UNIT III

CHAPTER 3

Process Coordination

Syllabus

Synchronization : Principles of Concurrency, Requirements for Mutual Exclusion, Mutual Exclusion: Hardware Support, Operating System Support (Semaphores and Mutex), Programming Language Support (Monitors).

Classical synchronization problems : Readers/Writers Problem, Producer and Consumer problem, Inter-process communication (Pipes, shared memory: system V)

Deadlock : Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery from Deadlock.

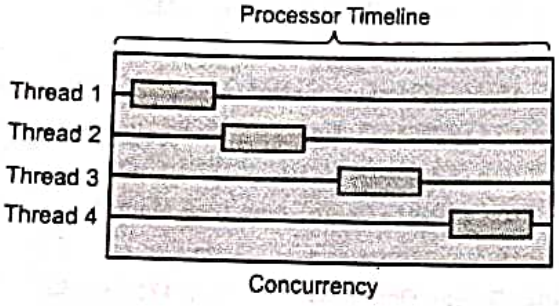
3.1 CONCURRENCY AND PARALLELISM

3.1.1 Concurrency

(1) Concurrency is an execution of the multiple instruction sequences at the same time. Concurrency also means that multiple processes or threads are making progress concurrently.

(2) It happens in the operating system when there are several process threads running in parallel. The running process threads always communicate with each other through **shared memory or message passing.**

(3) In concurrency, one thread is executed at a time by the CPU, these threads can be switched in and out as required. This means that no thread is actually completed totally before another one is scheduled. So all the threads are executing concurrently.



(1C1)**Fig. 3.1.1 : Concurrency**

(4) Concurrency results in sharing of resources result in problems like deadlocks and resources starvation. It helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput.

(5) The terms concurrency and parallelism are used in context of multithreaded programs. However, they are quite different.

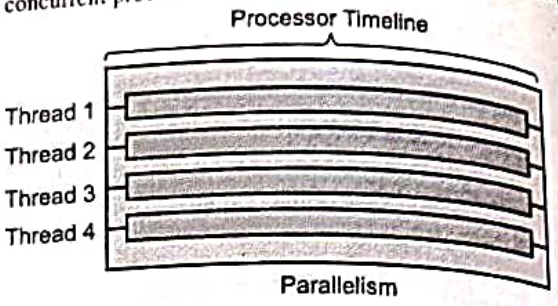
3.1.2 Parallelism

Parallelism means that multiple processes or threads are making progress in parallel. This means that the threads are executing at the same time. This can happen if all the threads are scheduled on parallel processors.

Concurrency arises in three different contexts:

(1) **Multiple applications :** Multiprogramming was invented to allow processing time to be **dynamically** shared among a number of active applications.

(2) **Structured applications :** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.



(1C2)**Fig. 3.1.2 : Parallelism**

(3) **Operating system structure :** The same structuring advantages apply to system programs, and we have seen that the operating systems are themselves often implemented as a set of processes or threads

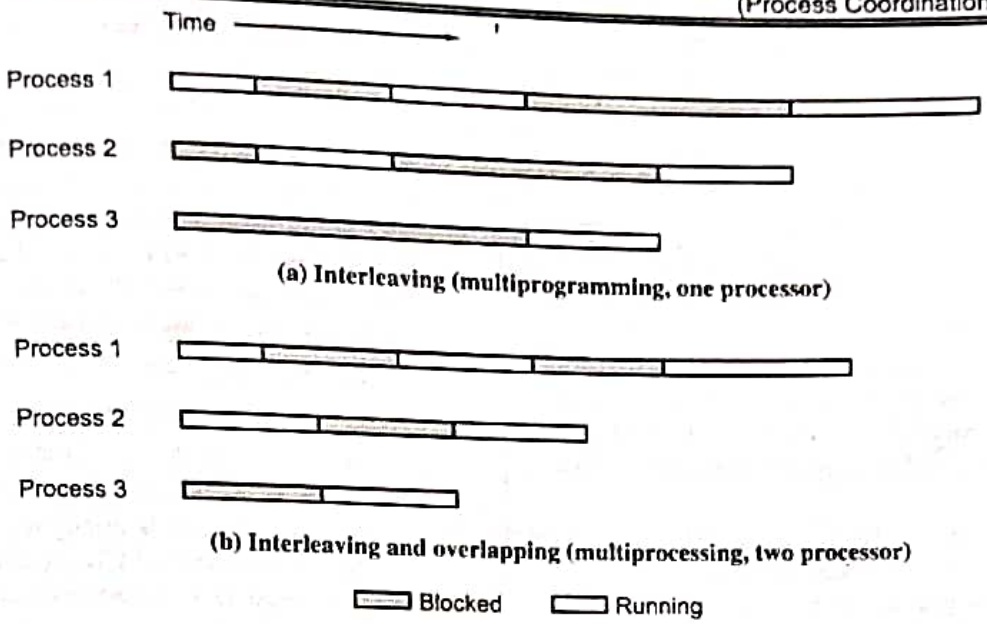
3.1.3 Principle and Issues in Parallelism Problem

(1) In a single-processor multiprogramming system, processes are **interleaved in time** to get the appearance of simultaneous execution (Fig. 3.1.3(a)).

(2) In **Single processor multiprogramming system** parallel processing is not achieved and there are certain overheads involved in switching between the processes, the interleaved execution has main advantage in processing efficiency and interleaved execution.

(3) But in **multiple-processor system**, it is possible not only to interleave the execution of multiple processes but also to overlap them (Fig. 3.1.3(b)).

NOTES



(a) Interleaving (multiprogramming, one processor)

(b) Interleaving and overlapping (multiprocessing, two processor)

(IC3)Fig. 3.1.3 : Multiprocessing

3.1.4 Issues in Uniprocessor and Multiprocessor System

- Interleaving and overlapping represent fundamentally different modes of execution and present different problems. But both the techniques can be seen as examples of concurrent processing, and both present the same problems.
- In the case of a uniprocessor, the problems start from a basic characteristic of multiprogramming systems: The relative speed of execution of processes cannot be predicted. It depends on the activities of other processes, the way in which the OS handles interrupts, and the scheduling policies of the OS. The following difficulties arise:
  - The sharing of global resources. For example, if two processes both make use of the same global variable and both perform reads and writes on that variable, then the order in which the various reads and writes are executed is critical.
  - It is difficult for the OS to manage the allocation of resources optimally. For example, process A may request use of, and be granted control of, a particular I/O channel and then be suspended before using that channel. It may be undesirable for the OS simply to lock the channel and prevent its use by other processes; indeed this may lead to a deadlock condition.
  - It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible.
- All of the foregoing difficulties present themselves in a multiprocessor system as well, because here too the relative speed of execution of processes is unpredictable.
- A multiprocessor system must also deal with problems arising from the simultaneous execution of multiple processes. Fundamentally, however, the problems are the same as those for uniprocessor systems.

3.1.5 Example for Demonstration

Consider the following procedure:

```
void display()
{
    inpchar = getchar();
    writechar = inpchar;
    putchar(writechar);
}
```

In above procedure display()

1. Input is read from char, character at time
2. Read char is stored in inpchar variable
3. inpchar data is stored in writechar variable
4. write variable data is send to display.

Any program can call this procedure repeatedly to accept user input and display it on the user's screen.

(a) Single Processor and Single user

- Let's consider single-processor multiprogramming system supporting a single user. The user switch between application and every application uses the same keyboard for input and the same screen for output. Because each application needs to use the procedure display, this procedure is shared procedure that is loaded into a portion of memory common to all applications.
- Thus, only a single copy of the display procedure is used, saving space. The sharing of main memory among processes is useful to permit efficient and close interaction among processes. But this sharing of procedure can lead to problems.

(b) Let consider some Sequence of Execution

1. Process P1 calls the display procedure and is interrupted immediately after getchar returns its value and stores it in inpchar. Now the most recently entered character, a, is stored in variable inpchar.

2. Process P2 is activated and invokes the display procedure, which complete it processing, inputting and then displaying a single character, b, on the screen.

Explanation

When Process P1 is resumed. By this time, the value has been overwritten in inpchar and therefore lost. Instead, inpchar contains b, which is transferred to writechar and displayed. Instead of display a, b, output b twice. The main cause of this problem is the shared global variable, inpchar.

Since this variable is accessed by multiple character and if any one process updates the global variable and then is interrupted, another process may change the variable before the first process can use its value. Now if only one process can call procedure at time what will happen let see

- Process P1 calls the display procedure and is interrupted immediately after the completion of the input function. At this point, the most recently entered character, a, is stored in variable inpchar.
- Process P2 is starts can call the display procedure. However, because P1 yet have not completed the display process, so P2 process will in blocked from entering the procedure. Therefore, P2 is suspended state and awaiting for availability of display procedure.
- After some time, process P1 is resumed and completes execution of display. The proper character, a, is displayed.
- When P1 exits display this removes the block on P2. When P2 is resumed later, the display procedure is successfully resumed.

From this example it is clear that it is necessary to protect shared global variables (and other shared global resources) and it is only way to control the code that accesses the variable. If criteria of one process is set then at a time may enter display and it should complete itself before it is made available for another process, the above discussed problem will not occur.

This problem was discussed with the assumption that there was a single-processor, multiprogramming OS. The example states the problems of concurrency occur even when there is a single processor. In a multiprocessor system, the same problems of protected shared resources arise, and the same solution works.

Process P1	Process P2
inpchar = getchar();	
	inpchar= getchar();
writechar= inpchar;	writechar= inpchar;
putchar(writechar);	
	putchar(writechar);

Fig. 3.1.4

Consider there is no mechanism for controlling access to the shared global variable :

- Processes P1 and P2 are both executing, each on a separate processor. Both processes calls the display procedure.
- The following events occurs on the same line in parallel: The result is that the character input to P1 is lost before being displayed, and the character input to P2 is displayed by both P1 and P2. Again, Now let's see the operation in case one process at a time is considered in display procedure.

Sequence occurs as follows :

- Processes P1 and P2 are both executing, each on a separate processor. P1 calls the display procedure.
- While P1 is inside the display procedure, P2 call display. Because P1 is still inside the display procedure (whether P1 is suspended or executing), P2 is blocked from entering the procedure. Therefore, P2 is suspended and waiting for display procedure.
- After some time, process P1 completes execution of display and exit that procedure, and continues executing. As soon as P1 exit from display procedure, P2 is resumed and begins executing display.

☑ Solution to the Problem

In the case of a uniprocessor system, the problem is that an interrupt can stop instruction execution anywhere in a process. In the case of a multiprocessor system, the same condition and, also a problem can be caused because two processes may be executing simultaneously and both trying to access the same global variable. But the solution to both types of problem is the same that is control access to the shared resource.

3.1.6 Advantages and Disadvantages in Concurrency

☞ Advantages

1. **Running of multiple applications :** It enables to run multiple applications at the same time.
2. **Better resource utilization :** It enables that the resources that are unused by one application can be used for other applications.
3. **Better average response time :** Without concurrency, each application has to be run to completion before the next one can be run.
4. **Better performance :** It enables the better performance by the operating system. When one application uses only the processor and another application uses only the disk drive then the time to run both applications concurrently to completion will be shorter than the time to run each application consecutively.

Disadvantages

1. It is required to protect multiple applications from one another.
2. It is required to coordinate multiple applications through additional mechanisms.
3. Additional performance overheads and complexities in operating systems are required for switching among applications.
4. Sometimes running too many applications concurrently leads to severely degraded performance.

3.2 SYNCHRONIZATION

GQ. Write short note on synchronization

Communication of a message between two processes implies some level of synchronization between the two :

- The receiver cannot receive a message until it has been sent by another process.
- We need to specify what happens to a process after it issues a send or receive primitive.
- Consider the send primitive first. When a send primitive is executed in a process, there are two possibilities : Either the sending process is blocked until the message is received, or it is not.

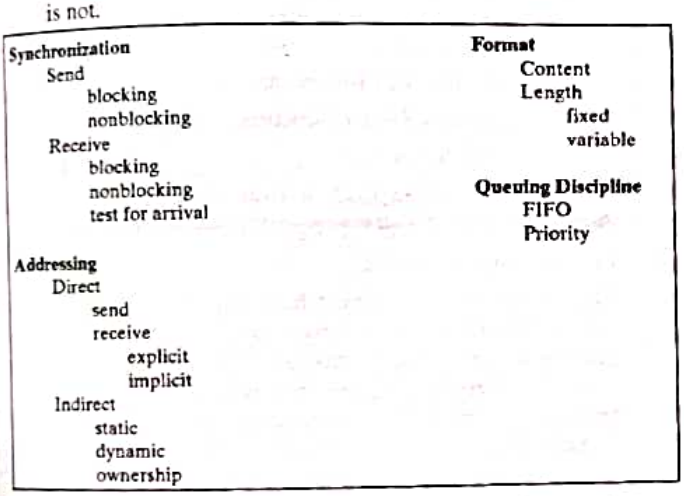


Fig. 3.2.1 : Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Similarly, when a process issues a receive primitive, there are two possibilities :

1. If a message has previously been sent, the message is received and execution continues.
2. If there is no waiting message, then either
   (a) The process is blocked until a message arrives, or
   (b) The process continues to execute, abandoning the attempt to receive.

Thus, both the sender and receiver can be blocking or nonblocking. Three combinations are common, although any particular system will usually have only one or two combinations implemented :

- **Blocking send, blocking receive :** Both the sender and receiver are blocked until the message is delivered, this is sometimes referred to as a *rendezvous*. This combination allows for tight synchronization between processes.
- **Nonblocking send, blocking receive :** Although the sender may continue on, the receiver is blocked until the requested message arrives. This is probably the most useful combination. It allows a process to send one or more messages to a variety of destinations as quickly as possible. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives. An example is a server process that exists to provide a service or resource to other processes.
- **Nonblocking send, nonblocking receive :** Neither party is required to wait.

Non-blocking Send

- The nonblocking send is more natural for many concurrent programming tasks. For example, if it is used to request an output operation, such as printing, it allows the requesting process to issue the request in the form of a message and then carry on.
- One main dis-advantage of the nonblocking send is that an error could lead to a situation in which a process repeatedly generates messages. Because there is no blocking to discipline the process, these messages could consume system resources, including processor time and buffer space, to the detriment of other processes and the OS. Also, the nonblocking send places the burden on the programmer to determine that a message has been received: Processes must employ reply messages to acknowledge receipt of a message.

Blocking Receive

- For the receive primitive, the blocking version appears to be more natural for many concurrent programming tasks.
- Generally, a process that requests a message will need the expected information before proceeding. However, if a message is lost, which can happen in a distributed system, or if a process fails before it sends an anticipated message, a receiving process could be blocked indefinitely.

Non Blocking Receive

- This problem discussed in blocking receive can be solved by the use of the nonblocking receive. However, the danger of this approach is that if a message is sent after a process has already executed a matching receive, the message will be lost.
- Other possible approaches are to allow a process to test whether a message is waiting before issuing a receive and allow a process to specify more than one source in a receive

primitive. The latter approach is useful if a process is waiting for messages from more than one source and can proceed if any of these messages arrive.

3.3 MUTUAL EXCLUSION

GQ. What is mutual exclusion ? Explain its significance.

UQ. Write short note on : Mutual Exclusion.

(SPPU - Dec. 18, Q. 4(b), May 19, 5 Marks)

1. Formally speaking, while one process executes the shared variable, all other processes desiring to do so at the same moment should be kept waiting.
2. When that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously.
3. When oonly one process should be allowed to execute in critical section. ***This is called Mutual Exclusion.***
4. The **mutual exclusion** is put into effect only if processes access shared changeable data. If during execution, the operations of the processes do not conflict with each other, they should be permitted to proceed in parallel.
5. **Significance** of mutual execution :
   - Prevents race around condition.
   - Prevents multiple threads to enter critical section at the same time.

3.3.1 Requirements for Mutual Exclusion

GQ. State requirement of Mutual Exclusion.

UQ. List the requirements of mutual exclusion.

(SPPU - Q. 4(b)OR, Dec. 17, 5 Marks)

(a) To provide support for mutual exclusion should meet the following requirements

- Mutual exclusion must be implemented: Only one process at a time is allowed into its critical section, even though all processes that have critical sections for the same resource or shared object.
- A process that halts in its noncritical section must do so without interfering with other processes.
- It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
- When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
- No assumptions are made about relative process speeds or number of processors.
- A process remains inside its critical section for a finite time only.

