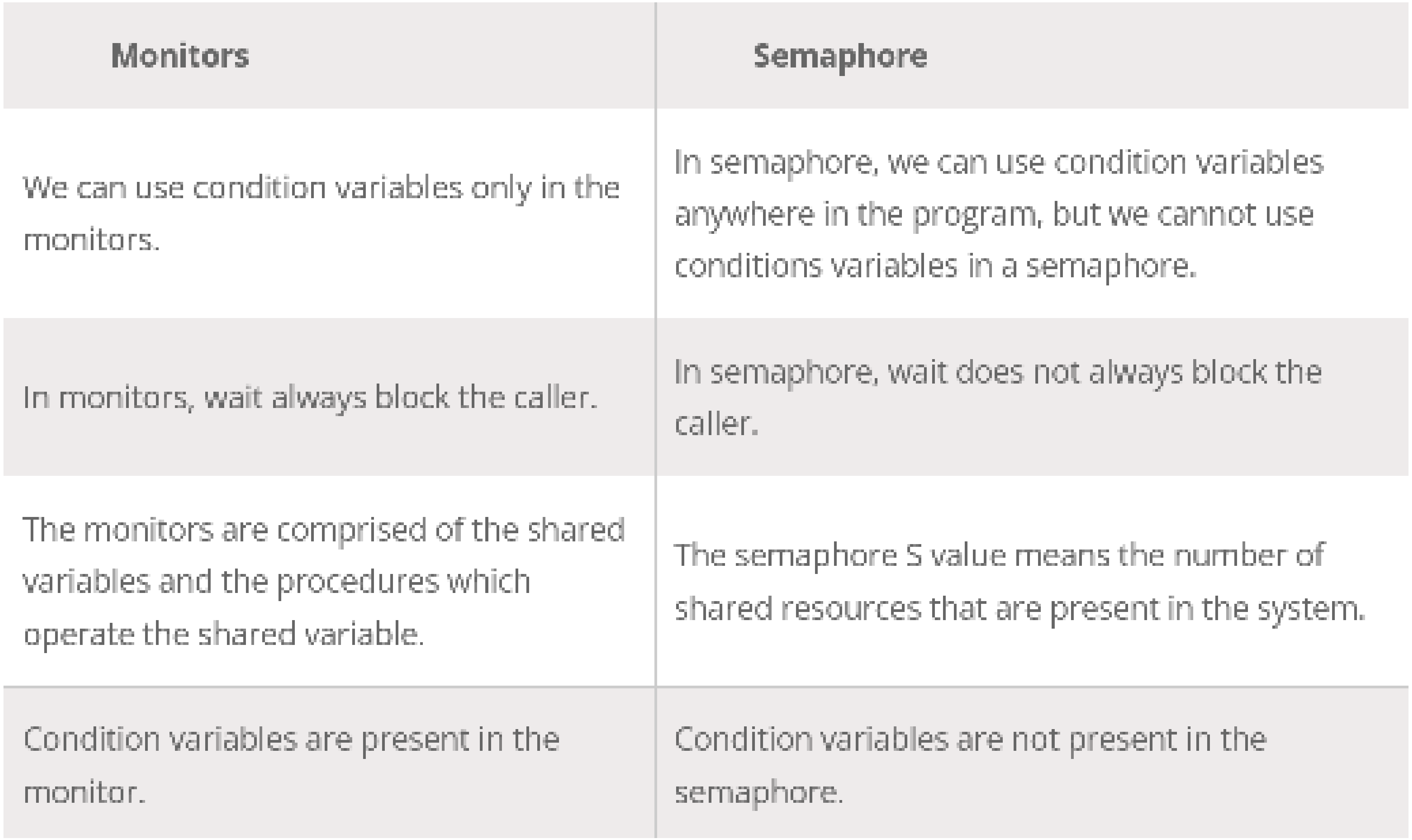
Initialization: – Initialization comprises the code, and when the monitors are created, we use this code exactly once.

Private Data: – Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.

Monitor Procedure: – Monitors Procedures are those procedures that can be called from outside the monitor.

Monitor Entry Queue: – Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.

Difference between Monitors and Semaphore



Monitors	Semaphore
We can use condition variables only in the monitors.	In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore.
In monitors, wait always block the caller.	In semaphore, wait does not always block the caller.
The monitors are comprised of the shared variables and the procedures which operate the shared variable.	The semaphore S value means the number of shared resources that are present in the system.
Condition variables are present in the monitor.	Condition variables are not present in the semaphore.