UNIT – 3 Deadlock and process Synchronization

3.1 What is a Deadlock?

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 waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

In deadlock, there is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. For example, when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other.

A classic case of traffic deadlock on four one-way streets. This is "gridlock," where no vehicles can move forward to clear the traffic jam.

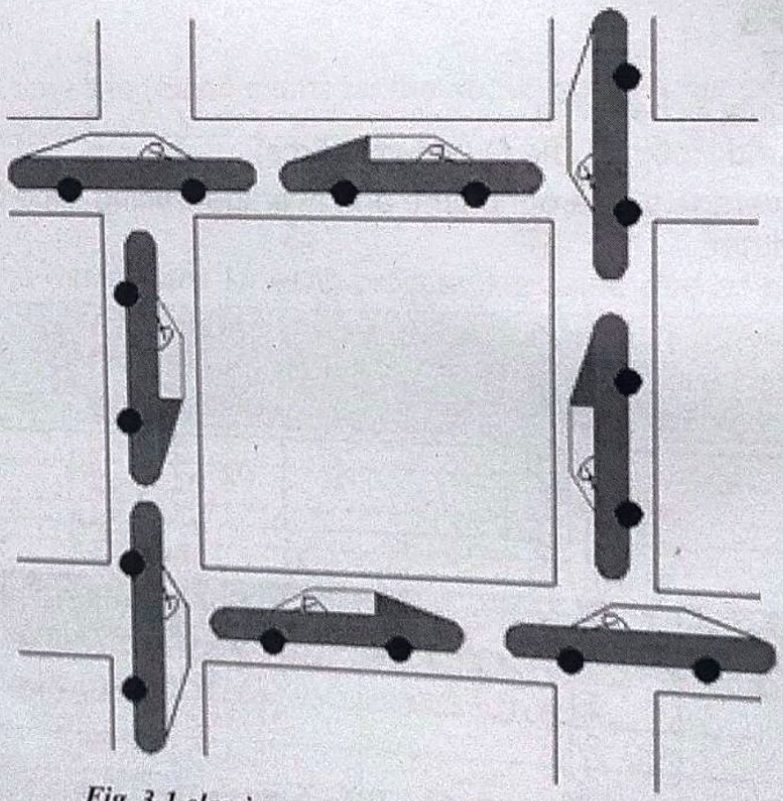


Fig. 3.1 classic case of traffic deadlock (gridlock)

Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below

diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

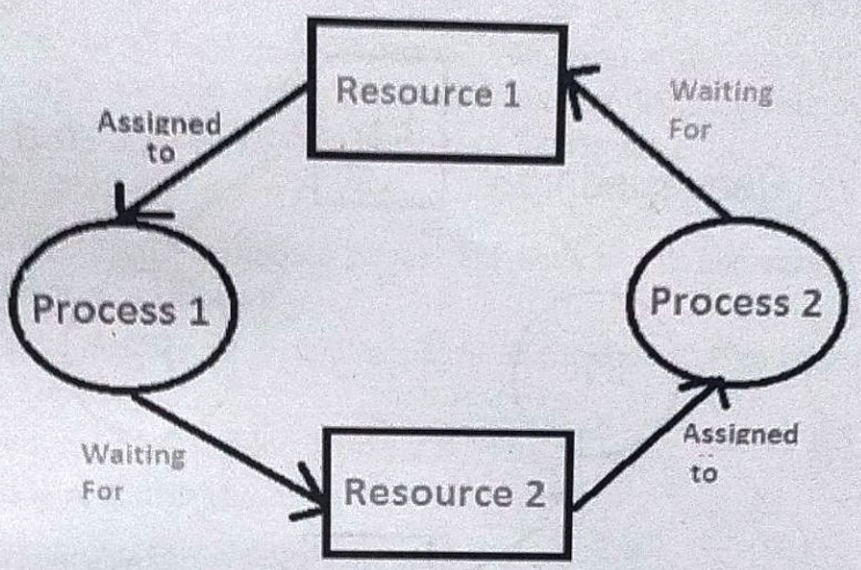


Fig. 3.2 classic case of operating system

3.2 Seven cases for deadlock

A deadlock usually occurs when non-sharable, non-preemptable resources, such as files, printers, or scanners, sharable resources that are locked, such as disks and databases are allocated to jobs that eventually require other non-sharable, non-preemptive resources that have been locked by other jobs.

The seven cases for deadlock are listed below:

1. Deadlocks on file requests
2. Deadlocks in databases
3. Deadlocks in dedicated device allocation
4. Deadlocks in multiple device allocation
5. Deadlocks in spooling
6. Deadlocks in a network
7. Deadlocks in disk sharing

3.2.1 Case 1: Deadlocks on File Requests

→ If processes are allowed to request and hold files for the duration of their execution, deadlocks can occur.

→ Here, two processes – P1 and P2, shown as circles, are each waiting for a resource (files), shown as rectangles – F1 and F2, that has already been allocated to the other process, thus creating a deadlock.

→ Any other programs that require F1 or F2 are put on hold as long as this situation continues.

→ Deadlock remains until a programs is withdrawn or forcibly removed and its file is released.

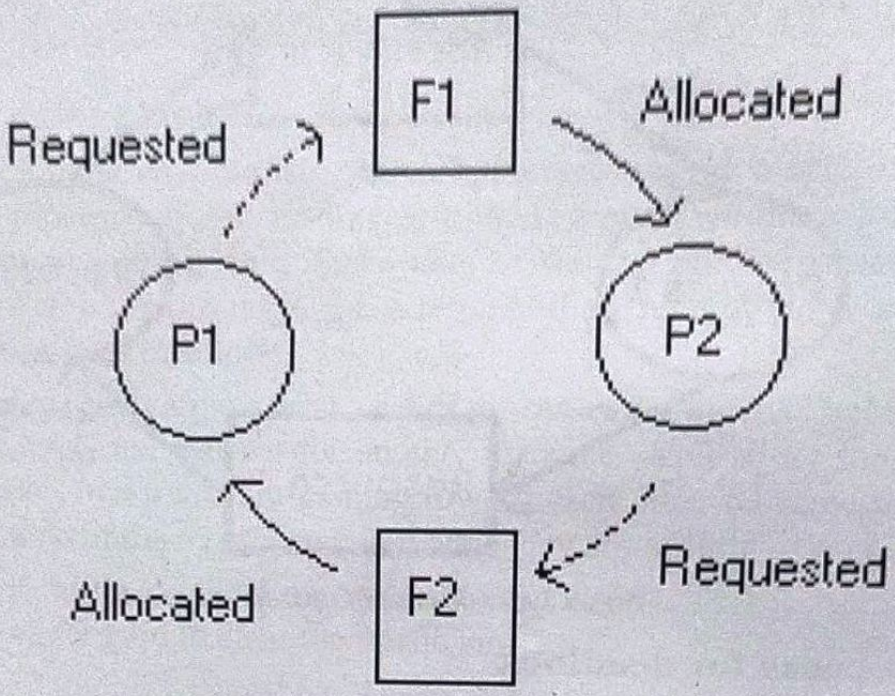


Fig. 3.3 Deadlocks on File Requests

Example:

- P1 requires raw data from R1.dat file and sends results to R2.dat file
- P2 requires raw data from R2.dat file and sends results to R1.dat file
- P1 and P2 have to work simultaneously

3.2.2 Case 2: Deadlocks in Databases

→ Deadlock can occur if two processes access and lock records in a database.