(b) Different ways in which requirements for mutual exclusion can be satisfied

- To leave the responsibility with the processes that wish to execute concurrently. Thus processes, whether they are system programs or application programs, would be required to coordinate with one another to enforce mutual exclusion, with no support from the programming language or the OS. This can be called as software approaches.
- But this approach is prone to high processing overhead and bugs, in spite it is useful to examine such approaches to gain a better understanding of the complexity of concurrent processing.
- The use of special-purpose machine instructions. These have the advantage of reducing overhead but nevertheless will be shown to be unattractive as a general-purpose solution.

3.3.2 Hardware Support

For implementation mutual exclusion many software algorithms are developed. But there is high processing overhead and the risk of logical errors in those software algorithms. Several interesting **hardware approaches** to mutual exclusion. Some are discussed here those are:

(a) Interrupt Disabling
(b) Special Machine Instructions
(c) Compare and Swap Instruction
(d) Exchange Instruction
(e) Properties of the Machine-Instruction Approach

(a) Interrupt Disabling

Concurrent processes cannot have overlapped execution but can be interleaved in uniprocessor system. Furthermore, a process will continue to run until OS invokes a service or until it is interrupted. Therefore, to have mutual exclusion, the process should not be interrupted. In OS kernel some primitives can be defined to enable or disable the interrupt. A process can then enforce mutual exclusion in the following way :

```
while (true)
{
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Issues with Interrupt approach

- Mutual exclusion is guaranteed because critical section cannot be interrupted. But the price of this approach is high.

Also the efficiency of execution could be is degraded because the processor is limited in its ability to interleave processes.

- This approach will not work in a multiprocessor architecture. When the computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion.

(b) Special Machine Instructions

- In a multiprocessor configuration, several processors share access to a common main memory. In this case, there is not a master/slave relationship; rather the processors behave independently in a peer relationship. There is no interrupt mechanism between processors on which mutual exclusion can be based.
- At the hardware level, access to a memory location excludes any other access to that same location. With this as a foundation, processor designers have proposed several machine instructions that carry out two actions atomically (The term *atomic* means that the instruction is treated as a single step that cannot be interrupted)such as reading and writing or reading and testing, of a single memory location with one instruction fetch cycle. During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location.

(c) Compare and Swap Instruction

GQ. Describe how swap() instruction can be used to provide mutual exclusion that satisfies he bounded-waiting requirement.

The compare and swap instruction, also called a compare and exchange instruction, can be defined as follows :

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
        return oldval;
}
```

- One verison of the instruction checks a memory location (*word) against a test value (testval). If the memory location's current value is testval, it is replaced with newval; otherwise it is left unchanged. The old memory value is always returned; thus, the memory location has been updated if the returned value is the same as the test value. This atomic instruction therefore has two parts:
  1. A compare is made between a memory value and
  2. A test value; if the values differ a **swap** occurs.
- The entire compare & swap function is carried out atomically; that is, it is not subject to interruption.
- Another version of this instruction returns a Boolean value: true if the swap occurred; false if no swapping. Some version of this instruction is available on nearly all processor families (x86, IA64, sparc, /390, etc.), and most operating systems use this instruction for support of concurrency.

```
/* program mutualexclusion */

const int n = /* number of processes */;
int status;
void P(int i)
{
while (true)
{
 while
(compare_and_swap(status, 0, 1) == 1)
  /* do nothing */;
  /* critical section */;
  status = 0;
  /* remainder */;
 }
}
void main()
{
 status = 0;
  parbegin (P(1), P(2), ...
,P(n));
}
```

(a) Compare and swap instruction

```
/* program mutualexclusion */

int const n = /* number of processes**/;
int status;
void P(int i)
{
int keyi = 1;
while (true)
 {
  do exchange (keyi, status)
    while (keyi != 0);
  /* critical section */;
   status = 0;
  /* remainder */;
 }
}
void main()
{
 status = 0;
  parbegin (P(1), P(2), ...,
P(n));
}
```

(b) Exchange instruction

Fig. 3.3.1 : Hardware Support for Mutual Exclusion

- Fig. 3.3.1(a) shows a mutual exclusion protocol based on the use of this instruction. A shared variable status = 0 is initialized.
- The only process with status = 0 enters its critical section, rest all other processes into a busy waiting mode since they cannot enter their critical section. The term **busy waiting**, or **spin waiting**, is a technique in which a process does nothing but continues to execute an instruction or set of instructions that check the variable so that it can enter critical section or again access.
- As soon as process moves out of its critical section, it makes status = 0; now one of the waiting processes is granted access to its critical section. The choice of process depends on which process happens to execute the compare and swap instruction next.

(d) Exchange Instruction

The exchange instruction can be defined as follows:

```
void exchange (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

- The instruction exchanges the contents of a register with that of a memory location. The Intel Pentium IA-32 architecture and the intel Itanium IA-64 architecture contain an XCHG instruction.
- Fig. 3.3.1(b) shows a mutual exclusion protocol based on the use of an exchange instruction. A shared variable status is initialized to 0. Each process uses a local variable *key* that is initialized to 1. A process with status = 0 can enter the critical section and rest all other process are excluded.
- Process set status = 0 when it leaves its critical section, which allows other process to gain access to its critical section. Note that the following expression always holds because of the way in which the variables are initialized and because of the nature of the exchange algorithm: If status = 0, then no process is in its critical section. If status = 1, then exactly one process is in its critical section, namely the process whose *key* value equals 0.

(e) Properties of the Machine-Instruction Approach

The use of a special machine instruction to enforce mutual exclusion has a number of advantages:

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- It is simple and therefore easy to verify.
- It can be used to support multiple critical sections; each critical section can be defined by its own variable.

Disadvantages

1. **Busy waiting is employed.** Thus, while a process is waiting for access to a critical section, it continues to consume processor time.
2. **Starvation is possible.** When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.
3. **Deadlock is possible.** In a single-processor system. Process P1 executes the special instruction (e.g., compare and swap, exchange) and enters its critical section. P1 is then interrupted to give the processor to P2, which has higher priority. If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism. Thus, it will go into a busy waiting loop. However, P1 will never be dispatched because it is of lower priority than another ready process, P2. Because of the drawbacks of both the software and hardware solutions discussed, we need to look for other mechanisms.

3.4 RACE CONDITION

GQ. Explain race condition with example.

UQ. Explain the following terms Race Condition

(SPPU - Q. 3(b), Dec. 18, Q. 3(b), Dec. 19, 3 Marks)

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Example : Let us consider two simple examples.

- Let us consider two processes, P1 and P2, share the global variable num. At some point in its execution, P1 updates num to the value "10", and P2 updates num to the value 20. Thus, the two tasks are in a race to write variable num. From the above example the process that updates last determines the final value of num.
- Let us consider two process, P3 and P4, that share global variables num1 and num2, with initial values num1 = 100 and num2 = 200. At some point in its execution, P3 executes the assignment num1 = num1 + num2, and also P4 executes the assignment num num2 = num1 + num2. This two processes update different variables.
- However, the final values of the two variables depend on the order in which the two processes execute these two expressions. If P3 executes its expression statement first, then the final values are b = 300 and c = 500. If P4 executes its expression statement first, then the final values are b = 400 and c = 300.

3.4.1 Operating System Concerns

Design and management issues are raised because of concurrency.

1. The OS must be able to keep track of the various processes. This can be with using Process Control Blocks (PCB).
2. The OS must allocate and deallocate various resources for each active process. At times, multiple processes want access to the same resource. These resources include **Processor time, Memory, Files, I/O devices.**
3. The OS must protect the data and physical resources of each process against unintended interference by other processes. This involves techniques that relate to memory, files, and I/O devices.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

5. In order to understand how the issue of speed independence can be addressed, let see the ways in which processes can interact.

3.4.2 Process Interaction

We can classify the process interaction in the ways they interact on the basis of the degree to which they are aware of each other's existence. Table 3.4.1 lists three possible degrees of awareness plus the consequences of each :

1. Processes unaware of each other
2. Processes indirectly aware of each other
3. Processes directly aware of each other

Table 3.4.1 : Process Interaction

Degree of Awareness	Relationship Influence That One	Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others. • Timing of process may be affected.	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data Coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected.	• Deadlock (renewable resource) • Starvation

1. Processes unaware of each other

- Independent processes that are not intended to work together that's why they are unaware of each other. The multiprogramming of multiple independent processes is one of the best example. These can either be batch
- Jobs or interactive sessions or a mixture. Even though processes are not working together, the OS needs to be concerned about competition for resources. For example, two independent applications may both want to access the same disk or file or printer. The OS must regulate these accesses.

2. Processes indirectly aware of each other

These are processes that are not necessarily aware of each other by their respective process IDs but that share access to some object, such as an I/O buffer. Such processes exhibit **cooperation** in sharing the common object.

3. Processes directly aware of each other

- These are processes that are able to communicate with each other by process ID and that are designed to work jointly on some activity. Again, such processes exhibit cooperation.
- Situation and condition will not always as suggested in Table 3.4.1. But, several processes may exhibit aspects of both competition and cooperation. Nevertheless, it is productive to examine each of the three items in the preceding list separately and determine their implications for the OS.

3.5 CRITICAL SECTION AND PROBLEMS

UQ. Explain Critical Section problem. What are the requirements should be a solution to critical section problem satisfy ?

SPPU - Q. 3(a), May 16, Q. 3(b), Dec. 18, Q. 4(a), May 19, Q. 3(b), Dec. 19, 5 Marks

- The portion of the program where the shared memory is accessed is referred as the Critical Section. In order to avoid race conditions and faulty results, one must be able to recognize the codes in Critical Sections in each thread.

- **The typical properties of the code that comprises** Critical Section are as follows :
  - These codes reference one or more variables in a "read-update-write" way. At the same time, any of those variables maybe changed by another thread.
  - These codes modify one or more variables that can be referenced in "read-update-write" fashion by another thread.
  - Any part of data structure used by the codes can be modified by another thread.
  - Codes modify any part of a data structure that is currently in use by another thread.
- When one process is currently executing shared modifiable data in its critical section, no other process is to be permitted to execute in its critical section. Hence, the execution of critical sections by the processes is mutually exclusive in time.
- Critical section is a code where only one process at a time can be executing. Critical section problem is design an algorithm that allows at most one process into the critical section at a time, without deadlock. Solution of the critical section problem must satisfy mutual exclusion, progress, bounded waiting.
- Any process directly cannot enter in critical section. First process has to obtain permission for its entry in its critical section. The segment of code which implements this appeal of process is the entry section. When process comes out of the critical section after completing its execution there, it has to execute exit section. The rest of the code is the remainder section.

Any solution to the critical-section problem must satisfy the following three necessary conditions :

1. Mutual exclusion
2. Progress
3. Bounded waiting

- **1. Mutual exclusion :** If one process is executing in its critical section, then other processes should not be executing in their critical sections.
- **2. Progress :** If any process is not executing in its critical section and some processes desire to enter their critical sections, then only those processes that are not executing in their remainder sections should take part in the decision on which will enter its critical section next, and this selection cannot be delayed indefinitely.
- **3. Bounded waiting :** There should be bound, on the number of times that other processes are permitted to enter their critical sections after a process has made a request to enter its critical section and before that request is approved.

Semaphore and monitor are the solutions to achieve mutual exclusion. Semaphore is a synchronization variable that tasks on positive integer values. Binary are those that only have two values 0 or 1. Hardware does not provide the semaphore. The critical section problem can be solved by using semaphores. Like semaphore, a monitor also solves critical section problem. It is a software component which contains one or more procedures, an initialization sequence and local data.

Following are the components of monitors :

1. Shared data declaration
2. Shared data initialization
3. Operations on shared data
4. Synchronization statement

3.5.1 Competition among Processes for Resources

- Conflict in concurrent process starts when they are competing for the same resource. Take example of two or more processes need to access a resource when they are executed. Also these process are unaware of the presence of other processes, and they are unaffected by the execution of the other processes.
- It follows from this that each process should leave the state of any resource that it uses unaffected. Examples of resources include I/O devices, memory, processor time, and the clock. There is no exchange of information between the competing processes. However, the execution of one process may affect the behavior of competing processes.
- In particular, if two processes both need the access of a single resource, then one process will be allocated that resource by the OS, and the other has to wait. Result of these is the process is slowed down since access is denied. If the blocked process may never get access to the resource and hence it process will be in waiting and never terminate.

3.5.1(A) Problems faced by Competing Process

1. Mutual exclusion
2. Deadlock
3. Starvation

1. Mutual Exclusion

- Let say two or more processes need access to a single non sharable resource, such as a printer or plotter. Now each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. This type of resource can also be referred as a **critical resource**, and the portion of the program that uses it a **critical section** of the program.
- Only one program at a time be allowed in its critical section. For such scenario we cannot simply rely on the OS to understand and implement this restriction because the detailed requirements may not be obvious. In the case of the

printer or plotter, for example, we want any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.

2. Deadlock

- Let's consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation:
  - The OS assigns R1 to P2, and R2 to P1.
  - Each process is waiting for one of the two resources.
- Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.

3. Starvation

- Let's say that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 possess the resource, and both P2 and P3 are delayed, waiting for that resource.
- Now P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.
- Control of competition inevitably involves the OS because it is the OS that allocates resources. In addition, the processes themselves will need to be able to express the requirement for mutual exclusion in some fashion, such as locking a resource prior to its use. Any solution will involve some support from the OS, such as the provision of the locking facility.
- Fig. 3.5.1 Illustrates the mutual exclusion mechanism in abstract terms. There are *n* processes to be executed concurrently.

```
/* PROCESS 1 */
void P1
{
while (true)
{
/* preceding code /;
entercritical (Ra);
/* critical section */;
exitcritical (Ra);
/* following code */;
}
}
```

```
/* PROCESS 2 */
void P2
{
 while (true)
 {
 /* preceding code */;
 entercritical (Ra);
 /* critical section */;
 exitcritical (Ra);
 /* following code */;
 }
}
```

```
*****
```

```
/* PROCESS n */
void Pn
{
while (true)
 {
 /* preceding code */;
 entercritical (Ra);
 /* critical section */;
 exitcritical (Ra);
 /* following code */;
 }
}
```

Fig. 3.5.1 : Illustration of Mutual Exclusion

- **Each process includes :**
  1. A critical section that operates on some resource Ra, and
  2. Additional code preceding and following the critical section that does not involve access to Ra.
- Because all processes access the same resource Ra, it is desired that only one process at a time be in its critical section.
- To enforce mutual exclusion, **two functions are provided** :
  - entercritical and
  - exitcritical
- Each function takes as an argument the name of the resource that is the subject of competition. Any process that attempts to enter its critical section while another process is in its critical section, for the same resource, is made to wait.
- It remains to examine specific mechanisms for providing the functions entercritical and exitcritical. For the moment, we defer this issue while we consider the other cases of process interaction.

3.5.2 Cooperation among Processes by Sharing

- The case of cooperation by sharing covers processes that interact with other processes without being explicitly aware of them. Let's say multiple processes may have access to shared variables or to shared files or databases. Processes may use and update the shared data without reference to other processes but know that other processes may have access to the same data. Thus, the processes must cooperate to ensure that the data they share are properly managed. The control mechanisms must ensure the integrity of the shared data.
- Since data are held on resources (devices, memory), the control problems of mutual exclusion, deadlock, and starvation are again present. The only difference is that data items may be accessed in two different modes,
  - reading and writing, and
  - Only writing operations must be mutually exclusive.
- New requirement is introduced with existing above

mentioned Problem is of data coherence. Let's consider a application in which many data items is to be updated.

- Suppose two items of data x and y are to be maintained in the relationship x = y. This means any program that updates one value must also update the other to maintain the relationship.

(a) Now consider the following two processes P1 and P2 : if any program updates one value must also update the other to maintain the relationship. Now consider the following two processes :

P1: x = x + 10; y = y + 10;

P2: y = 20 * y; x = 20 * x;

If the state is initially consistent, each process taken separately will leave the shared data in a consistent state.

(b) Now consider the following concurrent execution sequence, in which the two processes respect mutual exclusion on each individual data item (*x* and *y*):

x = x + 10; y = 20 * y;

y = y + 10; x = 20 * x;

- At the end of this execution sequence, the condition *x* = *y* no longer holds. For example, if we start with x = y = 10, at the end of this execution sequence we have x = 210 and y = 400. The problem can be avoided by declaring the entire sequence in each process to be a critical section.
- This concludes that the concept of critical section is important in the case of cooperation by sharing. The same abstract functions of **enter critical** and **exit critical** (Fig. 3.5.1) can be used here. In this case, the argument for the functions could be a variable, a file, or any other shared object.
- Also, if critical sections are used to provide data integrity, then there may be no specific resource or variable that can be identified as an argument. In that case, we can think of the argument as being an identifier that is shared among concurrent processes to identify critical sections that must be mutually exclusive.

3.5.3 Cooperation among Processes by Communication

In the first two cases that we have discussed,

(a) First Case

- Each process has its own isolated environment that does not include the other processes.
- The interactions among processes are indirect.
- In both cases, there is a sharing.
- In the case of competition, they are sharing resources without being aware of the other processes.

(b) Second Case

- They are sharing values, and although each process is not explicitly aware of the other processes, it is aware of the need to maintain data integrity.
- When processes cooperate by communication, however, the various processes participate in a common effort that links all of the processes.
- The communication provides a way to synchronize, or coordinate, the various activities.

Typically, communication can be characterized as consisting of messages of some sort. Primitives for sending and receiving messages may be provided as part of the programming language or provided by the OS kernel.

Because nothing is shared between processes in the act of passing messages,

- Mutual exclusion is not a control requirement for this sort of cooperation. But, the problems of deadlock and starvation are still present.
- In deadlock, two processes may be blocked, each waiting for a communication from the other.
- In starvation, consider three processes, P1, P2, and P3, that exhibit the following behavior. P1 is repeatedly attempting to communicate with either P2 or P3, and P2 and P3 are both attempting to communicate with P1.
- A sequence could arise in which P1 and P2 exchange information repeatedly, while P3 is blocked waiting for a communication from P1. There is no deadlock, because P1 remains active, but P3 is starved.

3.6 SEMAPHORES

UQ. Write a short note on semaphore with example. **(SPPU - May 18, May 19)**

UQ. Explain with definition, the concept of general Semaphore. **(SPPU - Q. 3(a), Dec. 17, 2 Marks)**

Semaphore in OS and programming language **software approach** or mechanisms that are used to provide concurrency. Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a **simple integer value**, which is known as a semaphore.

Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Table 3.6.1 shows common concurrency mechanism.

Table 3.6.1 : Common Concurrency Mechanism

Mechanism	Explanation
Semaphore	An integer value used for signalling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**.
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

3.6.1 Types of Semaphores

UQ. Define binary semaphore. **(SPPU - Q. 3(a), Dec. 17, 2 Marks)**

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

1. **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.
2. **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.

Wait and Signal

GQ. Show that if the wait and signal semaphore operations are not executed automatically, then manual exclusion may be violated.

UQ. Explain how semaphore is used to solve critical section problem. **(SPPU - May 19)**

UQ. What is busy waiting with respect to process synchronization ? Explain how semaphore solves problem of synchronization. **(SPPU - Q. 6(a), Oct.19, 6 Marks)**

- In busy waiting process keeps checking some condition continuously without any productive result
- The main problem with semaphores is that they require busy waiting.
- If a process is in the critical section, then other processes trying to enter critical section will be waiting until the critical section is not occupied by any process.
- Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle.
- There is also a chance of "spinlock" as the processes keep on spins while waiting for the lock.

The **solutions of the critical section problem** represented in the section are not easy to generalize to more complex problems. To avoid this complicatedness, we can use a synchronization tool called as **semaphore**. A semaphore S is an integer variable that,

apart from initialization, is accessed only through two standard atomic operations: wait and signal.

- **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.

```
wait(S)
 {
      while (S<=0);
      S--;
 }
```

- **Signal :** The signal operation increments the value of its argument S.

```
signal(S)
 {
      S++;
 }
```

☞ Some point regarding P and V operation

- P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
- Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in between read, modify and update no other operation is performed that may change the variable.
- A critical section is surrounded by both operations to implement process synchronization as shown in Fig. 3.6.1. Critical section of Process P is in between P and V operation.

☞ Process P

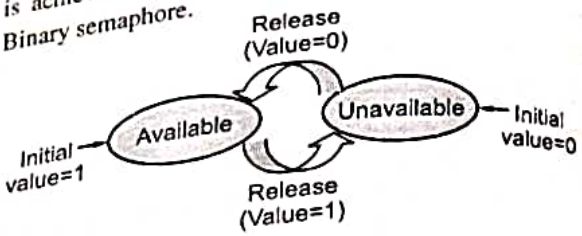
```
//Some code
     P(S)
//critical Section
     V(S)
//remainder Section
```

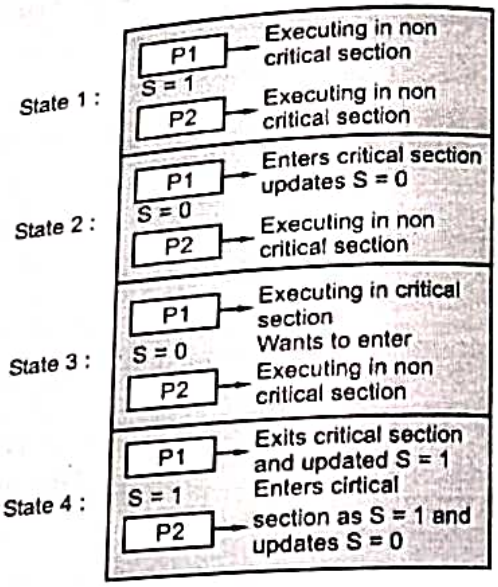
Fig. 3.6.1 : Critical Section of Process P

☞ Implementation of Binary Semaphore (Mutual Exclusion)

UQ. Explain with definition, the concept of binary semaphore. **(SPPU - Q. 3(a), Dec. 17, 3 Marks)**

- Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details which is Binary semaphore.





(1C7 & 8)Fig. 3.6.2 : Binary Semaphore

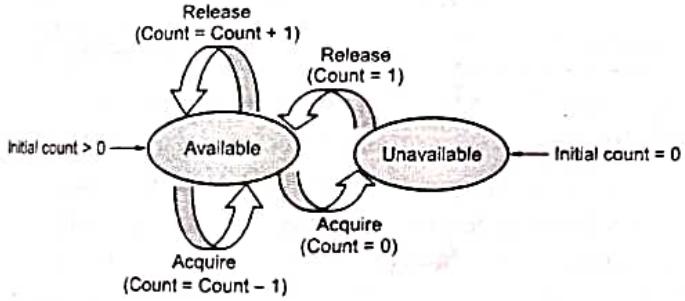
☞ Sample Code for Implementation of Binary semaphore

```
struct semaphore {
 enum value(0, 1);

 // q contains all Process Control Blocks (PCBs)
 // corresponding to processes got blocked
 // while performing down operation.
 Queue<process> q;

} P(semaphore s)
{
 if (s.value == 1) {
  s.value = 0;
 }
 else {
  // add the process to the waiting queue
  q.push(P)
  sleep();
```

```
}
}
V(Semaphore s)
{
if (s.q is empty) {
 s.value = 1;
}
else {
 // select a process from waiting queue
 q.pop();
 wakeup();
}
}
```

Counting Semaphore

- The description above is for binary semaphore which can take only two values 0 and 1 and ensure the mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.
- Now suppose there is a resource whose number of instance is 4. Now we initialize S = 4 and rest is same as for binary semaphore. Whenever process wants that resource it calls P or wait function and when it is done it calls V or signal function.
- If the value of S becomes zero then a process has to wait until S becomes positive.
- For example, Suppose there are 4 process P1, P2, P3, P4 and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls signal function and value of semaphore becomes positive.



(1C8a)Fig. 3.6.3 : Counting Semaphore

Sample Code for Implementation of Counting Semaphore

```
struct Semaphore {
 int value;

 // q contains all Process Control Blocks(PCBs)
 // corresponding to processes got blocked
 // while performing down operation.
 Queue<process> q;

} P(Semaphore s)
{
 s.value = s.value - 1;
 if (s.value < 0) {

  // add process to queue
  // here p is a process which is currently executing
  q.push(p);
  block();
 }
 else
  return;
}

V(Semaphore s)
{
 s.value = s.value + 1;
 if (s.value <= 0) {

  // remove process p from queue
  q.pop();
  wakeup(p);
 }
 else
  return;
}
```

In this implementation whenever process waits it is added to a waiting queue of processes associated with that semaphore. This is done through system call block () on that process. When a process is completed it calls signal function and one process in the queue is resumed. It uses wakeup () system call.

3.6.2 Advantages and Disadvantages of Semaphores

Advantages

1. 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.
2. 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.

3. Semaphores are implemented in the machine independent code of the microkernel. So, they are machine independent.

Disadvantages

1. Semaphores are complicated as the wait and signal operations must be implemented in the correct order to prevent deadlocks.
2. Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
3. Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

3.6.3 Difference between Counting and Binary Semaphores

GQ. Differentiate between Counting and Binary Semaphores.

Sr. No.	Parameter	Binary Semaphore	Counting Semaphore
1.	**Counter**	The counter logically goes between 0 and 1.	A counting semaphore has multiple values for count.
2.	**Handle**	Cannot handle Bounded wait as it's just a variable that hold binary value.	It can handle bounded wait as it has converted a variable into a structure with a Queue.
3.	**Implementation**	Structure implementation Binary semaphore : int s	Counting Semaphore : Struct S { int s; Queue q; }
4.	**Mutual Exclusion**	Whereas in binary semaphore only two processes can enter in to the CRITICAL SECTION at any time and mutual exclusion is also guaranteed.	In counting semaphore more than two processes can enter in to the CRITICAL SECTION at any time (The value of the semaphore variable decides the number of processes which can enter in to the critical section at a time), Mutual exclusion is not guaranteed.

3.6.4 Mutex and Monitor

GQ. What is Mutex and Monitor ? Explain

(I) Mutex

- A mutex is a binary variable whose purpose is to provide locking mechanism. It is used to provide mutual exclusion to a section of code, means only one process can work on a particular code section at a time.
- At times, there may be multiple threads that may be trying to access same resource like memory or I/O etc. To make sure that there is no overriding. Mutex provides a locking mechanism.
- Only one thread at a time can take the ownership of a mutex and apply the lock. Once it done utilizing the resource and it may release the mutex lock.
- The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section.
- This is shown with the help of the following example

```
wait (mutex);
.....
Critical Section
.....
signal (mutex);
```

- A Mutex is different than a semaphore as it is a locking mechanism while a semaphore is a signaling mechanism. A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore.

(II) Monitor

UQ. Explain monitors in brief.

(SPPU - Q. 3(b), May 18, Q. 4(b), May 19, Q. 3(a), Dec. 19, 5 Marks)

- Monitors and semaphores are used for process synchronization and allow processes to access the shared resources using mutual exclusion. However, monitors and semaphores contain many differences. Details about both of these are given as follows –
- Monitors are a synchronization construct that were created to overcome the problems caused by semaphores such as timing errors.
- Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.

- This is demonstrated as follows :

```
monitor monitorName
{
 data variables;

 Procedure P1(....)
 {

 }

 Procedure P2(....)
 {

 }

 Procedure Pn(....)
 {

 }

 Initialization Code(....)
 {

 }
}
```

- Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

3.6.5 Difference between Semaphore and Monitor

Parameter	SEMAPHORE	MONITOR
Basic	Semaphores is an integer variable S.	Monitor is an abstract data type.
Action	The value of Semaphore S indicates the number of shared resources available in the system	The Monitor type contains shared variables and the set of procedures that operate on the shared variable.
Access	When any process access the shared resources it perform wait() operation on S and when it releases the shared resources it performs signal() operation on S.	When any process wants to access the shared variables in the monitor, it needs to access it through the procedures.
Condition variable	Semaphore does not have condition variables.	Monitor has condition variables.

3.6.6 Difference Between in Semaphore and Mutex

Parameters	Semaphore	Mutex
Mechanism	It is a type of signalling mechanism.	It is a locking mechanism.
Data Type	Semaphore is an integer variable.	Mutex is just an object.
Modification	The wait and signal operations can modify a semaphore.	It is modified only by the process that may request or release a resource.
Resource management	If no resource is free, then the process requires a resource that should execute wait operation. It should wait until the count of the semaphore is greater than 0.	If it is locked, the process has to wait. The process should be kept in a queue. This needs to be accessed only when the mutex is unlocked.
Thread	You can have multiple program threads.	You can have multiple program threads in mutex but not simultaneously.
Ownership	Value can be changed by any process releasing or obtaining the resource.	Object lock is released only by the process, which has obtained the lock on it.
Types	Types of Semaphore are counting semaphore and binary semaphore.	Mutex has no subtypes.

Parameters	Semaphore	Mutex
Operation	Semaphore value is modified using wait () and signal () operation.	Mutex object is locked or unlocked.
Resources Occupancy	It is occupied if all resources are being used and the process requesting for resource performs wait () operation and blocks itself until semaphore count becomes >1.	In case if the object is already locked, the process requesting resources waits and is queued by the system before lock is released.

3.6.7 POSIX Semaphores in Linux

UQ. Explain following functions (along with parameter passed) with reference to semaphore programming in 'C'.

(i) sem_wait() (ii) sem_post()

(SPPU - Q. 4(a)OR, Dec. 17, 5 Marks)

POSIX semaphore calls are much simpler than the System V semaphore calls. However, System V semaphores are more widely available, particularly on older Unix-like systems. POSIX semaphores have been available on Linux systems post version 2.6 that use glibc.

- There are two types of POSIX semaphores :

  (A) named and (B) unnamed.

- As the terminology suggests, named semaphores have a name, which is of the format */somename*. The first character is a forward slash, followed by one or more characters, none of which is a slash. We will first look at the named semaphores and then the unnamed ones.
- Programs using POSIX semaphores need to be linked with the pthread library.

(A) POSIX Named Semaphore calls

1. sem_open

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
sem_t *sem_open (const char *name, int oflag);
sem_t *sem_open (const char *name, int oflag,
        mode_t mode, unsigned int value);
```

- sem_open is the call to get started for a semaphore. sem_open opens an existing semaphore or creates a new semaphore and opens it for further operations.
- The first parameter, *name*, is the name of the semaphore, coined as described earlier. The *oflag* can have O_CREAT, in which case, the semaphore is created if it does not already exist. If both O_CREAT and O_EXCL are specified, the call gives an error, if the semaphore with the specified name already exists.
- If the *oflags* parameter has O_CREAT set, the second form of sem_open has to be used and two additional parameters, *mode* and *value* have to be specified. The *mode* parameter specifies the permissions for the semaphore, which are masked with the umask for the process, similar to the *mode* in the open system call for files.
- The last parameter, *value* is the initial value for the semaphore. If O_CREAT is specified in *oflag* and the semaphore already exists, both the *mode* and *value* parameters are ignored.
- sem_open returns a pointer to the semaphore on success. This pointer has to be used in the subsequent calls for the semaphore. If the call fails, sem_open returns SEM_FAILED and errno is set the appropriate error.
- Under Linux, POSIX semaphores are created under the /dev/shm directory. The semaphores are named with a prefix, sem. followed by the *name* passed in the sem_open call.

2. sem_post

```
#include <semaphore.h>
int sem_post (sem_t *sem);
```

- sem_post increments the semaphore. It provides the V operation for the semaphore. It returns 0 on success and -1 on error.

3. sem_wait

```
#include <semaphore.h>
int sem_wait (sem_t *sem);
```

- sem_wait decrements the semaphore pointed by *sem*. If the semaphore value is non-zero, the decrement happens right away.
- If the semaphore value is zero, the call blocks till the time semaphore becomes greater than zero and the decrement is done. sem_wait returns zero on success and -1 on error.
- In case of error, the semaphore value is left unchanged and errno is set to the appropriate error number. sem_wait provides the call for the P operation for the semaphore.

4. sem_trywait

```
#include <semaphore.h>
int sem_trywait (sem_t *sem);
```

- sem_trywait is just like the sem_wait call, except that, if the semaphore value is zero, it does not block but returns immediately with errno set to EAGAIN. In other system calls we have flags like IPC_NOWAIT, but here, we have a full-fledged system call for that purpose.