→ Three different levels of locking:

— The entire database for the duration of the request: prevents a deadlock from occurring but it restricts access to the database to one user at a time and, in a multiuser environment, response times are significantly slowed

— A subsection of the database: access time is improved but the possibility of a deadlock is increased because different processes sometimes need to work with several parts of the database at the same time

— Only the individual record until the process is completed: There are two processes (P1 and P2), each of which needs to update two records (R1 and R2), and the following sequence leads to a deadlock:

1. P1 accesses R1 and locks it.
2. P2 accesses R2 and locks it.

3. P1 requests R2, which is locked by P2.
4. P2 requests R1, which is locked by P1.

→ If don't use locks, can lead to a race condition – the updated records in the database might include only some of the data and their contents would depend on the order in which each process finishes its execution.

Example:

- Program 1 (P1) needs to update pages and authors for one record in Pages and Authors tables within Database
- Program 2 (P2) also needs to update pages and authors for another record in both tables of the Database
- P1 holds and locks Pages table
- P2 holds and lock Authors table
- P1 requests Authors table to finish
- P2 requests Pages table to finish
- And so on . . .

3.2.3 Case 3: Deadlocks in Dedicated Device Allocation

→ Deadlock can occur when there is a limited number of dedicated devices such as printers, plotters or tape drives.

→ Consider the following sequence of events:

1. P1 requests tape drive 1 and gets it.
2. P2 requests tape drive 2 and gets it.
3. P1 requests tape drive 2 but is blocked.
4. P2 requests tape drive 1 but is blocked.

→ Neither job can continue because each is waiting for the other to finish and release its tape drive - an event that will never occur.

3.2.4 Case 4: Deadlocks in Multiple Device Allocation

→ Deadlocks can happen when several processes request, and hold on to, dedicated devices while other processes act in a similar manner.

→ Three processes P1, P2 and P3, are each waiting for a device – scanner/printer/plotter that has already been allocated to another process, thus creating a deadlock.

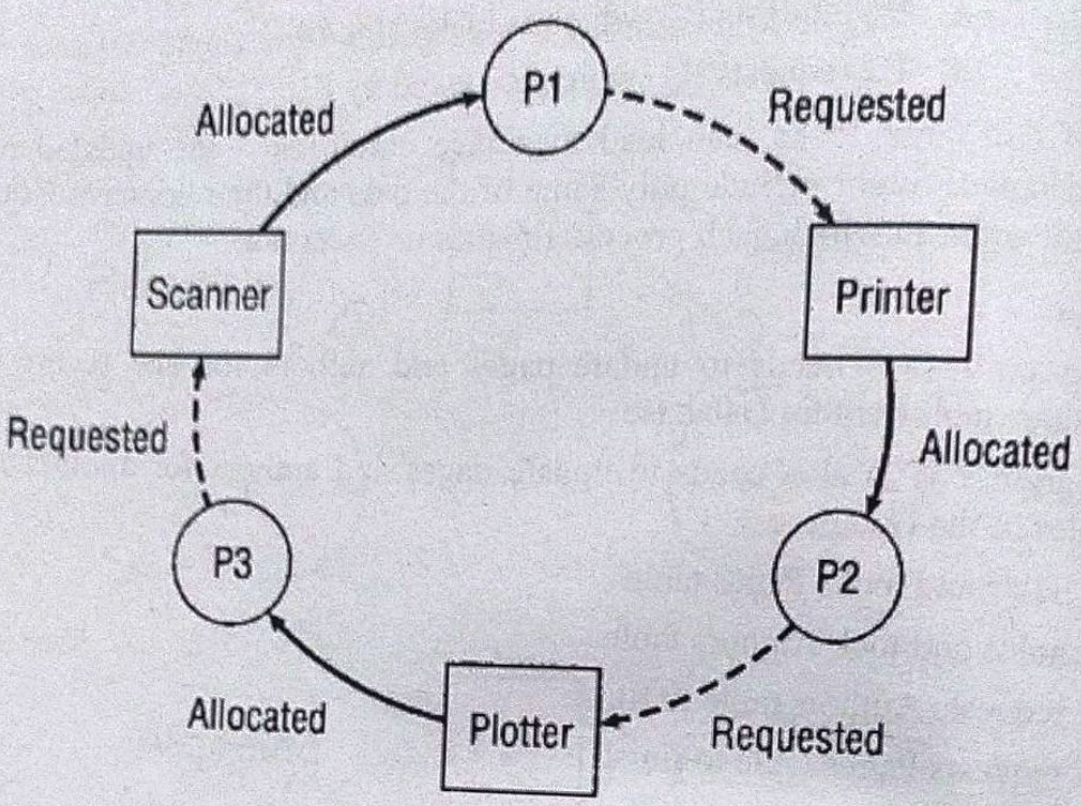


Fig. 3.4 Deadlocks on File Requests

→ The following sequence of events will result in deadlock:

1. P1 requests and gets the scanner.
2. P2 requests and gets the printer.
3. P3 requests and gets the plotter.
4. P1 requests the printer but is blocked.
5. P2 requests the plotter but is blocked.
6. P3 requests the scanner but is blocked.

→ As in the earlier examples, none of the jobs can continue because each is waiting for a resource being held by another.

3.2.5 Case 5: Deadlocks in Spooling

→ Most systems have transformed dedicated devices such as a printer into a sharable/virtual device by installing a high-speed device, a disk, between it and the CPU.

→ Disk accepts output from several users and acts as a temporary storage area for all output until printer is ready to accept it (spooling).

→ If printer needs all of a job's output before it will begin printing, but spooling system fills available disk space with only partially completed output, then a deadlock can occur.

3.2.6 Case 6: Deadlocks in a Network

→ A network that's congested (or filled a large percentage of its I/O buffer space) can become deadlocked if it doesn't have protocols to control flow of messages through network.

→ For example, consider a medium-sized network having seven computers, each on different nodes.

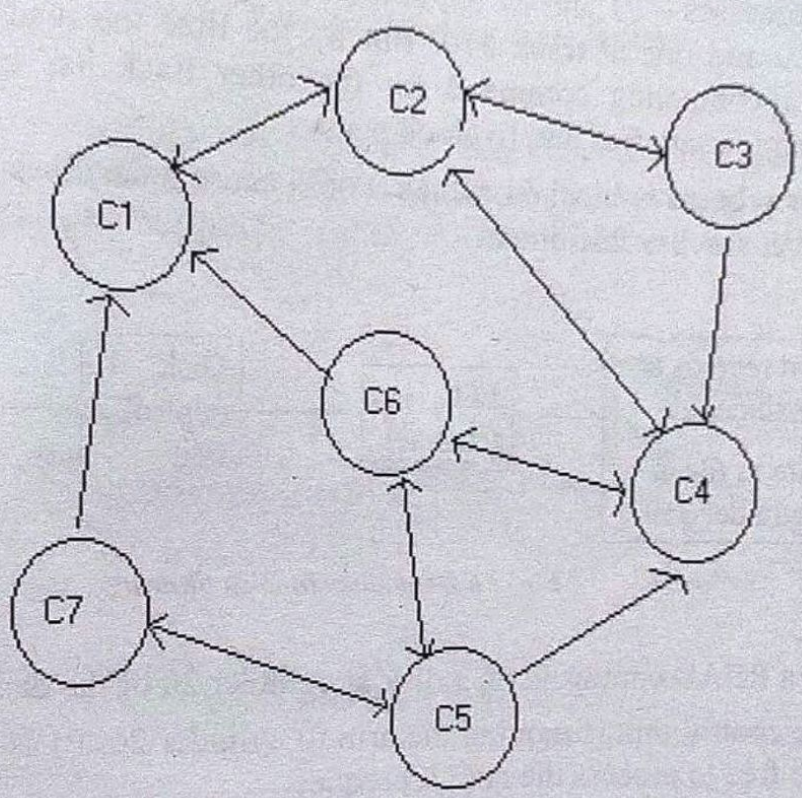


Fig. 3.5 Deadlocks in a Network

- C1 receives messages from nodes C2, C6 and C7
- C1 sends message to C2
- C2 receives messages from C1, C3, and C4
- C2 sends messages to C1 and C3
- messages received by C1 from C6 and C7 destined for C2, are buffered in output queue
- messages received by C2 from C3 and C4 destined for C1, are buffered in output queue
- at high traffic, buffer space is filled
- C1 can't accept more
- C2 can't accept more
- communication between C1 and C2 is deadlocked

3.2.7 Case 7: Deadlocks in Disk Sharing

→ Disks are designed to be shared, so it's not uncommon for two processes - P1 and P2, to access different areas of same disk.