5. sem_timedwait

```
#include <semaphore.h>
int sem_timedwait (sem_t *sem, const struct timespec
*abs_timeout);
```

- sem_timedwait is also like the sem_wait call, except that, there is timer specified with the pointer, *abs_timeout*.
- If the semaphore value is greater than zero, it is decremented and the timeout value pointed by abs_timeout is not used. That is, in that case, the call works just like the sem_wait call. If the semaphore value is zero, the call blocks, the maximum duration of blocking being the time till the timer goes off.
- If the semaphore value becomes greater than zero during the blocking period, the semaphore is decremented immediately and the call returns. Otherwise, the timer goes off and the call returns with errno set to ETIMEDOUT.
- The timer is specified in the struct timespec, which is,

```
struct timespec {
  time_t tv_sec;      /* Seconds */
  long   tv_nsec;     /* Nanoseconds [0 .. 999999999] */
};
```

6. sem_getvalue

```
#include <semaphore.h>
int sem_getvalue (sem_t *sem, int *sval);
```

- sem_getvalue gets the value of semaphore pointed by *sem*. The value is returned in the integer pointed by *sval*. It returns 0 on success and -1 on error, with errno indicating the actual error.

7. sem_unlink

```
#include <semaphore.h>
int sem_unlink (const char *name);
```

- sem_unlink removes the semaphore associated with the *name*.

(B) POSIX Unnamed Semaphore calls

- The semaphores explained above are local to a process; they are only used by its threads. No other process uses them. So, it looks like a waste of effort to have system-wide semaphore names and use calls like sem_open. There are POSIX unnamed semaphores which can do what we need in a much simpler and efficient manner. First, the system calls,

1 sem_init

```
#include <semaphore.h>
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

- sem_init is the equivalent of sem_open for unnamed semaphores.
- One defines a variable of type sem_t and passes its pointer as *sem* in the sem_init call. Or, one can define a pointer and allocate memory dynamically using malloc or a similar function call. sem_init initializes the semaphore pointed by *sem* with the *value*.
- The second argument *pshared* indicates whether the semaphore is shared between threads of a process or between processes. If *pshared* has the value 0, the semaphore is shared between threads of a process.
- The semaphore should be placed at a place where it is visible to all threads. If *pshared* has a nonzero value, it indicates that the semaphore is shared by processes. In that case the semaphore has to be placed in a shared memory segment which is attached to the concerned processes.

2. sem_destroy

```
#include <semaphore.h>
int sem_destroy (sem_t *sem);
```

sem_destroy destroys the unnamed semaphore pointed by *sem*.

3.7 CLASSIC PROBLEMS OF SYNCHRONIZATION

- There are number of classical problems of synchronization as examples of a large class of concurrency-control problems. The solutions to the problems, is semaphores can be used for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.
- These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems :
  1. Producer-Consumer (or Bounded-buffer) Problem,
  2. Dining-Philosophers Problem,
  3. Readers and Writers Problem,
  4. Sleeping Barber Problem

3.7.1 The Producer-Consumer Problem (Bound Buffer Problem)

UQ. Write a note on : Producer consumer problem. **(SPPU - May 18, Dec. 18)**

UQ. Give semaphore solution for producer-consumer problem. **(SPPU - Q. 4(a), May 19, 5 Marks)**

(1) The programs we write usually divided into multiple functions and these functions are called **procedures**. The

program in execution is called a **process**; therefore in a sequential program execution when one procedure calls another procedure and passes data, they do not execute concurrently because they belong to a single process. In the concurrent process executions one process may pass data to another process during communications with each other. Such communication among processes is called as inter process communication.

(2) The producer-consumer is one of the classic problems discussed in association with Inter process communication by almost all the researchers of operating system analysis and design field. One problem we may consider for description is the print-queue management by the operating system. This is based on the multiprogramming systems where multiple processes run as producers of output files to be printed.

(3) The process or processes used to print using one printer or many printers if available with the machine called as consumer(s) processes. The print-queue which has a limited number of slots is the shared resource.

(4) The processes used to produce items in the print queue for printing are called as producer processes and one process or many processes in case of multiple printers are called as consumer processes. The consumer(s) are used to take the items from the shared queue for printing using the printer.

(5) The producer and consumer are the names used in general for certain types of communicating processes in the problems, similar to the print - queue problem. Producer(s) produces items or entities and store them in the shared queue or shared buffer space. Consumer(s) take items or entities from the shared space and use them for the purpose they are designed to do.

(6) There is a class of problems in which producer processes place items in a buffer and consumer processes take it out from that buffer.

(7) This always as synchronization and mutual exclusion issues. A process attempting to take items from an empty buffer must be in the waiting state for availability of an item in the buffer. In the similar fashion a process putting items into a full buffer must also be blocked. Here the buffer must be protected from simultaneous access by more than one process.

(8) As an example we can assume a circular buffer with N number of fixed – size slots. The slots are numbered from 1 to N. The last slot is N, after filling up of slot N the next slot to be filled is slot – 1again. The Fig. 3.7.1 shows a six slot buffer. The Fig. 3.7.1 shows a buffer with six slots in Fig. 3.7.1(b) and its memory representation is shown in Fig. 3.7.1(a).

(9) The algorithmic description uses the circular buffer representation as in Fig. 3.7.1(b) but during execution a process uses the memory representation shown in Fig. 3.7.1(a).

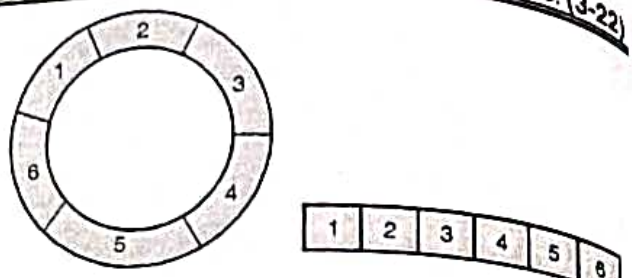


Fig. 3.7.1 : Circular buffer with 6 slots

3.7.1(A) Producer-Consumer Problem using Semaphore

UQ. How to solve producer-consumer problem using Semaphore and Mutex ?

(SPPU - Q. 4(a), May 17, Dec. 18, 6 Marks)

Using Mutex

- A mutex provides **mutual exclusion**, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa.
- At any point of time, only one thread can work with the entire buffer. The concept can be generalized using semaphore.

Using Semaphore

A **semaphore** is a **generalized mutex**. In lieu of a single buffer, here single buffer can be splatted into multiple buffers. A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Design

- To design the system for producer-consumer problem, we start by listing the constraints and assigning a semaphore to each constraint that a semaphore can enforce.
- The semaphore can be used for **mutual exclusion** as well as other constraints required for handling synchronized access to the buffer by the producer and consumer processes.
- As we have already discussed previously about the buffer that; it is used to put items by the producer processes and consumer processes are used to retrieve items from that buffer.
  1. Enforce mutual exclusion for accessing the item buffer.
  2. If buffer is full then do not allow to produce an item in the buffer by the producer.
  3. If buffer is empty then consumer is not allowed to retrieve items from it.
- The variable **available** is used here to show the number of slots available in the buffer currently.

- In the similar manner the variable occupied is used to show how many slots of the buffer are occupied at any given time. It is also used to check whether or not the buffer is full.
- The two variables available and occupied used as semaphore variables are initialized with some initial values. The available semaphore is initialized with the size of the buffer which is represented in terms of slots.
- This makes sense as it represents the number of available buffer slots at the start-up of the producer-consumer processes.
- In the similar fashion the occupied semaphore is initialized with value zero. The semaphore based solution for the producer consumer problem is described below.

```
/*Global variables*/
#define true 1

#define Buffer_Capacity          100/*Buffer capacity*/
typedefint semaphore;
        /*typedefint as semaphore to declare variables */

semaphore available = Buffer_Capacity;
                                /*All slots are available*/
semaphore occupied = 0;         /*Buffer is empty*/
semaphore mutex =1;
                        /*Shared Resource is not in use*/
```

3.7.1(B) Producer Function Definition

```
void producer_function ( )
{
   structbuffer_slot item;
   while (true)
   {
      produce_an_item(&item);
      wait (&available);
                  /*calling process will wait for free slot*/
      wait (&mutex);                //Get permission
      insert_an_item(&item);  //Put item in queue
      signal (&mutex);              //Free resource
      signal (&occupied);           //One slot is filled
   }
}
```

In the producer function, the shared buffer is grabbed after the item is produced and is ready to be inserted in the buffer.

3.7.1(C) Consumer Function Definition

```
void consumer_function ( )
{
    structbuffer_slot item;
    while (true)
    {
        wait (&occupied);
                              /*Wait until there is an entity*/
        wait (&mutex);      //Get permission
        remove_an_item(&item);
                              /*remove itemfrom the queue*/
        signal (&mutex);
        signal (&available);        //One slot is filled
        consume_an_item(&item);
    }
}
```

- In the producer - consumer problem; when a process is in sleeping state, other processes have to wake it up. It is usually not possible for a process to wake up itself, after having fallen asleep.
- The statement wait(&available) is executed by the producer then it goes into sleeping state when the buffer is full. The signal(&available) operation is performed by the consumer to wake up the sleeping producer.
- The producer and consumer processes execute two procedures to perform predefined operations indefinitely. The two processes will be required to create for the executions of these two procedures to start producer and consumer process. The following is the definition of the main function of the program.

```
void main (){
    intpid;
    pid = fork();
    if (pid != 0) {
        producer;
    }else{
        consumer;
    }
}
```

3.7.1(D) One Producer and One Consumer Process

- This is the simplest case of producer consumer problem in which only one producer process and only one consumer process present.
- This form of producer consumer needs two semaphores. First semaphore is associated with the maximum number of free slots in the buffer and the second is associated with the available items.

- First semaphore is initialized with the maximum number of slots in the buffer whereas second semaphore is initialized to 0, as there are no items available at initialization time.
- Let us call first semaphore as slot_free_sem and second semaphore as available_sem. At any instance of time slot_free_sem contains the number of free slots in the buffer and available_sem contains number of unconsumed items in the buffer.
- In this case there are two pointers used and they are **NextSlotIn** and **NextSlotOut**. **NextSlotIn** points to the next slot to be used by the producer process whereas **NextSlotOut** is used to point the slot from where the consumer will consume an item.
- At the startup both pointers point to the first slot. When these get to the end of the buffer, it wraps around to point to the first slot again.

3.7.1(E) Algorithm for Producer and Consumer process

- The **algorithm for the producer** process is given in below :
  1. Produce an item to be stored in the buffer
  2. wait(slot_free_sem); //maximum number of free slots
  3. Put an item in the buffer at the pointer NextSlotIn
  4. Increment NextSlotIn
  5. signal(available_sem)
- The corresponding **algorithm for the consumer** is shown in below :
  1. wait(available_sem)
  2. Get an item from buffer at NextSlotOut
  3. Increment NextSlotOut
  4. Signal(slot_free_sem); //maximum number of free slots
  5. Consume the item
  6. signal()
- The consumer should not be allowed to retrieve item from the empty buffer; the consumer process will be blocked in the wait() operation on the semaphore available_sem, which is 0.
- The consumer process woke up when the producer performs signal() operation on the same semaphore on which consumer was waiting after putting an item on the buffer, that is available_sem in this case.
- In the similar way the producer will be blocked when the buffer is full; this happens because producer performs wait() operation on the semaphore slot_free_sem, which now has a value of 0.
- The producer only get past of this point, when the consumer performs signal() operation on thesemaphore on which producer was waiting after retrieving the item from the buffer, that is slot_free_sem.

3.7.1(F) Multiple Producer Processes and Multiple Consumer Processes

- In case of multiple producers and consumers the same semaphores and pointers are used as in the previous case discussed.
- In addition we must use another mutual exclusion semaphore named as guard and initialized with 0.
- This mutual exclusion semaphore guard is used to protect a slot from two or more producer processes, because they could try to put an item into the same slot at the same time.
- The algorithm for a producer and consumer are given in below. All of the comments on the previous pair of algorithms apply here.

```
Produce an item
wait(slot_free_sem);
                        //maximum number of free slots
wait(guard)
                    /*guard Semaphore for mutual exclusion*/
Put an item in the buffer at the pointer NextSlotIn
Increment NextSlotIn
signal(guard)
signal(available_sem)
wait(available_sem);
                        //maximum number of free slots
wait(guard)
                    /*guard Semaphore for mutual exclusion*/
Get an item from buffer at the pointer NextSlotOut
Increment NextSlotOut
signal(guard)
signal(slot_free_sem)
consume the item
```

- Only one process can be active in the buffer at a time; because the guard semaphore implements mutual exclusion so that only one producer process or consumer process will be active at a time in the buffer.
- It is also possible to have two mutual exclusion semaphores; one each for producer and consumer processes.
- One semaphore would be associated with the producer process so that there would be a guarantee about the using of one slot by only one producer at one time.
- In the similar fashion second semaphore would be associated with the consumer process so that only one consumer would be allowed to consume an item from the associated buffer slot at one time. In this case, one producer and one consumer could be active in the buffer at the same time.

- The special attention is required here about the order in which a wait() operation is performed. If a producer process performed the wait() operation on guard semaphore before performing wait() on slot_free_sem semaphore, then no consumer process could get past from the wait() operation on guard semaphore and the system would stuck in deadlock situation.

3.7.2 Reader Writer Problem

UQ. Write a note on reader writer problem in process synchronization. **(SPPU - May 19)**

UQ. Explain Readers-Writers problem with example. **(SPPU - Q. 4(b), May 15, 3 Marks, (Q. 3(b), May 16, 2 Marks)**

UQ. Write a semaphore solution for readers – writers problem. **(SPPU - Q. 4(a), May 18, Dec. 18, Q. 1(a), Dec. 18, Q. 5(b), Oct. 19, 5 Marks)**

1. Another famous problem is the readers and writers problem which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it.
2. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader.
3. Consider a situation where we have a file shared between many people.
4. If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
5. However if some person is reading the file, then others may read it at the same time. Precisely in OS we call this situation as the **readers-writers problem**

(a) Problem parameters

1. One set of data is shared among a number of processes.
2. Once a writer is ready, it performs its write. Only one writer may write at a time.
3. If a process is writing, no other process can read it.
4. If at least one reader is reading, no other process can write.
5. Readers may not write and only read.

(b) Solution to Readers Writer problems using semaphore

```
typedef int semaphore;        /* use your imagination */
semaphore mutex = 1;/* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */
int rc = 0;    /* # of processes reading or wanting to */
void reader(void)
{
    while (TRUE)
    {       /* repeat forever */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc + 1; /* one reader more now */
            if (rc == 1) down(&db);
                                /* if this is the first reader ... */
                    up(&mutex);
                        /* release exclusive access to 'rc' */
            read_data_base();
                                        /* access the data */
            down(&mutex);
                    /* get exclusive access to 'rc' */
            rc = rc 1;      /* one reader fewer now */
            if (rc == 0) up(&db);
                                /* if this is the last reader ... */
                up(&mutex);
                        /* release exclusive access to 'rc' */
            use_data_read();    /* noncritical region */
    }
}
void writer(void)
{
    while (TRUE)
        {                           /* repeat forever */
            think_up_data();    /* noncritical region */
            down(&db);          /* get exclusive access */
            write_data_base();  /* update the data */
            up(&db);          * release exclusive access */
        }
}
```

In this solution, the first reader to get access to the data base does a down on the semaphore db. Subsequent readers merely have to increment a counter, rc. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

3.7.3 Dining Philosophers Problems

UQ. Explain dining philosopher's problem with example. **(SPPU - Q. 4(b), May 15, 3 Marks, Q. 3(b), May 16, May 19, 2 Marks)**

UQ. What is difference between starvation and deadlock ? Explain it with the help of 'Dining Philosophers Problem. **(SPPU- Q. 3(b), May 17, May 18 6 Marks)**

UQ. Write a semaphore solution for dining philosopher's problem. **(SPPU - Q. 3(b), Dec. 17, Q. 4(a), Dec. 19, 5 Marks)**

UQ. Write and explain the deadlock free solution for a Dinning Philosopher problem.

(SPPU - Q. 3(a), May 19, 5 Marks)

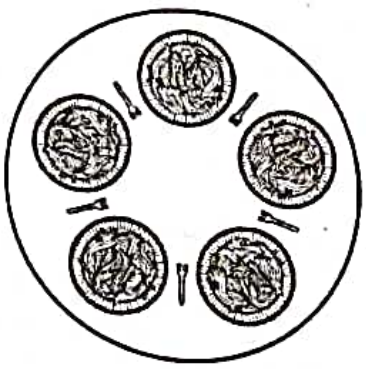
There are N philosphers sitting around a circular table eating spaghetti and discussing philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that ensures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock

The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible.

One idea is to instruct each philosopher to behave as follows:

1. think until the left fork is available; when it is, pick it up
2. think until the right fork is available; when it is, pick it up
3. eat
4. put the left fork down
5. put the right fork down
6. repeat from the start



(1C24)**Fig. 3.7.2 : Dining Philosopher Problem**

(a) Incorrect Solution leading to Deadlock and Starvation

- It allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a **deadlock**.
- The program can be modified so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason.
- But if all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, and picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called **starvation.**
- The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

(b) Solution

```
#define N 5                     /* number of philosophers */
#define LEFT (i+N-1)%N  /* number of i's left neighbor */
#define RIGHT (i+1)%N   /* number of i's right neighbor */
#define THINKING 0      /* philosopher is thinking */
#define HUNGRY 1   /* philosopher is trying to get forks */
#define EATING 2        /* philosopher is eating */
typedef int semaphore;
                        /* semaphores are a special kind
of int */
int state[N];       /* array to keep track of everyone's state */
semaphore mutex = 1;
                /* mutual exclusion for critical regions */
semaphore s[N];
                        * one semaphore per philosopher */
void philosopher(int i)
                /* i: philosopher number, from 0 to N1 */
{
    while (TRUE)
        {
            think();                /* repeat forever */
            take_forks(i);  /* philosopher is thinking */
            eat();          /* acquire two forks or block */
            put_forks(i);           /* yum-yum, spaghetti */

        }                   /* put both forks back on table */
}
void take_forks(int i)

to N1 */                    /* i: philosopher number, from 0
{

    down(&mutex);
    state[i] = HUNGRY;              /* enter critical region */
```

```
                                /* record fact that philosopher i
is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);            /* block if forks were not acquired */
}
void put_forks(i)
                            /* i: philosopher number, from 0 to N1
                                                              */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;
                            /* philosopher has finished eating */
    test(LEFT);             /* see if left neighbor can now eat */
    test(RIGHT);            /* see if right neighbor can now eat */
    up(&mutex);             /* exit critical region */
}
void test(i)          /* i: philosopher number, from 0 to N1* /
{
    If (state[i] == HUNGRY && state[LEFT] != EATING
&& state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
}
}
```

3.7.4 Difference between Deadlock and Starvation

UQ. What is difference between starvation and deadlock? **(SPPU - Q. 3(b), May 17, 6 Marks)**

Parameter	Deadlock	Starvation
Basic	Deadlock is where no process proceeds, and get blocked.	Starvation is where low priority processes get blocked, and high priority process proceeds.
Arising condition	The occurrence of Mutual exclusion, Hold and wait, No preemption and Circular wait simultaneously.	Enforcement of priorities, uncontrolled resource management.
Other name	Circular wait.	Lifelock.
Resources	In deadlocked, requested resources are blocked by the other processes.	In starvation, the requested resources are continuously used by high priority processes.
Prevention	Avoiding mutual exclusion, hold and wait, and circular wait and allowing preemption.	Aging.

3.8 INTERPROCESS COMMUNICATION

UQ. Why there is need for communication between two processes ? Explain various modes of communication. **(SPPU - Dec. 15, May 16, May 18)**

UQ. List and explain different inter-process communication mechanisms in the Linux operating system. **(SPPU - Q. 9(b), Dec. 18, Q. 9(b), Q. 9(b), May 18, Dec. 19, 8 Marks)**

UQ. State and explain different Linux inter-process communication mechanisms. **(SPPU - Q. 9(b), May 19, 8 Marks)**

UQ. Describe in brief different IPC mechanisms. **(SPPU - Q. 3(a), Dec. 18, 5 Marks)**

UQ. Enlist and explain different IPC mechanisms. **(SPPU - Q. 3(a), May 18, 5 Marks)**

UQ. Write short note on : Linux IPC mechanisms. **(SPPU - Q. 10(c)OR, Dec. 17, Q. 10(c), May 18, 4 Marks)**

(1) Inter-Process communications or IPC, is the mechanism where in one process can communicate, that is, exchange data, with another process. While these techniques are very useful for communicating with other programs they do not provide the fine grained control that's sometimes needed for larger scale applications.

(2) In these applications it is quite common for several processes to be used, each performing a dedicated task and other processes requesting services from them.

(3) **Need of Communication between Process :** Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. Since every single user request may result in multiple processes

running in the operating system, the process may require to communicate with each other

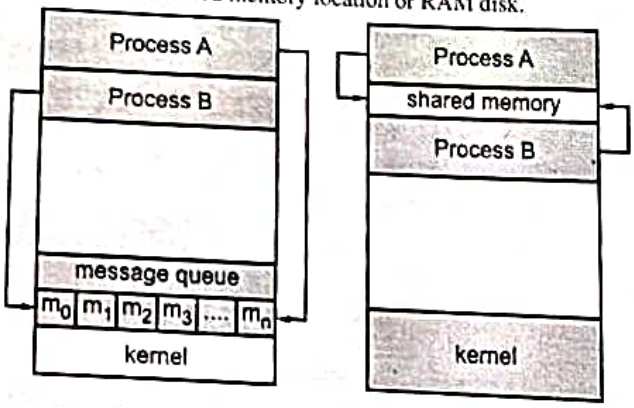
(4) **Techniques to implement IPC :** There are two fundamental models of Inter-Process communication that are commonly used, these are :

1. Shared Memory Model 2. Message Passing Model

3.8.1 Shared Memory Model

UQ. Explain the following terms under IPC : Shared Memory **(SPPU - Q. 4(b), Dec. 18, 3 Marks)**

- In shared memory model shown in Fig. 3.8.1. the cooperating process shares a region of memory for sharing of information. some operating systems use the supervisor call to create a share memory space.
- The shared files are stored in RAM disk (Virtual Disk created in RAM)to share the information between processes.
- The Process can share information by writing and reading data to the shared memory location or RAM disk.

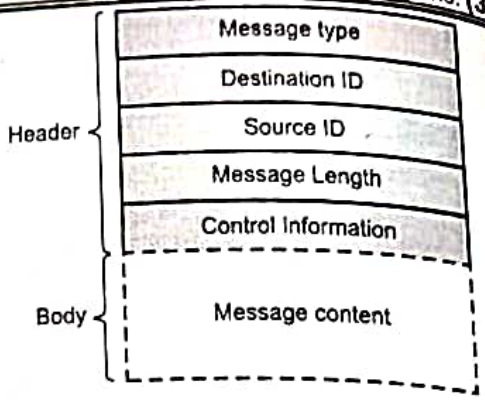


(1C4)**Fig. 3.8.1 : Inter Process Communication Model**

3.8.2 Message Passing Model

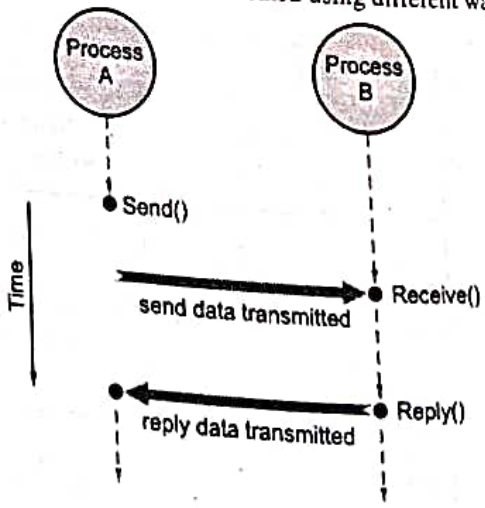
UQ. Explain the following terms under IPC : Message passing **(SPPU - Q. 4(b), Dec. 18, 3 Marks)**

- In this model shown in Fig. 3.8.2, data is shared between process by passing and receiving messages between co-operating process. Message passing mechanism is easier to implement than shared memory but it is useful for exchanging smaller amount of data. In message passing mechanism data is exchange between processes through kernel of operating system using system calls. Message passing mechanism is particularly useful in a distributed environment where the communicating processes may reside on different components connected by the network.



(1C5)**Fig. 3.8.2 : Message Format**

- For example, A data program used on the internet could be designed so that chat participants communicate with each other by exchanging messages. It must be noted that passing message technique is slower than shared memory technique.
- **A message contains the following information :**
  - Header of message that identifies the sending and receiving processes
  - Block of data
  - Pointer to block of data
  - Some control information about the process
- Typically Inter-Process Communication is based on the ports associated with process. A port represent a queue of processes. Ports are controlled and managed by the kernel. The processes communicate with each other through kernel.
- In message passing mechanism, two operations are performed. These are sending message and receiving message.
- The function send() and receive() are used to implement these operations as shown in Fig. 3.8.3. Supposed P1 and P2 want to communicate with each other. A communication link must be created between them to send and receive messages. The communication link can be created using different ways.



1C8)**Fig. 3.8.3 : Send and Receive Message**

- The most important methods are :
  - Direct model
  - Indirect model
  - Buffering

3.8.3 Methods to Implement Interprocess Communication

Inter-process communication (IPC) is a set of interfaces, which is usually programmed in other for a programmer to communicate between a series of processes. This allows the running of programs concurrently in an operating system.

There are quite a number of methods used in inter-process communications. They are :

1. **Pipes :** This allows the flow of data in one direction only. Data from the output is usually buffered until the input process receives it which must have a common origin.
2. **Named Pipes :** This is a pipe with a specific name. It can be used in processes that do not have a shared common process origin. Example is FIFO where the data is written to a pipe is first named.
3. **Message queuing :** This allows messages to be passed between messages using either a single queue or several message queues. This is managed by the system kernel. These messages are coordinated using an application program interface (API).
4. **Semaphores :** This is used in solving problems associated with synchronization and avoiding race conditions. They are integer's values which are greater than or equal to zero.
5. **Shared Memory :** This allows the interchange of data through a defined area of memory. Semaphore value has to be obtained before data can get access to shared memory.
6. **Sockets :** This method is mostly used to communicate over a network, between a client and a server. It allows for a standard connection which I computer and operating system independent.

3.8.4 Facility Provided by IPC

- IPC provides a mechanism to allow Processes to communicate and to synchronize their actions without sharing the same address space.
- IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network. An example is a chat program used on the World Wide Web.
- IPC is best provided by a message-passing system, and message systems can be defined in many ways.

3.8.5 Message Passing

- The function of a message system is to allow processes to communicate with one another without the need to resort to shared data. We have already seen message passing used as a method of communication in microkernel's. In this scheme, services are provided as ordinary user processes. That is the services operate outside of the kernel.
- Communication among the user process is accomplished through the passing of messages. An IPC facility provides at least the two operations :
  - send(message) and
  - receive(message).
- Messages sent by a process can be of either

1. Fixed	2. Variable size

1. **Fixed Sized Message :** When fixed-sized messages are sent, the system-level implementation is straightforward i.e. easy but makes the task of programming more difficult.
2. **Variable Sized Message :** If variable-sized messages needs a more complex system-level implementation, but the programming task becomes simpler.

Example

If processes X and Y want to communicate, they must send messages to and receive messages from each other, there exist a link between two processes. There are varieties of ways to implement this link. Here physical implementation is not of much concern (such as shared memory, hardware bus, or network), but logical implementation is more important. There are several methods for logically implementing a link and the send/receive operations :

1. Direct or indirect communication
2. Symmetric or asymmetric communication
3. Automatic or explicit buffering
4. Send by copy or send by reference
5. Fixed-sized or variable-sized messages

We look at each of these types of message systems next.

Naming

Processes that want to communicate can use either direct or indirect communication to refer to each other.

3.8.6 Direct Communication

With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send and receive primitives are defined as :

1. send(X, message) - Send a message to process X.
2. receive(Y, message) - Receive a message from process Y.

Properties of communication Links

A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

- A link is associated with exactly two processes.

- Exactly one link exists between each pair of processes.

There can be symmetry or not to address both sender and receiver.

☞ Symmetry in addressing

Symmetry in addressing means that, both the sender and the receiver processes must name the other to communicate.

- send(X, message) - Send a message to process X.
- receive(Y, message) - Receive a message from process Y.

☞ Asymmetry in Addressing

Another scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender and recipient name.

The send and receive primitives are defined as follows :

- Send(P, message) - Send a message to process X.
- Receive (id, message) - Receives message from any process; the variable id is set to the name of the process with which communication has taken place.

☞ Disadvantage in both symmetric and asymmetric schemes

1. Is the limited modularity of the resulting process definitions.
2. Changing the name of a process may necessitate examining all other process definitions.
3. All references to the old name must be found, so that they can be modified to the new name.
4. This situation is not desirable from the viewpoint of separate compilation.

3.8.7 Indirect Communication

The messages are sent to and Received from mailboxes, or ports in indirect communication. A mailbox can be viewed as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In indirect communication, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if they share a mailbox. The send and receive primitives are defined as follows :

- Send(P, message) - Send a message to mailbox P.
- Receive(P, message) - Receive a message from mailbox P.

☞ Properties of Communication Link

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Example

Let processes X and Y all share mailbox P. Process X sends a message to P, while Y and Z each execute a receive from P which process will receive the message sent by X.

The answer depends on the scheme that we choose :

- Allow a link to be associated with at most two processes.
- Allow at most one process at a time to execute a receive operation.

Allow the system to select arbitrarily which process will receive the message (that is, either X or Y but not both, will receive the message). The system may identify the receiver to the sender.

⏭ 3.9 INTRODUCTION TO DEADLOCK

UQ. Explain concept of deadlock.

UQ. What is deadlock ? **(SPPU - May 16, Dec. 18, May 19)**

UQ. Define Deadlock. **(SPPU - Q. 4(a), May 16, Q. 4(a), Dec. 18, Q. 3(a), May 19, 2 Marks)**

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. **This situation is called a deadlock.**

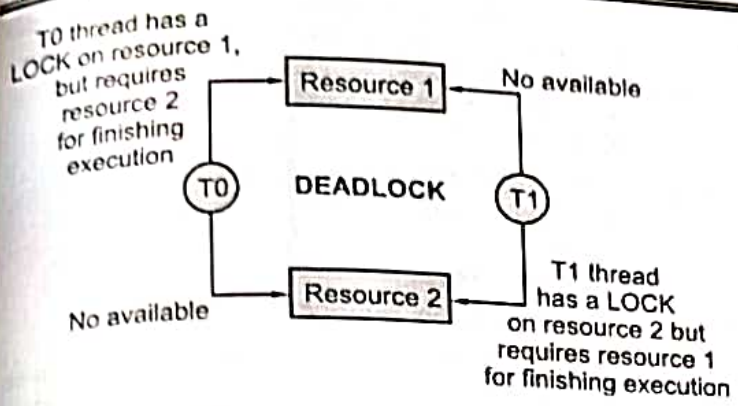
❒ Definition of Dead Lock

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function

3.9.1 System Model

☞ Resource

- A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types.
- If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type *printer* may have five instances.
- If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly.



(1C10)**Fig. 3.9.1 : Dead Lock Situation**

- For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.
- Hence a process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. But the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence :
  1. **Request :** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
  2. **Use :** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
  3. **Release :** The process releases the resource. Examples are the request and release device, open and close file, and allocate and free memory system calls.
- Request and release of other resources can be accomplished through the *wait* and *signal* operations on semaphores. Therefore, for each use, the operating system checks to make sure that the using process has requested and been allocated the resource.
- A system table is used which
  1. Records whether each resource is free or allocated, and,
  2. If a resource is allocated, to which process.
  3. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.
- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors).

Resource Acquisition

- For some kinds of resources, such as records in a database system, it is up to the user processes rather than the system to manage resource usage themselves. One way of allowing this is to associate a semaphore with each resource. These semaphores are all initialized to 1. Mutexes can be used equally well.
- The three steps listed above are then implemented as a down on the semaphore to acquire the resource, the use of the resource, and finally an up on the resource to release it. These steps are shown in Fig. 3.9.2.

```
typedef int semaphore;
semaphore resourece_1;

void process_A(void) {
down(&resource_1);
use_resource_1( );
up(&resource_1);
}
```

```
typedef int semaphore;
semaphore resourece_1;
semaphore resourece_2;

void process_A(void) {
down(&resource_1);
down(&resource_2);
use_both_resources( );
up(&resource_2);
up(&resource_1);
}
```

Fig. 3.9.2 : Using Semaphore to protect resources (a) one resource (b) Two Resource

Sometimes processes need two or more resources. They can be acquired sequentially,

- as shown in Fig. 3.9.2(b). If more than two resources are needed, they are just acquired one after another. So far, so good. As long as only one process is involved, everything works fine. Of course, with only one process, there is no need to formally acquire resources, since there is no competition for them.
- Now let us consider a situation with two processes, *A* and *B*, and two resources. Two scenarios are depicted in Fig. 3.9.3. In Fig. 3.9.3(a), both processes ask for the resources in the same order. In Fig. 3.9.3(b), they ask for them in a different order. This difference may seem minor, but it is not.