→ Without controls to regulate use of disk drive, competing processes could send conflicting commands and deadlock the system.

→ Two processes – P1 and P2 are each waiting for an I/O request to be filled: one at track 20 and one at track 310. But by the time the read/write arm reaches one track, a competing command for the other track has been issued, so neither command is satisfied and livelock occurs.

→ Neither process is blocked, which would cause a deadlock. Instead, each is active but never reaches fulfillment.

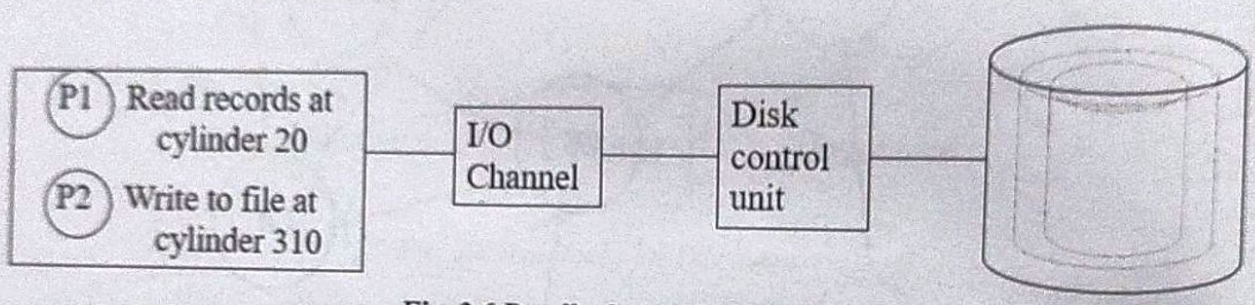


Fig. 3.6 Deadlocks in Disk Sharing

Example:

- P1 issues a READ command on a disk at cylinder 20 of the disk pack.
- While the control unit is moving the arm to cylinder 20, P1 is put on hold and the IO channel is free to process the next IO request.
- P2 gains control of the IO channel and issues WRITE command on the disk at cylinder 310 of the disk pack. P2 will be put on hold until the arm moves to cylinder 310.
- The channel is free and so captured by P1, which reconfirms the command READ cylinder 20.
- The arm is in constant motion, moving back and forth between cylinder 20 and 310, responds to two competing commands but satisfies neither.

3.3 Conditions for Deadlock

Deadlock preceded by simultaneous occurrence of four conditions that operating system could have recognized:

mutual exclusion, resource holding, no preemption, and circular wait.

➢ **Mutual Exclusion:**

There should be a resource that can only be held by one process at a time. Resources shared such as read-only files do not lead to deadlocks but resources, such as printers and tape drives, requires exclusive access by a single process. (Cases: 1, 2, 7)

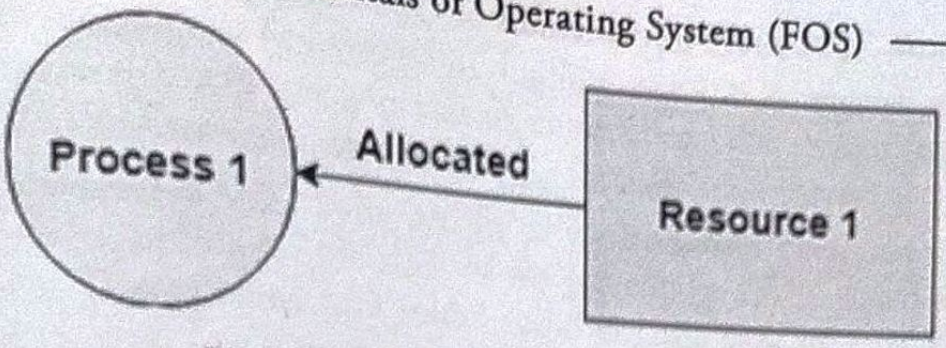


Fig. 3.7 Mutual Exclusion (single process)

- **Hold and Wait:**

A process can hold multiple resources and still request more resources from other processes which are holding them. Processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. (Cases: 1, 2, 3, 4, 6, 7)

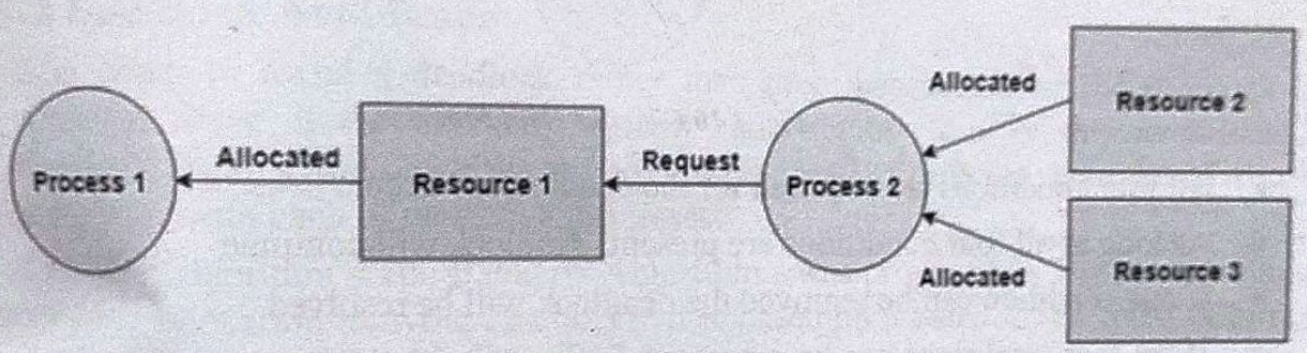


Fig. 3.8 Hold and Wait Process

- **No Preemption:**

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile. (Case: 5)

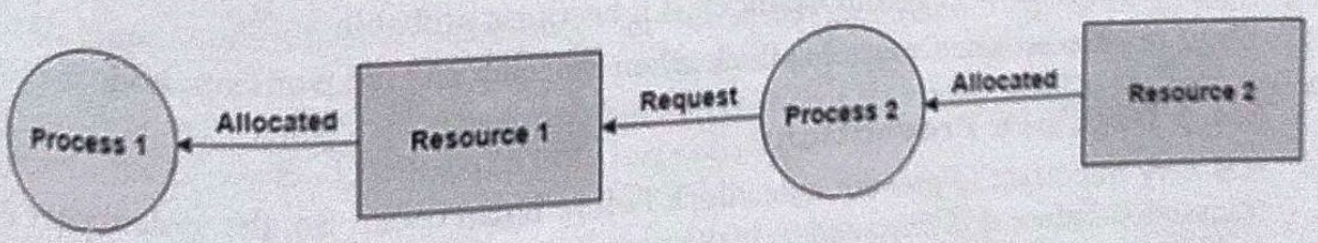


Fig. 3.9 No Preemption Process

- **Circular Wait:**

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. Circular wait can be avoided if we number all resources, and require that processes request resources only in strictly increasing (or decreasing) order. (Case: 2)