```
typedef int semaphore;
semaphore resourece_1;
semaphore resourece_2;

void process_A(void) {
down(&resource_1);
```

```
semaphore resourece_1;
semaphore resourece_2;

void process_A(void) {
down(&resource_1);
```

```
down(&resource_2);
use_both_resources( );
up(&resource_2);
up(&resource_1);
}

void process_B(void) {
down(&resource_1);
down(&resource_2);
use_both_resources( );
up(&resource_2);
up(&resource_1);
}
```

(a) Deadlock Free Code

```
down(&resource_2);
use_both_resources( );
up(&resource_2);
up(&resource_1);
}

void process_B(void) {
down(&resource_1);
down(&resource_2);
use_both_resources( );
up(&resource_2);
up(&resource_1);
}
```

(b) Code with potential deadlock

Fig. 3.9.3

- In Fig. 3.9.3(a), one of the processes will acquire the first resource before the other one. That process will then successfully acquire the second resource and do its work. If the other process attempts to acquire resource 1 before it has been released, the other process will simply block until it becomes available. In Fig. 3.9.3(b), the situation is different. It might happen that one of the processes acquires both resources and effectively blocks out the other process until it is done. However, it might also happen that process *A* acquires resource 1 and process *B* acquires resource 2. Each one will now block when trying to acquire the other one. Neither process will ever run again. Bad news: this situation is a deadlock.
- Here we see how what appears to be a minor difference in coding style which resource to acquire first turns out to make the difference between the program working and the program failing in a hard-to-detect way. Because deadlocks can occur so easily, a lot of research has gone into ways to deal with them. This chapter discusses deadlocks in detail and what can be done about them.

3.10 DEADLOCK CHARACTERIZATION

UQ. Explain the condition under which deadlock occurs.
(SPPU - Q. 4(a), Dec. 18, 4 Marks, Q. 3(b), Dec. 15, 6 Marks, Q. 4(a), May 18, 6 Marks)

UQ. What is mutual exclusion. ? **(SPPU - Dec. 18)**

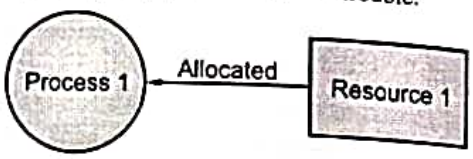
UQ. Explain the necessary and sufficient conditions for the occurrence of a deadlock.
(SPPU – Q. 6(b)OR, Aug. 17, 4 Marks)

Coffman, Elphick and Shoshani in 1971 have shown that there are **four conditions all of which must be satisfied for a deadlock to take place**. These conditions are given below.

1. Mutual Exclusion Condition
2. Wait for Condition
3. No Pre-emption Condition
4. Circular Wait Condition

1. Mutual Exclusion Condition

- Resources must be allocated to processes at any time in an exclusive manner and not on a shared basis for a deadlock to be possible. For instance, a disk drive can be shared by two processes simultaneously. This will not cause a deadlock. But printers, tape drives, plotters, etc. have to be allocated to a process in an exclusive manner until the process completely finishes its work with it (which normally happens when the process ends). This is the cause of the trouble.

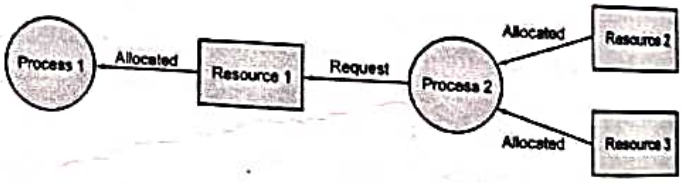


(1C12)**Fig. 3.10.1 : Mutual Exclusion Condition**

- At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait for Condition

- Even if a process holds certain resources at any moment, it should be possible for it to request for new ones. It should not have to give up the already held resources to be able to request for new ones. If this is not true, a deadlock can never take place.



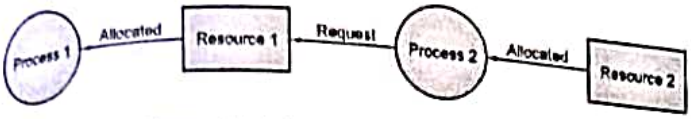
(1C13)**Fig. 3.10.2 : Hold and Wait Condition**

- A process can hold multiple resources and still request more resources from other processes which are holding them. In the Fig. 3.10.2 shown, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.

3. No Pre-emption Condition

- If a process holds certain resources, no other process should be able to take them away from it forcibly. Only the process

holding them should be able to release them explicitly, i.e. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

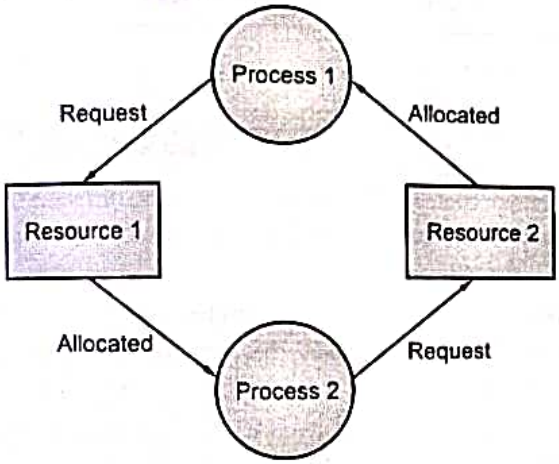


(1C14)**Fig. 3.10.3 : Pre-emption Condition**

- A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the Fig. 3.10.3, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.

4. Circular Wait Condition

- Processes (P1, P2, ...) and Resources (R1, R2, ...) should form a circular list as expressed in the form of a graph. In short, there must be a circular (logically, and not in terms of the shape) chain of multiple resources and multiple processes forming a closed loop.



(1C15)**Fig. 3.10.4 : Circular Wait condition**

- It is necessary to understand that all these four conditions have to be satisfied simultaneously for the existence of a deadlock. If any one of them does not exist, a deadlock can be avoided.
- The fourth condition is, actually, a potential consequence of the first three. That is, given that the first three conditions exist, a sequence of events may occur that lead to an unresolvable circular wait. The unresolvable circular wait is in fact the definition of deadlock. The circular wait listed as condition 4 is unresolvable because the first three conditions hold. Thus, the four conditions, taken together, constitute necessary and sufficient conditions for deadlock.

Table 3.10.1 : Possibility and Existence of Deadlock

Possibility Deadlock		Existence of Deadlock	
1.	Mutual Exclusion	1.	Mutual Exclusion
2.	No Pre-emption	2.	No Pre-emption
3.	Hold and Wait	3.	Hold and Wait
		4.	Circular Wait

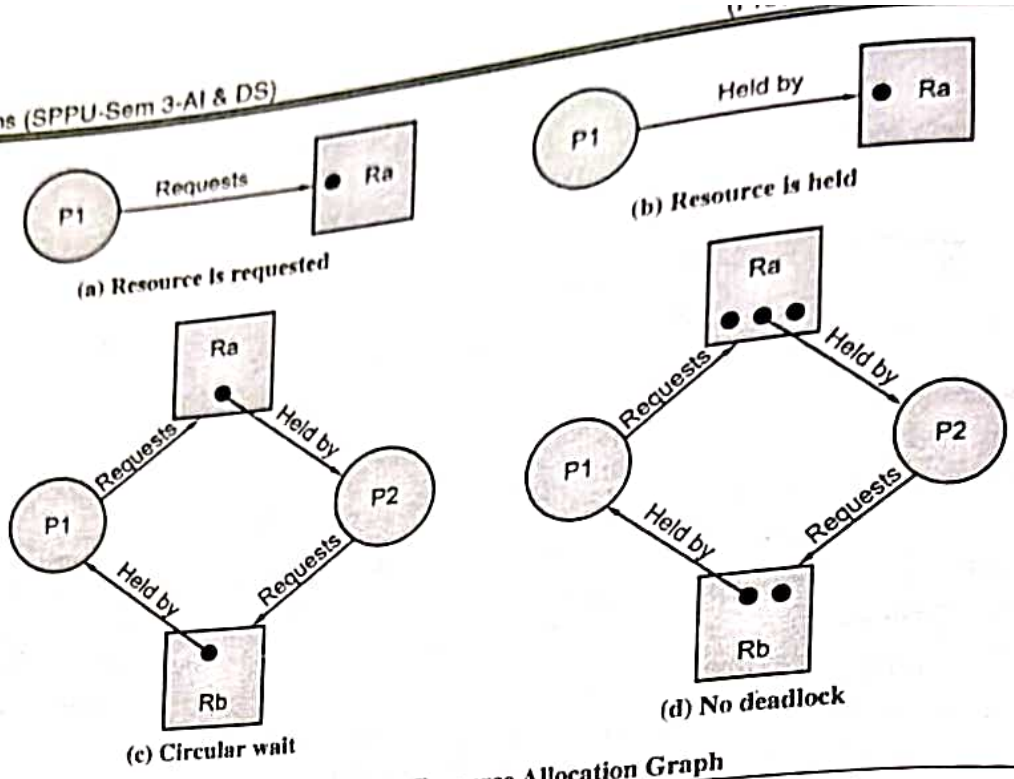
Three general approaches exist for dealing with deadlock.

1. One can **prevent** deadlock by adopting a policy that eliminates one of the conditions (conditions 1 through 4).
2. One can **avoid** deadlock by making the appropriate dynamic choices based on the current state of resource allocation.
3. One can attempt to **detect** the presence of deadlock (conditions 1 through 4 hold) and take action to recover.

3.11 RESOURCE ALLOCATION GRAPH

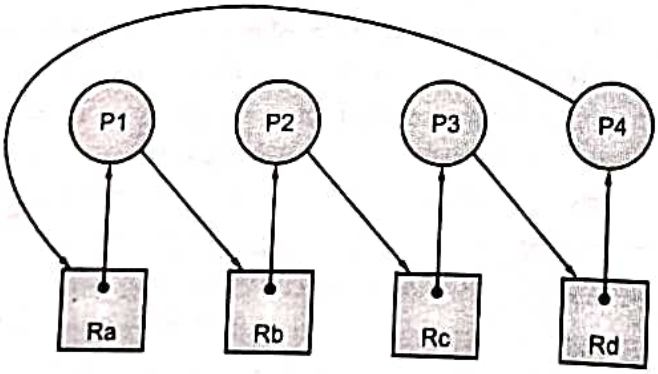
GQ. Write short note on : Resource allocation graph.
UQ. Write Short note on Resource Allocation Graph.
(SPPU - Q. 4(b), May 19, 3 Marks)

- The resource allocation graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node. It is useful tool in characterizing the allocation of resources to processes is the **resource allocation graph**.
- A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted (Fig. 3.11.1(a)).Within a resource node, a dot is shown for each instance of that resource. Examples of resource types that may have multiple instances are I/O devices that are allocated by a resource management module in the OS.
- A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted (Fig. 3.11.1(b)); that is, the process has been assigned one unit of that resource. A graph edge directed from a consumable resource node dot to a process indicates that the process is the producer of that resource.



(1C16)**Fig. 3.11.1 : Resource Allocation Graph**

- Fig. 3.11.1(c) shows an example deadlock. There is only one unit each of resources Ra and Rb. Process P1 holds Rb and requests Ra, while P2 holds Ra but requests Rb.
- Fig. 3.11.1(d) has the same topology as Fig. 3.11.1(c), but there is no deadlock because multiple units of each resource are available.
- The resource allocation graph of Fig. 3.11.2. corresponds to the deadlock situation in Fig. 3.11.2. Note that in this case, we do not have a simple situation in which two processes each have one resource the other needs. Rather, in this case, there is a circular chain of processes and resources that results in deadlock.



(1C17)**Fig. 3.11.2 : Resource Allocation graph**

3.12 DEADLOCK MODELING

UQ. Write short note on Deadlock Modelling.

SPPU - Q. 4(b), May 16, 8 Marks

Various strategies have been followed by different Operating Systems to deal with the problem of a deadlock. These are listed below.

1. **Ignore it :** We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. **Detect it and Recover from it :** We can allow the system to enter a deadlock state, detect it, and recover.
3. **Prevent it and Avoid it :** We can ignore the problem altogether, and pretend that deadlocks never occur in the system.
4. **Ignore it :** There are many approaches one can take to deal with deadlocks. One of them, and of course the simplest, is to ignore them. That is to pretend as if you are totally unaware of them. UNIX follows this approach on the assumption that most users would prefer an occasional deadlock to a very restrictive, inconvenient, complex and slow system.
5. **Detect it :** In reality, there could be a number of resource types such as printers, plotters, tapes, and so on. For instance, the system could have two identical printers, and the Operating System must be told about it at the time of system generation. It could well be that a specific process could do

with either of the printers when requested. The complexity arises due to the fact that allocation to a process is made of a specific resource by the Operating System, depending upon the availability but the request is normally made by the process to the Operating System for only a resource type (i.e. any resource belonging to that type).

3.13 DEADLOCK AVOIDANCE

UQ. Write short note on deadlock avoidance.

SPPU - Q. 4(a), May 16, 5 Marks

OR Explain deadlock avoidance method.

UQ. Explain in brief deadlock avoidance methods.

SPPU - Q. 3(a), May 19, 4 Marks

Deadlock avoidance requires that the system has some information available up front. Initially every process declares the maximum number of resources need which it may require.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. If the sequence in which process will request the resource and will release it known additionally, then we can decide for each request whether or not the process should wait.

To avoid the circular wait condition completely, a deadlock-avoidance algorithm inspects the resource-allocation state in dynamic way.

Following are the factors which define and describe resource allocation state.

- The number of available resources in the system.
- The number of allocated resources in the system.
- The maximum demands of the processes.

Dijkstra's Banker's algorithm is used for deadlock avoidance. The algorithm requires prior knowledge of number of resources each process needs. After that, the OS act like a sincere small-town banker. OS will allocate the resources to processes only it has sufficient number of resources available to fulfill possible demand. This is considered as a safe state.

3.13.1 Safe and Unsafe State

- If the system is able to allocate resources to each process up to its maximum need in some order and still avoid a deadlock then the system is in safe state.
- If the safe sequence of all processes present in system the system is considered to be in safe state :

Sequence <P1, P2, ... Pn > is considered as a safe sequence for the present allocation state if, for each process Px, the resources which N can still request can be fulfilled by,

- The at present available resources in the system plus
- The resources held by all of the processes Py's, where y < x.

- If process Pi cannot get all the needed resources as they are not available, it should wait until Pj complete the execution and frees the resources. If the system is in a safe state, there can be no deadlock. If the system is in an unsafe state, there is the possibility of deadlock.

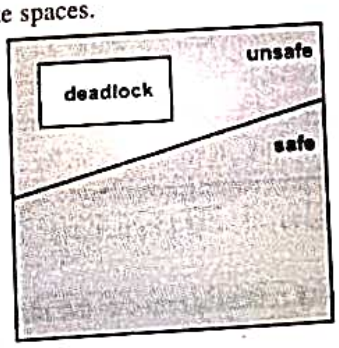
Example

- Consider a system with 10 disk drives and 3 processes (P0, P1, and P2).

Table 3.13.1 : Maximum and Current resource Requirement

Process	Maximum needs	Current needs
P0	8	4
P1	4	2
P2	7	2

- Initially 10 disk drives are available in the system. The sequence <P1, P0, P2> is safe sequence. Process P0 requires 8 disk drives, process P1 requires 4 disk drives and P2 may require up to 7 disk drives. At time t_1, process P0 is holding 4 disk drives, process P1s holding 2 disk drives and process P2 is holding 2 disk drives. (Thus in the system 2 disk drives are free.)
- At time to, the system is in safe state. The sequence <P1, P0, P2> satisfies safety condition. P1 can immediately be allocated its disk drives and returns them after completion. Now total 4 disk drives are available.
- P0 gets all the disk drives. After completion, it returns them and now total 8 disk drives are available in the system. Finally P2 can be allocated all the needed disk drives and returns them after completion. (The system will have now all 10 disk drives available). Fig. 3.13.1. shows safe, unsafe and deadlock state spaces.



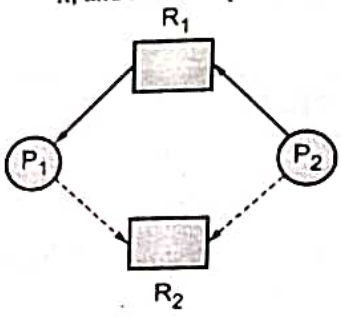
(1C18)**Fig. 3.13.1 : Safe, Unsafe and dead Lock**

- At time t if any process requests the additional resource say

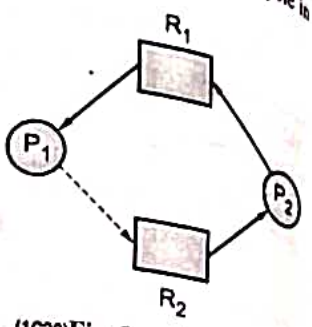
disk drives in our example and if after its allocation other process could not be allocated needed resource type due to unavailability, then system may go in unsafe state.

3.13.2 Dead Lock Avoidance Algorithm

- As seen already, most prevention algorithms have poor resource utilization, and hence result in reduced throughputs. Instead, we can try to avoid deadlocks by making use prior knowledge about the usage of resources by processes including resources available, resources allocated, future requests and future releases by processes.
- Most deadlock avoidance algorithms need every process to tell in advance the maximum number of resources of each type that it may need. Based on all these info we may decide if a process should wait for a resource or not, and thus avoid chances for circular wait.
- If a system is already in a safe state, we can try to stay away from an unsafe state and avoid deadlock. Deadlocks cannot be avoided in an unsafe state. A system can be considered to be in safe state if it is not in a state of deadlock and can allocate resources up to the maximum available. A safe sequence of processes and allocation of resources ensures a safe state. Deadlock avoidance algorithms try not to allocate resources to a process if it will make the system in an unsafe state. Since resource allocation is not done right away in some cases, deadlock avoidance algorithms also suffer from low resource utilization problem.
- **Following algorithms are used in the avoidance of deadlock :**
  1. Resource-Allocation graph Algorithm
  2. Banker's Algorithm
  3. Resource-Request Algorithm
  4. Safety Algorithm
- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with *claim edges*, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi -> Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge. This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P_2 requests resource R_2.
- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.



(1C19)**Fig. 3.13.2 : Resource allocation graph for deadlock avoidance**



(1C20)**Fig. 3.13.3 : An unsafe state in a resource allocation graph**

3.14 BANKERS ALGORITHM

UQ. Banker's algorithm is used for Deadlock avoidance. Explain. **(SPPU - Q. 4(b), Dec. 15, 6 Marks)**

UQ. What is Bankers algorithm ? Explain with suitable examples. **(SPPU - Q. 3(c), May 17, 6 Marks)**

- It is applicable to the resource allocation system with multiple instances of each resource type. It is less efficient than resource-allocation graph algorithm.
- A newly entered process should declare maximum number of instances of each resource type which it may require. The request should not be more than total number of resources in the system. System checks if allocation of requested resources will leave the system in safe state. If it will the requested resources are allocated. if system determines that resources cannot be allocated as it will go in unsafe state, the requesting process should wait until other process free the resources. Following **data structures** are used in Banker's algorithm :
  - **A[m]:** Array A of size m shows the number of available resources.
  - **M[m][n]:** Two dimensional array M shows maximum requirement of the resources by each process. M[i][j] = k indicates process N can request at the most k instances of resource type Rj.
  - **C[n][m]:** Two dimensional array C shows current allocation status of resources to each process. C[i][j] = k indicates process N allocated k instances of resource type Rj.
  - **N[n][m]:** Two dimensional array N shows the remaining possible need of each process.

(i.e., N[n][m] = M[n][m] - C[n][m]). N [n][m] = k indicates process N may need additional k instances of resource type R_j to accomplish the execution.

3.14.1 Resource Request Algorithm

Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself. This algorithm determines if a new request is safe, and grants it only if it is safe to do so.

When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows :

1. Let **Request[n] [m]** indicates the number of resources of each type currently requested by processes.

   If **Request[i] > Need[i] for any process i**, raise an error condition.

2. If **Request[i] > Available for any process i**, then that process must wait for resources to become available. Otherwise the process can continue to step 3.

3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.

   The procedure for granting a request (or pretending to for testing purposes) is:

   Available = Available – Request
   Allocation = Allocation + Request
   Need = Need – Request

- Once the resources are allocated, check to see if the system state is safe. if unsafe, the process must wait and the old resource-allocated state is restored.

3.14.2 Safety Algorithm

In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe. This algorithm determines if the current state of a system is safe, according to the following steps :

1. Let Work and Finish be vectors of length m and n respectively.
   - Work is a working copy of the available resources, which will be modified during the analysis.
   - Finish is a vector of Booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
   - Initialize Work to Available, and Finish to false for all elements.
2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

JTB's Modification

- In step 1. instead of making Finish an array of Booleans initialized to false, make it an array of integer initialized to 0. Also initialize an int s = 0 as a step counter.
- In step 2, look for Finish[i] == 0.
- In step 3, set Finish[i] to ++s. S is counting the number of finished processes.
- For step 4, the test can be either Finish[i] > 0 for all i, or s >= n. The benefit of this method is that if a safe state exists, then Finish[ ] indicates one safe sequence (of possibly many).

3.15 SOLVED PROBLEMS

UEx. 3.15.1 **SPPU - Q. 4(c), Dec. 16, 6 Marks**

Find out the safe sequence for execution of 4 processes using Bankers algorithm. Maximum Resources : R1 = 5, R2 = 5.

Allocation Matrix			Maximum Requirement Matrix		
	R1	R2		R1	R2
P1	1	0	P1	1	1
P2	1	1	P2	2	3
P3	1	2	P3	2	2
P4	1	1	P4	3	2

Soln. :

Given :

Maximum resources R1 = 5, R2 = 5

By adding the columns of allocation matrix

We get,

R1 = 4 and R2 = 4

Available resources (R1) = 5 – 4 = 1

Available resource (R2) = 5 – 4 = 1

Need = Maximum – Allocation

	R1	R2
P1	0	1
P2	1	2
P3	1	0
P4	2	1

Work = Work + Allocation

Finish(i)	Need (i)	Work	Need ≤ work	Safe sequence	Work	New finish
False	{0,1}	{1,1}	Yes	{P1}	{2,1}	True
False	{1,2}	{2,1}	Yes	{P1, P2}	{3,2}	True
False	{1,0}	{3,2}	Yes	{P1,P2,P3}	{4,4}	True
False	{2,1}	{4,4}	Yes	{P1,P2,P3,P4}	{5,5}	True

The safe sequence is {P1, P2, P3, P4}

UEx. 3.15.2 SPPU - Q. 3(c), Dec. 18, 6 Marks

Find out the safe sequence for the execution of the following processes using bankers algorithm.

Maximum resources R1 = 15, R2 = 8.

	Allocation Matrix			Maximum Required		
	R1	R2			R1	R2
P1	2	1		P1	5	6
P2	3	2		P2	8	5
P3	3	0		P3	4	8

☑ **Soln. :**

Maximum Required

	R1	R2		R1	R2
P1	2	1	P1	5	6
P2	3	2	P2	8	5
P3	3	0	P3	4	8

Given : Resources R1 = 15, R2 = 8

By adding the columns of allocation matrix, we get

Available resources (R1) = 15 – 8 = 7

Available resources (R2) = 8 – 3 = 5

Need = maximum – allocation

	R1	R2
P1	3	5
P2	5	3
P3	1	8

Finish (i)	Need (i)	Work	Need ≤ work	Safe sequence	Work	New finish
False	{3, 5}	{7,5}	Yes	{P1}	{9,6}	True
False	{5,3}	{9,6}	Yes	{P1,P2}	{12,8}	True
False	{1,8}	{12,8}	Yes	{P1,P2,P3}	{15,8}	True

The safe sequence is {P1, P2, P3}

UEx. 3.15.3 SPPU - Q. 4(c), Dec. 18, 6 Marks

Find out the safe sequence for the execution of the following processes using bankers algorithm.

Maximum resources R1 = 4, R2 = 4

	Allocation Matrix			Maximum Required		
	R1	R2			R1	R2
P1	1	0		P1	1	1
P3	1	2		P3	2	2

☑ **Soln. :**

Allocation Matrix **Maximum Required**

	R1	R2		R1	R2
P1	1	0	P1	1	1
P2	1	1	P2	2	3
P3	1	2	P3	2	2

Maximum resources = R1 = 4, R2 = 4

By adding the columns of allocation matrix

We get,

R1 = 2 and R2 = 2

Available resources (R1) = 4 – 2 = 2

Available resources (R2) = 4 – 2 = 2

Need = Maximum – Allocation

	R1	R2
P1	0	1
P2	1	0

Finish (i)	Need (i)	Work	Need ≤ work	Safe sequence	Work	New finish
False	{0,1}	{2,2}	Yes	{P1}	{3,2}	True
False	{1,0}	{3,2}	Yes	{P1,P2}	{4,4}	True

The safe sequence is {P1, P2}

UEx. 3.15.4 SPPU - Q. 3(c), Dec. 18, 6 Marks

Find out the safe sequence for the execution of the following processes using bankers algorithm.

Maximum resources R1 = 4, R2 = 4

Allocation Matrix		
	R1	R2
P1	1	0
P2	1	1
P3	1	2

Maximum Required		
	R1	R2
P1	1	1
P2	2	3
P3	2	2

☑ Soln. :

Given

Allocation Matrix			Maximum Required		
	R1	R2		R1	R2
P1	1	0	P1	1	1
P2	1	1	P2	2	3
P3	1	2	P3	2	2

Maximum resources = R1 = 4, R2 = 4

By adding the columns of allocation matrix,

We get,

R1 = 3 and R2 = 3

Available resources (R1) = 4 – 3 = 1

Available resources (R2) = 4 – 3 = 1

Need = Maximum – Allocation

	R1	R2
P1	0	1
P2	1	2
P3	1	0

Finish (i)	Need (i)	Work	Need ≤ work	Safe sequence	Work	New finish
False	{0,1}	{1,1}	Yes	{P1}	{2,1}	True
False	{1,2}	{2,1}	Yes	{P1,P2}	{3,2}	True
False	{1,0}	{3,2}	Yes	{P1,P2,P3}	{4,4}	True

The safe sequence is {P1, P2, P3}

UEx. 3.15.5 SPPU - Q. 3(c), May 18, 6 Marks

Find out the safe sequence for the execution of the following processes using bankers algorithm

Maximum resources R1 = 13, R2 = 7, R3 = 10 units

	R1	R2	R3
P1	2	1	1
P2	7	2	3
P3	3	2	2
P4	1	1	3

	R1	R2	R3
P1	4	3	3
P2	7	2	4
P3	4	2	5
P4	5	3	3

☑ Soln. :

Allocation matrix				Maximum required		
	R1	R2	R3	R1	R2	R3
P1	2	1	1	4	3	3
P2	7	2	3	7	2	4
P3	3	2	2	4	2	5
P4	1	1	3	5	3	3

Maximum resources = R1 = 13, R2 = 7, R3 = 10

By adding the columns of allocation matrix,

We get,

R1 = 13 and R2 = 6 and R3 = 9

Available resources (R1) = 13 – 13 = 0

Available resources (R2) = 7 – 6 = 1

Available resources (R3) = 10 – 9 = 1

Need = Maximum – Allocation

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

Finish (i)	Need (i)	Work	Need ≤ work	Safe sequence	Work	New finish
False	{2,2,2}	{0,1,1}	No	-	{0,1,1}	False
False	{0,0,1}	{0,1,1}	Yes	{P2}	{7,3,4}	True
False	{1,0,3}	{7,3,4}	Yes	{P2,P3}	{10,5,6}	True
False	{4, 2, 0}	{10, 5, 6}	Yes	{P2, P3, P4}	{11, 6, 9}	True

The safe sequence is {P3, P3, P4, P1}

UEx. 3.15.6 SPPU - Q. 3(a), May 16, 6 Marks

Find out the safe sequence for execution of 3 processes using Bankers algorithm maximum resources : R1 = 7, R2 = 7, R3 = 10.

	Allocation matrix				Maximum requirement matrix		
	R1	R2	R3		R1	R2	R3
P1	2	2	3	P_1	3	6	8
P2	2	0	3	P_2	4	3	3
P3	1	2	4	P_3	3	4	4

☑ **Soln. :**

	Allocation matrix				Maximum requirement matrix		
	R1	R2	R3		R1	R2	R3
P1	2	2	3	P1	3	6	8
P2	2	0	3	P2	4	3	3
P3	1	2	4	P3	3	4	4

Maximum resources = R1 = 7, R2 = 7, R3 = 10

By adding the columns of allocation matrix,

We get,

R1 = 5 and R2 = 4 and R3 = 10

Available resources (R1) = 2

Available resources (R2) = 3

Available resources (R3) = 0

Need = Maximum – Allocation

	R1	R2	R3
P1	1	4	5
P2	2	3	0
P3	2	2	0

Finish (i)	Need (i)	Work	Need ≤ work	Safe sequence	Work	New finish
False	{1,4,5}	{2,3,0}	No	-	{2,3,0}	False
False	{2,3,0}	{2,3,0}	Yes	{P2}	{4,3,3}	True
False	{2,2,0}	{4,3,3}	Yes	{P2,P3}	{5,5,7}	True

The safe sequence is {P2, P3, P1}

UEx. 3.15.7 (SPPU - Q. 5(a), Oct. 19, Q. 5(a), Aug. 17, 6 Marks)

Consider the following snapshot of a system :

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Answer the following questions using bankers algorithm.

(i) Find need matrix. (ii) Is the system in a safe state ?

(iii) If a request from process P_1 arrives for (0, 4, 2, 0), can the request be granted immediately.

☑ **Soln. :**

Need = Max – Allocation

▶ **Step 1 : Need matrix is,**

	N(Need)			
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

Step 2 : To decide whether system is in safe state or not

Process P0

Need <= Availability i.e. (0,0,0,0) <= (1,5,2,0) is True, Process P0 is executed

New Availability = Availability + Allocation = (1,5,2,0) + (0,0,1,2) = (1,5,3,2)

Process P1

Need <= Availability i.e. (0,7,5,0) <= (1,5,3,2) is False, Resource B and C , need is more than Availability.

P1 is not Executed

Process P2

Need <= Availability i.e. (1,0,0,2) <= (1,5,3,2) is True, Process P2 is executed.

New Availability = Availability + Allocation = (1,5,3,2) + (1,3,5,4) = (2,8,8,6)

Process P3

Need <= Availability i.e. (0,0,2,0) <= (2,8,8,6) is True, Process P3 is executed.

New Availability = Availability + Allocation = (2,8,8,6) + (0,6,3,2) = (2,14,11,8)

Process P4

Need <= Availability i.e. (0,6,4,2) <= (2,14,11,8) is True, Process P4 is executed.

New Availability = Availability + Allocation = (2,14,11,8) + (0,0,1,4) = (2,14,12,12)

Process P1

Need <= Availability i.e. (0,7,5,0) <= (2,14,12,12) is True, Process P1 is executed.

New Availability = Availability + Allocation = (2,14,12,12) + (1,0,0,0) = (3,14,12,12)

System is in safe state because safe sequences exist such as (P0; P2; P3; P4; P1).

Step 3 : If a request from process P_1 arrives for (0, 4, 2, 0), can the request be granted immediately?

i. Check for the following Condition – Request is of (1,0,2) resources

Request <= need i.e. (0,4,2,0) <= (0,7,5,0) is True , need of Process P1 from snapshot is (0,4,2,0)

Request <= availability i.e. (0,4,2,0)<=(1,5,2,0) is True , Availability is (1,5,2,0)

ii. Update the Snapshot with new Availability , Need and Allocation

Available = Available – Request = (3,3,2) - (1,0,2) = (2,3,0)

Allocation = Allocation + request = (2,0,0) + (1,0,2) = (3,0,2)

Need = Need – Request = (1,2,2) + (1, 0,2) = (0,2,0)

iii. Updated SnapShot is

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	1	0	0	0	0	0	0
P_1	1	4	2	0	1	7	5	0					1	3	3	0
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	6	4	2

iv. Check for Safe State by applying safety Algorithm

Process P0

Need <= Availability i.e. (0,0,0,0) <= (1,1,0,0) is True , Process P0 is executed.
New Availability = Availability + Allocation = (1,1,0,0) + (0,0,1,2) = (1,1,1,2)

Process P1

Need <= Availability i.e. (1,3,3,0) <= (1,1,1,2) is False. Resource needed are more than available.
Process P1 is not executed.

Process P2

Need <= Availability i.e. (1,0,0,2) <= (1,1,1,2) is True, P2 is executed.
New Availability = Availability + Allocation = (1,1,1,2) + (1,5,3,4) = (2,4,6,6)

Process P3

Need <= Availability i.e. (0,0,2,0) <= (2,4,6,6) is True., Process P3 is executed.
New Availability = Availability + Allocation = (2,4,6,6) + (0,6,3,2) = (2,10,9,8)

Process P4

Need <= Availability i.e. (0,6,4,2) <= (2,10,9,8) is True, Process P4 is executed.
New Availability = Availability + Allocation = (2,10,9,8) + (0,0,1,4,) = (2,10,10,12)

- At this point we can check for Process P1, There can be many safe sequence depending on which process is considered next. Here Process P1 considered.

Process P1

Need <= Availability i.e. (1,3,3,0) <= (2,10,10,12) is True, Process P1 is executed.
New Availability = Availability + Allocation = (2,10,10,12) + (1,4,2,0) = (3,14,12,12)

- System is in safe state because safe request exist and safe sequence is P0,P2,P3,P4,P1.Hence request from process P1 arrives for (0,4,2,0) can the request be granted immediately.,because request <= available i.e. request is (0,4,2,0) <= Available(1,1,0,0)

Ex. 3.15.8 : Consider following snapshot of the system Using Banker's algorithm answer the following questions :

(i) What are the content of need matrix ?

(ii) Find if system is in safe state? If it is, find the safe sequence.

(iii) If request P1 arrives for (1,1,0,0) resources, can required resource be granted immediately.

(iv) If request P4 arrives for (0,0,2,0) resources, can required resource be granted immediately.

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	2	0	0	1	4	2	1	2	3	3	2	1
P_1	3	1	2	1	5	2	5	2				
P_2	2	1	0	3	2	3	1	6				
P_3	1	3	1	2	1	4	2	4				
P_4	1	4	3	2	3	6	6	5				

☑ **Soln. :**

Need = Max – Allocation

▶ **Step 1 : Need Matrix is**

	Need			
	A	B	C	D
P0	2	2	1	1
P1	2	1	3	1
P2	0	2	1	3
P3	0	1	1	2
P4	2	2	3	3

Process P0

Need <= Availability i.e. (2,2,1,1) <= (3,3,0,1) is False, Resource C, Need is more and availability is less.

Process P0 is not executed.

Process P1

Need <= Availability i.e. (3,12,1) <= (3,3,0,1) is False, Resource C, Need is more and availability is less.

Process P1 is not executed.

Process P2

Need <= Availability i.e. (2,1,0,3) <= (3,3,0,1) is False, Resource D, Need is more and availability is less.