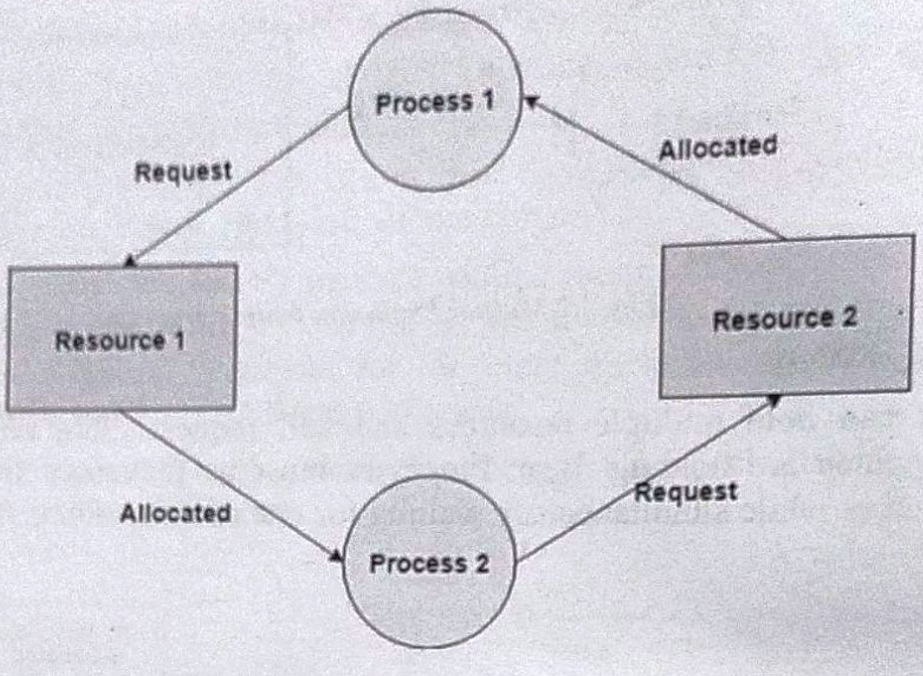


Fig. 3.10 Circular Wait process

- All four conditions are required for the deadlock to occur
- As long as all four conditions are present, deadlock will continue
- If one condition can be removed the deadlock will be resolved
- If the four conditions can be prevented from occurring at the same time, deadlock can be prevented (not easy to implement)

3.4 Strategies for handling Deadlocks

→ In general there are three strategies to deal with deadlock:

- **Prevention:** Prevent one of the four conditions of deadlock
- **Avoidance:** Avoid deadlock if it becomes probable
- **Detection:** Detect deadlock when it occurs and recover from it kindly

3.4.1 Deadlock Prevention:

It is very important to prevent a deadlock before it can occur. So, the system checks each transaction before it is executed to make sure it does not lead to deadlock. If there is even a slight chance that a transaction may lead to deadlock in the future, it is never allowed to execute.

To prevent a deadlock, OS must eliminate one out of four necessary conditions - mutual exclusion, resource holding, no preemption, and circular wait.

- Mutual exclusion is necessary in any computer system because some resources (memory, CPU, dedicated devices) must be exclusively allocated to one user at a time.

- Resource holding can be avoided by forcing each job to request, at creation time, every resource it will need to run to completion, but it significantly decreases degree of multiprogramming.
- No preemption could be bypassed by allowing OS to deallocate resources from jobs. This can be done if the state of the job can be easily saved and restored, as when a job is preempted in a round robin environment or a page is swapped to secondary storage in a virtual memory system.
- Circular wait can be bypassed if OS prevents formation of a circle. Apply Havender's solution which is based on a numbering system for resources such as: printer = 1, disk = 2, tape= 3 and so on. Forces each job to request its resources in ascending order. Any "number 1" devices required by job requested first; any "number 2" devices requested next … But a best order is difficult to determine.

3.4.2 Deadlock Avoidance:

It is better to avoid a deadlock rather than take measures after the deadlock has occurred. Even if the operating system can't remove one of the conditions for deadlock, it can avoid one if the system knows ahead of time the sequence of requests associated with each of the active processes.

One such algorithm - The Banker's Algorithm - was proposed by Dijkstra to regulate resource allocation to avoid deadlocks. It is based on a bank with a fixed amount of capital that operates on the following principles:

- No customer will be granted a loan exceeding the bank's total capital.
- All customers will be given a maximum credit limit when opening an account.
- No customer will be allowed to borrow over the limit.
- The sum of all loans won't exceed the bank's total capital.

Under these conditions, the bank isn't required to have on hand the total of all maximum lending quotas before it can open up for business. For example, the bank has a total capital fund of 10,000 and has three customers, C1, C2, and C3, who have maximum credit limits of 4,000, 5,000, and 8,000, respectively.

Safe State:

Customer	Loan amount	Maximum credit	Remaining credit
C1	0	4,000	4,000
C2	2,000	5,000	3,000
C3	4,000	8,000	4,000

Total loaned: 6,000
Total capital fund: 10,000

Table illustrates the state of the bank after some loans have been granted to C2 and C3. This is called a safe state because the bank still has enough money left to satisfy the maximum requests of C1, C2, or C3.

Unsafe State:

Customer	Loan amount	Maximum credit	Remaining credit
C1	2,000	4,000	2,000
C2	3,000	5,000	2,000
C3	4,000	8,000	4,000

Total loaned: 9,000
Total capital fund: 10,000

This is an unsafe state because with only 1,000 left, the bank can't satisfy anyone's maximum request; and if the bank lent the 1,000 to anyone, then it would be deadlocked (it can't make a loan).

Problems with Banker's Algorithm:

— As they enter system, jobs must state in advance the maximum number of resources needed.

— Number of total resources for each class must remain constant.

— Number of jobs must remain fixed.

— Overhead cost incurred by running the avoidance algorithm can be quite high.

— Resources aren't well utilized because the algorithm assumes the worst case.

— Scheduling suffers as a result of the poor utilization and jobs are kept waiting for resource allocation.

3.4.3 Deadlock Detection:

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods:

1. All the processes that are involved in the deadlock are terminated. This is not a good approach as all the progress made by the processes is destroyed.
2. Resources can be preempted from some processes and given to others till the deadlock is resolved.

Use directed graphs to show circular wait which indicates a deadlock. Algorithm used to detect circularity can be executed whenever it is appropriate. The detection algorithm can be explained by using directed resource graphs and "reducing" them. The steps to reduce a graph are these:

- Find a process that is currently using a resource and *not waiting* for one. Remove this process from graph and return resource to "available list."
- Find a process that's waiting only for resource classes that aren't fully allocated.
- Process isn't contributing to deadlock since eventually gets resource it's waiting for, finish its work, and return resource to "available list."
- Go back to Step 1 and continue the loop until all lines connecting resources to processes have been removed.

If there are any lines left, this indicates that the request of the process in question can't be satisfied and that a deadlock exists. Figure illustrates a system in which three processes—Pl, P2, and P3—and three resources—R1, R2, and R3—aren't deadlocked.

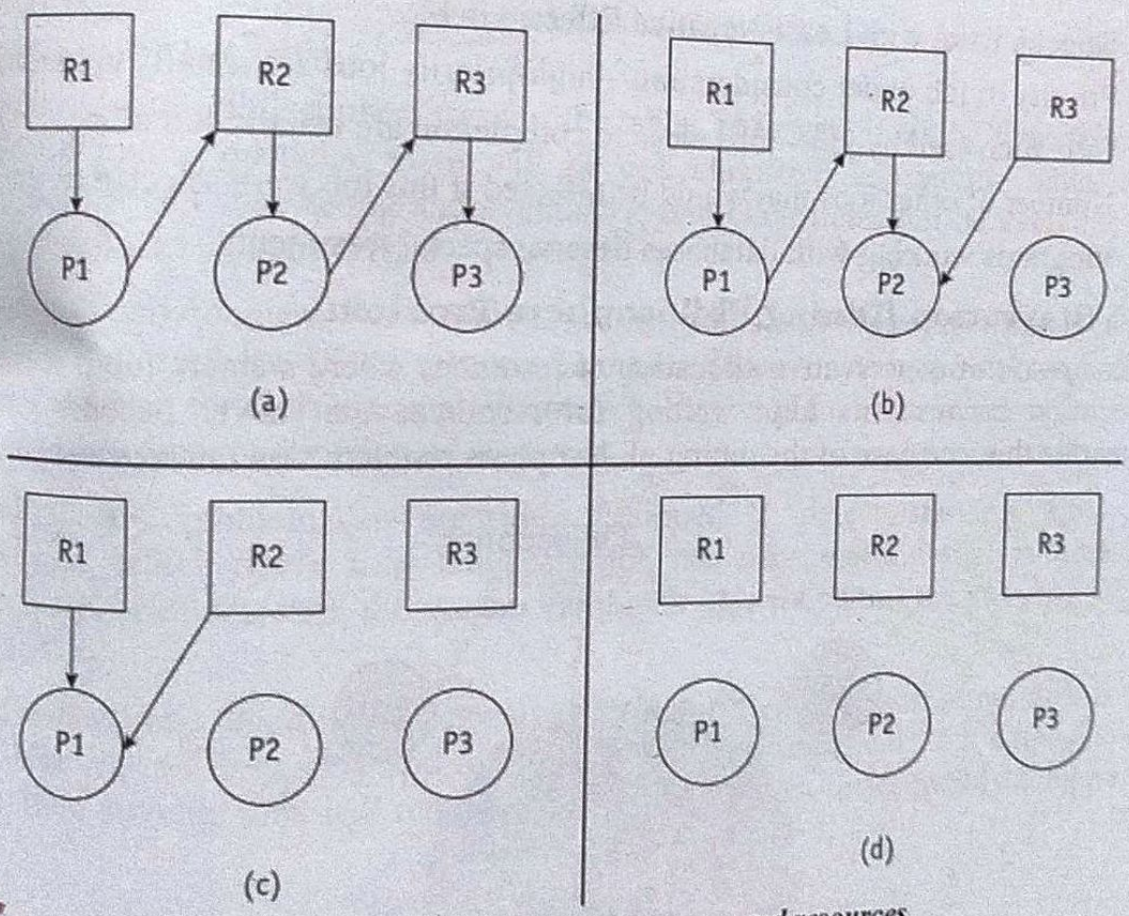


Fig. 3.11 illustrates a system processes and resources

3.4.5 Recovery:

→ Once a deadlock has been detected it must be untangled and system returned to normal as quickly as possible.

→ There are several recovery algorithms, all requiring at least one **victim**, an expendable job, which, when removed from deadlock, frees system.

1. Terminate every job that's active in system and restart them from beginning.

2. Terminate only the jobs involved in deadlock and ask their users to resubmit them.
3. Terminate jobs involved in deadlock one at a time, checking to see if deadlock is eliminated after each removal, until it has been resolved.
4. Have job keep record (snapshot) of its progress so it can be interrupted and then continued without starting again from the beginning of its execution.
5. Select a non-deadlocked job, preempt resources it's holding, and allocate them to a deadlocked process so it can resume execution, thus breaking the deadlock
6. Stop new jobs from entering system, which allows non-deadlocked jobs to run to completion so they'll release their resources (no victim).

Select Victim with Least-Negative Effect:

- Priority of job under consideration—high-priority jobs are usually untouched.
- CPU time used by job—jobs close to completion are usually left alone.
- Number of other jobs that would be affected if this job were selected as the victim.
- Programs working with databases deserve special treatment.

3.5 Starvation (Dining Philosophers Problem)

It is a result of conservative allocation of resources where a single job is prevented from execution because it's kept waiting for resources that never become available. To illustrate this, the case of the dining philosophers problem was introduced by Dijkstra.

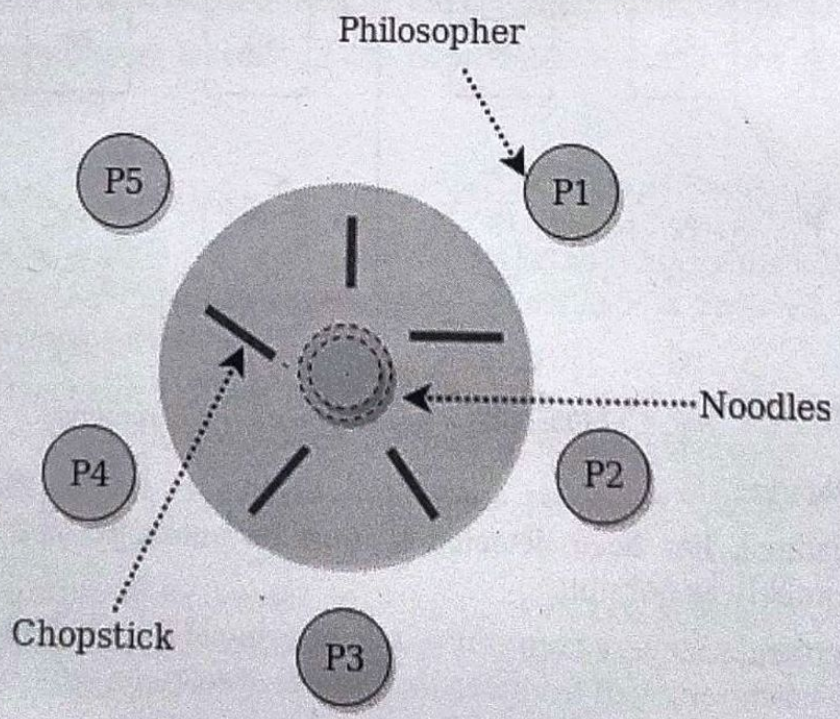


Fig. 3.12 Dining Philosopher Problem

The Dining Philosopher Problem states that K (let us take 5 in our example) philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

If all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

In a computer environment, the resources are like forks and the competing processes are like dining philosophers. If the resource manager doesn't watch for starving processes and jobs, and plan for their eventual completion, they could remain in the system forever waiting for the right combination of resources.

To address this problem, an algorithm designed to detect starving jobs can be implemented, which tracks how long each job has been waiting for resources. Once starvation has been detected, the system can block new jobs until the starving jobs have been satisfied.

This algorithm must be monitored closely: If monitoring is done too often, then new jobs will be blocked too frequently and throughput will be diminished. If it's not done often enough, then starving jobs will remain in the system for an unacceptably long period of time.