P2 is not executed.

Process P3

Need <= Availability i.e. (1,3,1,2) <= (3,3,0,1) is False, Resource C and D, Need is more and availability is less.

P3 is not executed.

Process P4

Need <= Availability i.e. (1,4,5,2) <= (3,3,0,1) is False, Resource C and D, Need is more and availability is less.

P4 is not executed.

- **Conclusion :** System is not in safe state. Hence request from process P4 arrives for (0, 0, 2,0) cannot be granted immediately.

3.16 DEADLOCK DETECTION AND RECOVERY

GQ. Write short note on Deadlock detection.

OR Explain prevention and detection.

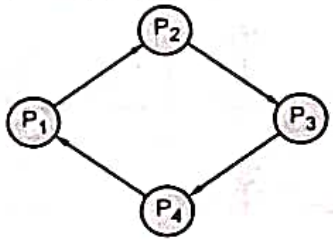
UQ. Explain with an appropriate example how resource allocation graph determines a deadlock.

(SPPU - Q. 4(b), May 18, Dec. 18, Dec. 19, 5 Marks)

UQ. How resource allocation graph helps to deadlock ? Write the necessary conditions of deadlock to be occurred. (SPPU - Q. 6(b), Oct.19, 4 Marks)

3.16.1 Dead Lock Detection

- Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists :
  - Temporarily prevent resources from deadlocked processes.
  - Back off a process to some check point allowing pre-emption of a needed resource and restarting the process at the checkpoint later.
  - Successively kill processes until the system is deadlock free.
- If deadlock prevention and avoidance are not done properly, as deadlock may occur and only things left to do is to detect the recover from the deadlock.
- If all resource types has only single instance, then we can use a graph called wait-for-graph, which is a variant of resource allocation graph. Here, vertices represent processes and a directed edge from P1 to P2 indicate that P1 is waiting for a resource held by P2. Like in the case of resource allocation graph, a cycle in a wait-for-graph indicate a deadlock. So the system can maintain a wait-for-graph and check for cycles periodically to detect any deadlocks.

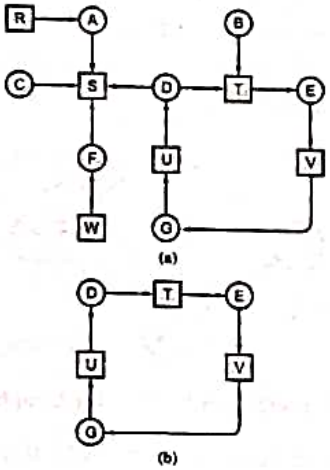


(1C21)Fig. 3.16.1 : Wait for Graph

- The wait-for-graph is not much useful if there are multiple instances for a resource, as a cycle may not imply a deadlock. In such a case, we can use an algorithm similar to Banker's algorithm to detect deadlock. We can see if further allocations can be made on not based on current allocations.

(a) Deadlock Detection with One Resource of Each Type

- Lets there is only one resource of each type. Such a system might have one scanner, one Blu-ray recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment.
- For such a system, we can construct a resource graph of the sort illustrated in Fig. 3.16.2.
  - If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked.
  - If no cycles exist, the system is not deadlocked.
  - As an example consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows :
  - Process *A* holds *R* and wants *S*.
  - Process *B* holds nothing but wants *T*.
  - Process *C* holds nothing but wants *S*.
  - Process *D* holds *U* and wants *S* and *T*.
  - Process *E* holds *T* and wants *V*.
  - Process *F* holds *W* and wants *S*.
  - Process *G* holds *V* and wants *U*.
- The resource graph of Fig. 3.16.2(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. 3.16.2.(b). From this cycle, one can see that processes *D*, *E*, and *G* are all deadlocked. Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.



(1C22)**Fig. 3.16.2 : A Resource Graph (b) A cycle extracted from (a)**

- Although it is relatively simple to pick out the deadlocked processes by visual inspection from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known. Given a simple one that inspects a graph nd terminates either when it has found a cycle or when it has shown that none exists. It uses one dynamic data structure, *L*, a list of nodes, as well as a list of arcs. During the algorithm, to prevent repeated inspections, arcs will be marked to indicate that they have already been inspected. The algorithm operates by carrying out the following steps as specified :
  - For each node, *N*, in the graph, performs the following five steps with N as the starting node.
  - Initialize L to the empty list, and designate all the arcs as unmarked.
  - Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
  - From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
  - Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
  - If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

(b) Deadlock Detection with Multiple Resources of Each Type

- A different approach is needed when multiple copies of some of the resources exist to detect deadlocks.
- Here is matrix-based algorithm for detecting deadlock among *n* processes, *P*1 through *Pn*. Let the number of resource classes be *m*,
  - with E1 resources of class 1,
  - E2 resources of class 2, and generally,
  - Ei resources of class i ≤ i ≤ m).
  - E is the existing resource vector.
- It gives the total number of instances of each resource in existence. For example, if class 1 is tape drives, then *E*1=2 means the system has two tape drives. At any instant, some of the resources are assigned and are not available. Let *A* be the **available resource vector**, with *Ai* giving the number of instances of resource *i* that are currently available (i.e., unassigned). If both of our two tape drives are assigned, *A*1 will be 0.
- Now we need two arrays, *C*, the **current allocation matrix**, and *R*, the **request matrix**. The i^{th} row of *C* tells how many instances of each resource class *Pi* currently holds. Thus, *Cij* is the number of instances of resource *j* that are held by process *i*. Similarly, R_{ij} is the number of instances of resource *j* that *Pi* wants.

- These four data structures are shown in Fig. 3.16.3. An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

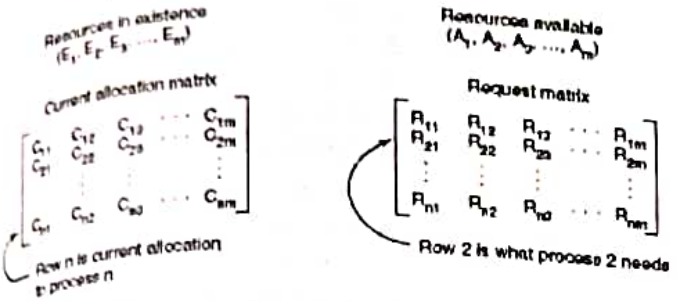
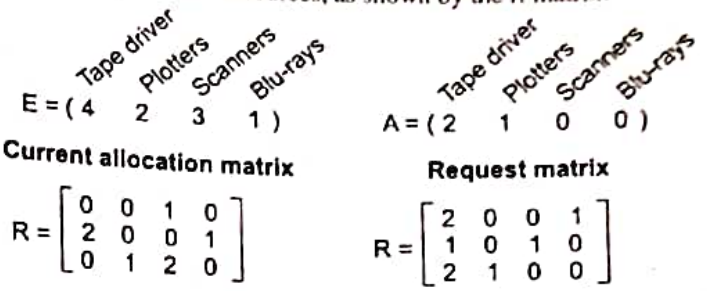


Fig. 3.16.3 : The four Data used by the deadlock algorithm

- In other words, if we add up all the instances of the resource j that have been allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist.
- The deadlock detection algorithm is based on comparing vectors.
  - Let us define the relation $A \leq B$ on two vectors A and B to mean that each element of A is less than or equal to the corresponding element of B.
  - Mathematically, $A \leq B$ holds if and only if $Ai \leq Bi$ for $1 \leq i \leq m$.
  - Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked.
- This algorithm assumes a worst-case scenario: all processes keep all acquired resources until they exit. The deadlock detection algorithm can now be given as follows.
  - Look for an unmarked process, *Pi*, for which the i^{th} row of R is less than or equal to A.
  - If such a process is found, add the i^{th} row of C to A, mark the process, and go back to step 1.
  - If no such process exists, the algorithm terminates.
- When the algorithm finishes, all the unmarked processes, if any, are deadlocked.
- What the algorithm is doing in step 1 is looking for a process that can be run to completion. Such a process is characterized as having resource demands that can be met by the currently available resources. The selected process is then run until it finishes, at which time it returns the resources it is holding to the pool of available resources. It is then marked as completed. If all the processes are ultimately able to run to completion, none of them are deadlocked. If some of them can never finish, they are deadlocked. Although the algorithm is nondeterministic (because it may run the processes in any feasible order), the result is always the same.
- As an example of how the deadlock detection algorithm works, see Fig. 3.16.4. Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanners, and Blu-ray drives. Process 1 has one scanner. Process 2 has two tape drives and a Blu-ray drive. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the *R* matrix.



(1C23)Fig. 3.16.4 : Dead Detection Algorithm

- To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied. The first one cannot be satisfied because there is no Blu-ray drive available. The second cannot be satisfied either, because there is no scanner free. Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving A = (2 2 2 0)
- At this point process 2 can run and return its resources, giving A = (4 2 2 1).
- Now the remaining process can run. There is no deadlock in the system.

☞ Recovery from Dead Lock

- It needs to recover from deadlock when it is detected. Several options exist after detection algorithm detects that a deadlock exists in the system. One possibility is to inform operator and let them decide how to deal with it manually.
- The second option is to allow the system to recover from the deadlock on its own (automatically). There are two solutions exist to break a deadlock. One solution is just to abort one or more processes so that circular wait will be broken. The second option includes preemption of some of the resources from one or more of the processes which are involved in deadlock.

a) Process Termination (Kill a process)

- When deadlock occurs, the operating system decides to kill one or more processes. This will reclaim the resources acquired by that killed processes. The deadlock detection algorithm detects whether deadlock exist or not after killing the process.
- Following two methods are used for killing the process :

1. Kill all deadlocked processes
2. Kill one process at a time

1. **Kill all deadlocked processes :** This method is simple and effective to eliminate the deadlock and clearly will break the deadlock cycle, but at a huge cost. If all these killed processes have been performed the computation for longer period of time then the partially computed result would be wasted. Recomputation will be done and is very costly.
2. **Kill one process at a time :** In this method, one process is killed at a time and detection algorithm is invoked to check whether deadlock is still exist or not in the system. Invocation of the detection algorithm after killing each process is considerable overhead. We must re-nm algorithm after each kill. Killing a process is not easy. We should kill only those processes whose killing will incur minimum cost.

Different parameter that needs to be considered to ensure the minimum cost are :

- Priority of the process How much computation process has finished and how much more time does it need to finish the execution.
- The number of the resources process has used.
- The type of resources the process has used.
- The number of the resources the process will need to complete the execution.
- The number of processes needs to be killed.
- Whether the process is interactive or batch.

(b) Resource Pre-emption

Incrementally pre-empt the resources from the process and reallocate resources to other processes until the circular wait is broken.

1. Selecting a victim	2. Rollback
3. Starvation	

1. **Selecting a victim :** Which process and which resources should be selected for pre-emption to minimize the cost? The process which has finished almost all computation and only few amount of computation is remained should not be selected as a victim.
2. **Rollback :** After pre-empting the resources from the particular process, it cannot continue the execution as needed resources are pre-empted. It is required to rollback the process to safe state and restarts the execution from that state. As safe state is difficult to define, total rollback will be the simple solution.
3. **Starvation :** It may happen that, same process will be selected many times to pre-empt the resources.

(c) Recovery through Rollback

- If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes checkpointed periodically. Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, in other words, which resources are currently assigned to the process.
- To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence accumulates.
- When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting at one of its earlier checkpoints.
- All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again). In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

3.17 ADVANTAGES AND DISADVANTAGES OF DEADLOCK METHODS

Advantages

1. This situation works well for processes which perform a single burst of activity.
2. No pre-emption needed for Deadlock.
3. Convenient method when applied to resources whose state can be saved and restored easily.
4. Feasible to enforce via compile-time checks.
5. Needs no run-time computation since the problem is solved in system design.

Disadvantages

1. Delays process initiation
2. Processes must know future resource need.
3. Pre-empts more often than necessary.
4. Dis-allows incremental resource requests.
5. Inherent pre-emption losses.

3.18 DIFFERENCE BETWEEN DEADLOCK AVOIDANCE AND PREVENTION

GQ. What is the difference between deadlock avoidance and prevention ?

Sr. No.	Factors	Deadlock prevention	Deadlock avoidance
1.	Concept	It blocks at least one of the conditions necessary for deadlock to occur.	It ensures that system does not go in unsafe state.
2.	Future resource requests	It doesn't require knowledge of future process resource requests.	It requires knowledge of future process resource requests.
3.	Procedure	It prevents deadlock by constraining resource request process and handling of resources.	It automatically considers requests and check whether it is safe for system or not.
4.	Information required	It does not require information about existing resources, available resources and resource requests.	It requires information about existing resources, available resources and resource requests.
5.	Pre-emption	Sometimes, pre-emption occurs more frequently.	In deadlock avoidance there is no pre-emption.
6.	Resource Request	All the resources are requested together.	Resource requests are done according to the available safe path.
7.	Resource allocation strategy	Resource allocation strategy for deadlock prevention is conservative.	Resource allocation strategy for deadlock prevention is not conservative.
8.	Advantage	It doesn't have any cost involved because it has to just make one of the conditions false so that deadlock doesn't occur.	There is no system under-utilization as this method works dynamically to allocate the resources.
9.	Disadvantage	Deadlock prevention has low device utilization.	Deadlock avoidance can block processes for too long.
10.	Example	Spooling and non-blocking synchronization algorithms are used.	Banker's and safety algorithm is used.

...Chapter Ends

□□□