Solution of Dining Philosophers Problem:

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is: semaphore chopstick [5];

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows:

```
do
{
        wait( chopstick[i] );
        wait( chopstick[ (i+1) % 5] );
        .       .
        .  EATING
        .
        signal( chopstick[i] );
        signal( chopstick[ (i+1) % 5] );
        .
        . THINKING
        .
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

3.6 Process Synchronization

- When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.
- A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.
- The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization.
- In short, Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

3.7 What is Parallel Processing?

- Parallel processing, one form of multiprocessing, is a situation in which two or more processors operate in unison. That means two or more CPUs are executing instructions simultaneously.
- In multiprocessing systems, the Processor Manager has to coordinate the activity of each processor, as well as synchronize cooperative interaction among the CPUs.
- There are two primary benefits to parallel processing systems: increased reliability (If one processor fails, then the others can continue to operate and absorb the load) and faster processing (The operating system can restructure its resource allocation strategies so the remaining processors don't become overloaded).

3.8 Typical Multi Processing Configurations

Multiprocessor is the set of multiple processors that executes instructions simultaneously. There are basically three configurations of multiprocessor:

Master/Slave, Loosely coupled, and Symmetric

3.8.1 Master/Slave Configuration:

→ An asymmetric multiprocessing system

→ A single-processor system with additional slave processors, each of which is managed by the primary master processor

→ Master processor is responsible for

- Managing the entire system
- Maintaining status of all processes in the system
- Performing storage management activities
- Scheduling the work for the other processors
- Executing all control programs

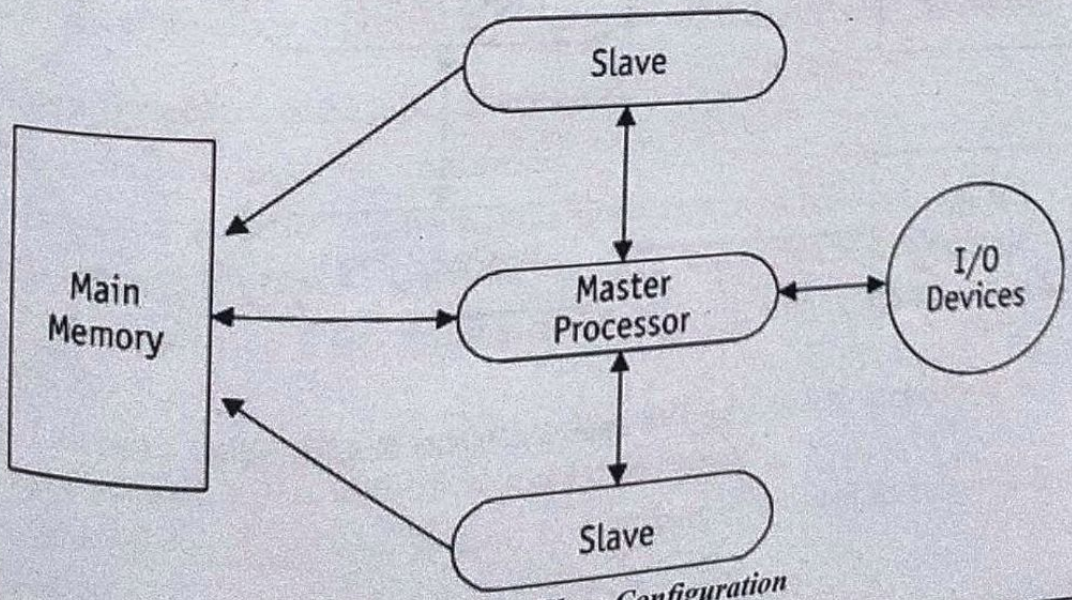


Fig. 3.13 Master/Slave Configuration

Advantage:

→ Simplicity

Disadvantages:

→ Reliability is no higher than for a single processor system

→ Can lead to poor use of resources

→ Increases the number of interrupts

3.8.2 Loosely Coupled Configuration:

→ Each processor has a copy of the OS and controls its own resources, and each can communicate and cooperate with others

→ Once allocated, job remains with the same processor until finished

→ Each has global tables that indicate to which processor each job has been allocated

→ Job scheduling is based on several requirements and policies

→ If a single processor fails, the others can continue to work independently

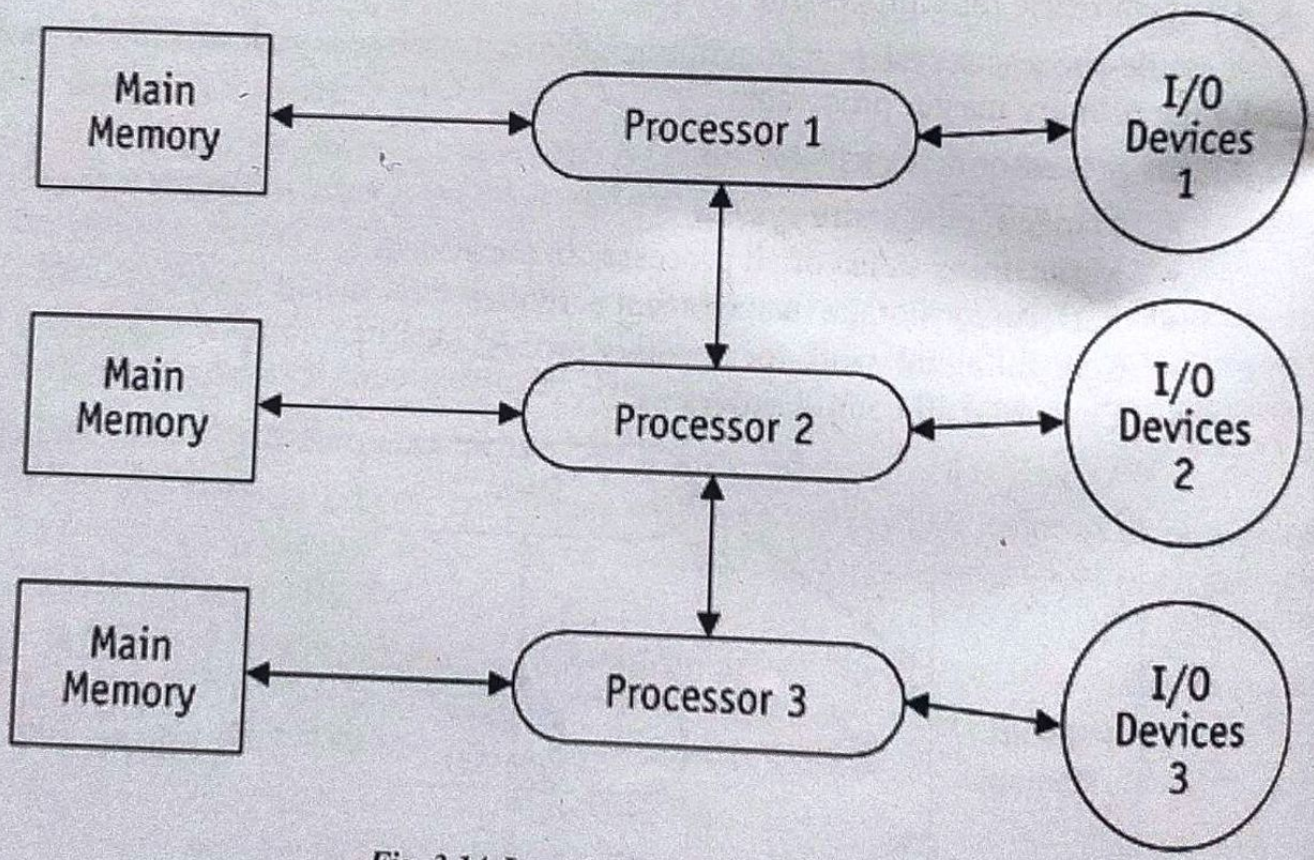


Fig. 3.14 Loosely Coupled Configuration

3.8.3 Symmetric/Tightly Coupled Configuration:

→ Processor scheduling is decentralized and each processor is of the same type

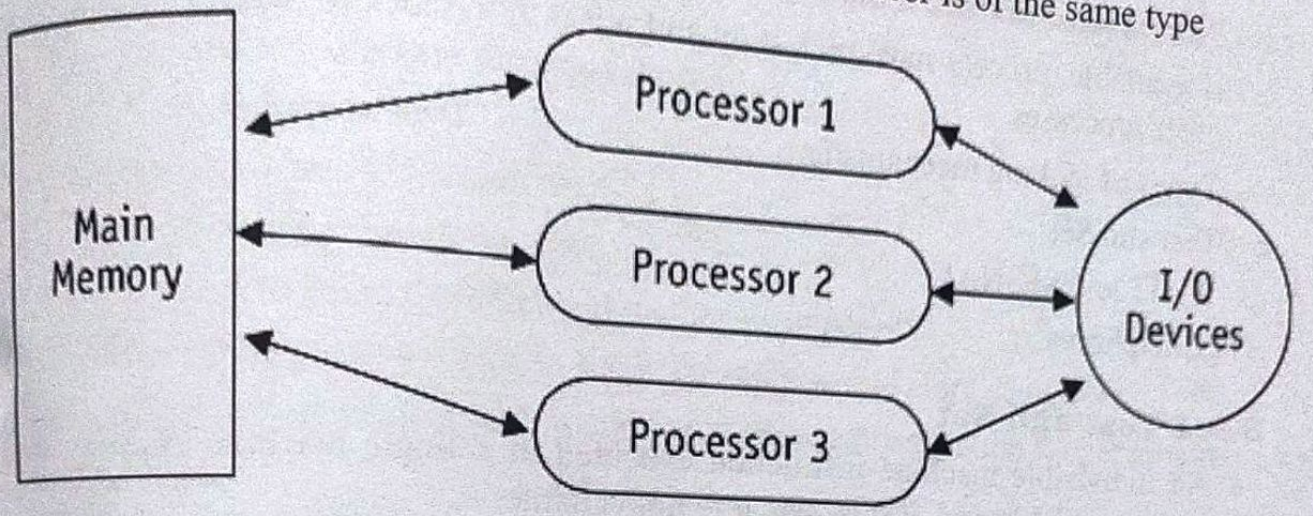


Fig. 3.15 Symmetric/Tightly Coupled Configuration

→ Advantages over loosely coupled configuration:

- More reliable
- Uses resources effectively
- Can balance loads well
- Can degrade gracefully in the event of a failure

→ All processes must be well synchronized to avoid races and deadlocks

→ Any given job or task may be executed by several different processors during its run time

→ More conflicts as several processors try to access the same resource at the same time

→ Process synchronization algorithms to resolve conflicts between processors

3.9 Process Synchronization Software

→ For a successful process synchronization:

- Used resource must be locked from other processes until released
- A waiting process is allowed to use the resource only when it is released

→ A mistake could leave a job waiting indefinitely or if it is a key resource, cause a deadlock

→ **Critical region:** A part of a program that must complete execution before other processes can have access to the resources being used

→ Processes within a critical region can't be interleaved without threatening integrity of the operation

→ Synchronization is sometimes implemented as a lock-and-key arrangement:
- Process must first see if the key is available
- If available, process must pick it up and put it in the lock to make it unavailable to all other processes

→ Types of locking mechanisms:
- Test and Set
- WAIT and SIGNAL
- Semaphores

3.9.1 Test and Set

→ An indivisible machine instruction executed in a single machine cycle to see if the key is available and, if it is, sets it to unavailable

→ The actual key is a single bit in a storage location that can contain a 0 (free) or a 1 (busy)

→ A process P1 tests the condition code using TS instruction before entering a critical region

- If no other process in this region, then P1 is allowed to proceed and condition code is changed from 0 to 1
- When P1 exits, code is reset to 0, allows other to enter

Advantages:

- Simple procedure to implement
- Works well for a small number of processes

Disadvantages:

- Starvation could occur when many processes are waiting to enter a critical region
- Processes gain access in an arbitrary fashion
- Waiting processes remain in unproductive, resource-consuming wait loops (busy waiting)

3.9.2 Wait and Signal

→ Modification of test-and-set designed to remove busy waiting

→ Two new mutually exclusive operations, WAIT and SIGNAL (part of Process Scheduler's operations)

→ *WAIT* is activated when process encounters a busy condition code

→ *SIGNAL* is activated when a process exits critical region and the condition code is set to "free"

3.10 Semaphores

→ A nonnegative integer variable that's used as a flag and signals if and when a resource is free and can be used by a process

→ Two operations to operate the semaphore

- P (proberen means to test)
- V (verhogen means to increment)

→ If "s" is a semaphore variable, then:

- V(s): s: = s + 1 (fetch, increment, and store sequence)
- P(s): If s > 0 then s: = s – 1 (test, fetch, decrement, and store sequence)

→ s = 0 implies busy critical region and the process calling on the P operation must wait until s > 0

→ Choice of which of the waiting jobs will be processed next depends on the algorithm used by this portion of the Process Scheduler

→ Following table shows P and V operations on the binary semaphore s.

	Actions		Results		
State Number	Calling Process	Operation	Running in Critical Region	Blocked on *s*	Value of *s*
0					1
1	P_1	P(*s*)	P_1		0
2	P_1	V(*s*)			1
3	P_2	P(*s*)	P_2		0
4	P_3	P(*s*)	P_2	P_3	0
5	P_4	P(*s*)	P_2	P_3, P_4	0
6	P_2	V(*s*)	P_3	P_4	0
7			P_3	P_4	0
8	P_3	V(*s*)	P_4		1
9	P_4	V(*s*)			

→ P and V operations on semaphore s enforce the concept of mutual exclusion
→ Semaphore is called mutex (MUTual EXclusion)
- P(mutex): if mutex > 0 then mutex: = mutex – 1
- V(mutex): mutex: = mutex + 1

→ Critical region ensures that parallel processes will modify shared data only while in the critical region
→ In parallel computations, mutual exclusion must be explicitly stated and maintained

3.11 Process Cooperation

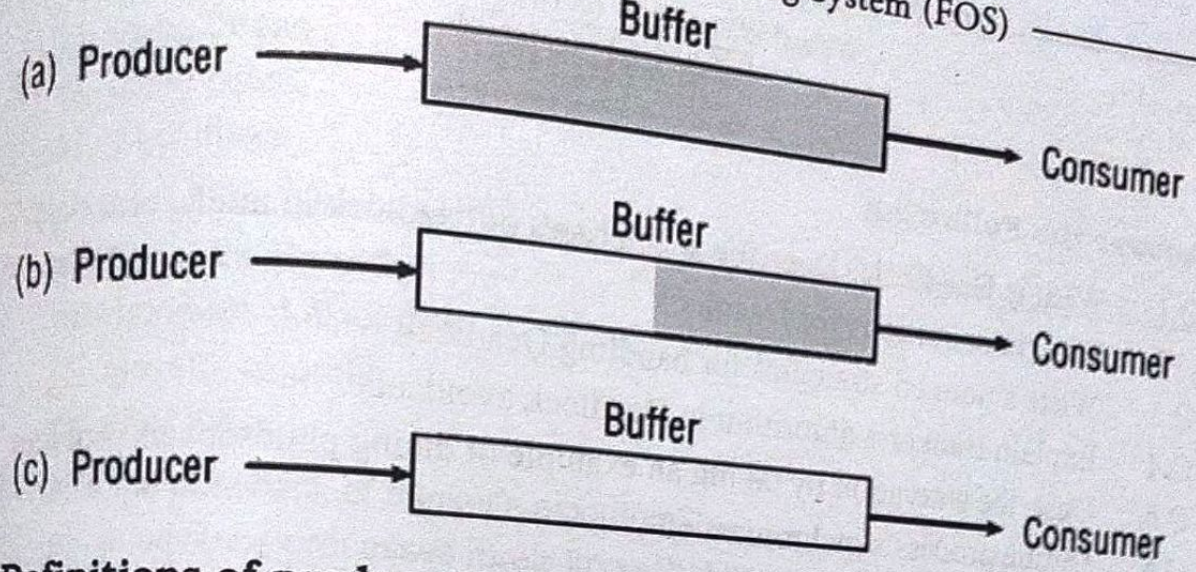
→ Process cooperation: When several processes work together to complete a common task
→ Each case requires both mutual exclusion and synchronization
→ Absence of mutual exclusion and synchronization results in problems
→ Examples:
- Problems of producers and consumers
- Problems of readers and writers

→ Each case is implemented using semaphores

3.11.1 Producers and consumers

→ Arises when one process produces some data that another process consumes later
→ Example: Use of buffer to synchronize the process between CPU and line printer:
- Buffer must delay producer if it's full, and must delay consumer if it's empty
- Implemented by two semaphores – one for number of full positions and other for number of empty positions
- Third semaphore, mutex, ensures mutual exclusion

→ The buffer can be in any one of these three states:
(a) full buffer,
(b) partially empty buffer, or
(c) empty buffer



→ **Definitions of producer and consumer processes:**

Producer	Consumer
produce data	P (full)
P (empty)	P (mutex)
P (mutex)	read data from buffer
write data into buffer	V (mutex)
V (mutex)	V (empty)
V (full)	consume data

→ **Definitions of variables and functions:**

Given: Full, Empty, Mutex defined as semaphores

n: maximum number of positions in the buffer

V (x): x: = x + 1 (x is any variable defined as a semaphore)

P (x): if x > 0 then x: = x – 1

mutex = 1 means the process is allowed to enter the critical region

→ **Producers and Consumers Algorithm:**

empty: = n
full: = 0
mutex: = 1
COBEGIN
repeat until no more data PRODUCER
repeat until buffer is empty CONSUMER
COEND

Exercises

Answer the Following

Q 1. What is Deadlock? Describe seven cases of deadlock in brief.
Q 2. Explain conditions for Deadlock.
Q 3. Write a note on strategies for handling Deadlock.
Q 4. Explain Banker's algorithm in Deadlock avoidance.
Q 5. Describe starvation by taking an example of dining philosophers problem.
Q 6. Define process synchronization.
Q 7. What is parallel processing?
Q 8. Explain Master Slave configuration for multiprocessing system with its merits and demerits.
Q 9. Write a note on Loosely Coupled configuration.
Q 10. Describe Symmetric/Tightly Coupled configuration.
Q 11. Explain Test and Set in process synchronization software.
Q 12. Explain Wait and Signal in process synchronization software.
Q 13. Write a note on Semaphores.
Q 14. Explain Producers and Consumers problem in process cooperation.

Multiple choice Question:

(Note : The Answer is indicated in Bold)

1. Which of the following condition is required for deadlock to be possible?
   a) mutual exclusion
   b) a process may hold allocated resources while awaiting assignment of other resources
   c) no resource can be forcibly removed from a process holding it
   d) all of the above
2. A system is in the safe state if _____
   a) the system can allocate resources to each process in some order and still avoid a deadlock
   b) there exist a safe sequence
   c) Both
   d) None
3. The circular wait condition can be prevented by _____

a) **defining a linear ordering of resource types**
b) using thread
c) using pipes
d) all of these

4. Which one of the following is the deadlock avoidance algorithm?
a) **banker's algorithm**
b) round-robin algorithm
c) elevator algorithm
d) all of these

5. What is the drawback of banker's algorithm?
a) in advance processes rarely know that how much resource they will need
b) the number of processes changes as time progresses
c) resource once available can disappear
d) **all of these**

6. A problem encountered in multitasking when a process is perpetually denied necessary resources is called
a) deadlock
b) **starvation**
c) aging
d) none of these

7. Which one of the following is a visual (mathematical) way to determine the deadlock occurrence?
a) **resource allocation graph**
b) starvation graph
c) inversion graph
d) none of these

8. To avoid deadlock
a) **there must be a fixed number of resources to allocate**
b) resource allocation must be done only once
c) all deadlocked processes must be aborted
d) inversion technique can be used

9. A set of processes is deadlock if
a) **each process is blocked and will remain so forever**
b) each process is terminated
c) all processes are trying to kill each other
d) none of these

10. To avoid the race condition, the number of processes that may be simultaneously inside their critical section is _____

a) 2
b) 1
c) 8
d) 16

11. A system has 3 processes sharing 4 resources. If each process needs a maximum of 2 units, then _____

a) deadlock can never occur
b) deadlock may occur
c) deadlock has to occur
d) none of these

12. A process said to be in _____ state if it was waiting for an event that will never occur.

a) Safe
b) Unsafe
c) Starvation
d) Deadlock

13. In one of the deadlock prevention methods, impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. This violates the _____ condition of deadlock.

a) Mutual exclusion
b) Hold and Wait
c) Circular Wait
d) No Preemption

14. A set of resources' allocations such that the system can allocate resources to each process in some order, and still avoid a deadlock is called _____.

a) Unsafe state
b) Safe state
c) Starvation
d) Greedy allocation

15. Situations where two or more processes are reading or writing some shared data and the final results depends on the order of usage of the shared data, are called _____.

a) Race conditions
b) Critical section

c) Mutual exclusion
d) Deadlocks

16. When two or more processes attempt to access the same resource a _____ occurs.
a) Critical section
b) Communication problem
c) Race condition
d) None of these

17. A process is starved _____
a) if it is permanently waiting for a resource
b) if semaphores are not used
c) if a queue is not used for scheduling
d) if demand paging is not properly implemented

18. Let S and Q be two semaphores initialized to 1, where P0 and P1 processes the following statements wait(S); wait(Q); ---; signal(S); signal(Q) and wait(Q); wait(S); --- ; signal(Q); signal(S); respectively. The above situation depicts a _____.
a) Semaphore
b) Deadlock
c) Signal
d) Interrupt

19. The Banker's algorithm is used _____
a) to prevent deadlock in operating systems
b) to detect deadlock in operating systems
c) to rectify a deadlocked state
d) none of the above

20. Semaphore is a/an _____ to solve the critical section problem.
a) Hardware for a system
b) Special program for a system
c) Integer variable
d) None of these

21. In a master/slave multiprocessing configuration, _____ processors can access main memory directly but they must send all I/O requests through the _____ processor.
a) Slave, Master
b) Master, Slave

c) Symmetric, Loosely Coupled
d) Tightly Coupled, Symmetric

22. In a _____ multiprocessing configuration, each processor has its own dedicated resources.
a) Master slave
b) Loosely coupled
c) Symmetric
d) None of these

23. A _____ multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation.
a) Master slave
b) Loosely coupled
c) Symmetric
d) None of these

24. _____ is a single, indivisible machine instruction and was introduced by IBM for its multiprocessing System 360/370 computers.
a) Test-and-set
b) Wait-and-signal
c) Semaphore
d) None of the above

25. _____ is activated when the process encounters a busy condition code.
a) WAIT
b) SIGNAL
c) TEST
d) SET

26. _____ is activated when a process exits the critical region and the condition code is set to "free."
a) WAIT
b) SIGNAL
c) TEST
d) SET

27. _____ sets the process's process control block (PCB) to the blocked state and links it to the queue of processes waiting to enter this particular critical region.
a) WAIT

b) SIGNAL
c) TEST
d) SET

28. A _____ is a non-negative integer variable that's used as a binary signal, a flag.
a) WAIT-SIGNAL
b) GO-AHEAD
c) TEST-SET
d) Semaphore

29. The bounded buffer problem is also known as _____.
a) Readers – Writers problem
b) Dining – Philosophers problem
c) Producer – Consumer problem
d) None of these

30. The dining – philosophers problem will occur in case of _____.
a) 5 philosophers and 5 chopsticks
b) 4 philosophers and 5 chopsticks
c) 3 philosophers and 5 chopsticks
d) 6 philosophers and 5 chopsticks

31. A mutex: _____
a) is a binary mutex
b) must be accessed from only one process
c) can be accessed from multiple processes
d) none of these

32. A binary semaphore is a semaphore with integer values: _____
a) 1
b) -1
c) 0.8
d) 0.5