UNIT-III

INTER-PROCESS COMMUNICATION

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

1. **Independent Process:** Any process that does not share data with any other process. An Independent process does not affect or be affected by the other processes executing in the system.
2. **Cooperating Process:** Any process that shares data with other processes. A cooperating process can affect or be affected by the other processes executing in the system. Cooperating processes require an **Inter-Process Communication (IPC)** mechanism that will allow them to exchange data and information.

Reasons for providing cooperative process environment:

· **Information Sharing**: Several users may be interested in the same piece of information (i.e. a shared file), we must provide an environment to allow concurrent access to such information.

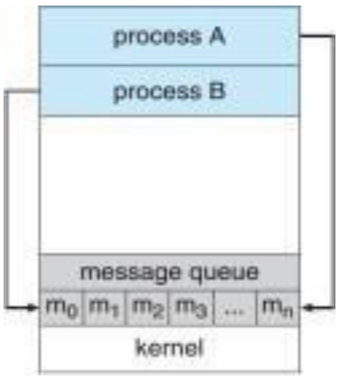
· **Computation Speedup:** If we want a particular task to run faster, we must break it into subtasks. Each task will be executing in parallel with the other tasks.

· **Modularity**: Dividing the system functions into separate processes or threads. · **Convenience**: Even an individual user may work on many tasks at the same time. For example a user may be editing, listening to music and compiling in parallel.

There are two models of IPC: **Message passing** and **Shared memory**.

Message-Passing Systems

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.



· Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

· It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

· A message-passing facility provides two operations: send, receive.

· Messages sent by a process can be either fixed or variable in size.

Example: An Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

P and Q are two processes wants to communicate with each other then they send and receive messages to each other through a communication link such as Hardware bus or Network. Methods for implementing a logical communication links are:
1. Naming
2. Synchronization
3. Buffering

Naming
Processes that want to communicate use either **Direct** or **Indirect** communication. In **Direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.
· send(P, message) — Send a message to process P.
· receive(Q, message) — Receive a message from process Q.
A communication link in direct communication scheme has the following properties: · 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. (i.e.) between each pair of processes, there exists exactly one link.

In **Indirect communication**, the messages are sent to and received from **mailboxes** or **ports**.
· A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
· Each mailbox has a unique integer identification value.

A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.
· send (A, message) — Send a message to mailbox A.
· receive (A, message) — Receive a message from mailbox A.

In this scheme, a communication link has the following properties:
· 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.
· Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Synchronization
Message passing done in two ways:
1. Synchronous or Blocking
2. Asynchronous or Non-Blocking
· **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
· **Non-blocking send:** The sending process sends the message and resumes operation. ·
Blocking receive: The receiver blocks until a message is available.
· **Non-blocking receive:** The receiver retrieves either a valid message or a null.

Buffering

Messages exchanged by communicating processes reside in a temporary queue. Those queues can be implemented in three ways:

1. **Zero Capacity:** Zero-capacity is called as a message system with no buffering. The sender must block until the recipient receives the message.
2. **Bounded Capacity:** The queue has finite length n. Hence at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. If the link is full, the sender must block until space is available in the queue.
3. **Unbounded Capacity:** The queue's length is potentially infinite. Hence any number of messages can wait in it. The sender never blocks.

<u>IPC - Pipes in UNIX</u>

Pipes were one of the first IPC mechanisms in early UNIX systems.
The pipes in UNIX are categorized into two types: **Ordinary Pipes** and **Named Pipes**

Ordinary Pipes

· Ordinary pipes allow two processes to communicate in standard producer–consumer fashion.

· The producer writes to one end of the pipe (**write-end**) and the consumer reads from the other end (**read-end**).

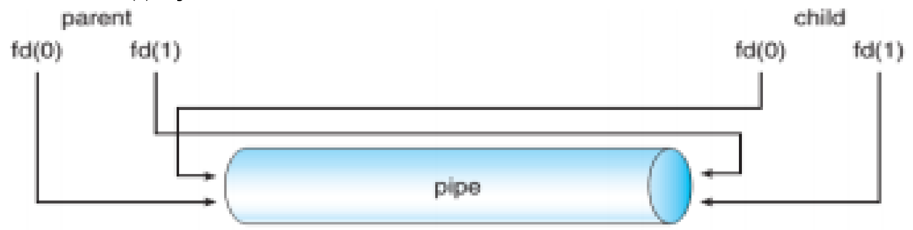
· Ordinary pipes are unidirectional which allows only one-way communication. If two-way communication is required, two pipes must be used. Each pipe transfer data in a different direction.

On UNIX systems ordinary pipes are constructed using the function:

pipe(int fd[ ])

· This function creates a pipe that is accessed through the int fd[ ] file descriptors: fd[0] is the read-end of the pipe and fd[1] is the write-end.

· UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read( ) and write( ) system calls.



An ordinary pipe cannot be accessed from outside the process that created it. · A parent process creates a pipe and uses it to communicate with a child process. · A child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process.

· If a parent writes to the pipe then the child reads from pipe.

Named Pipes

Named pipe provides the bidirectional communication and no parent–child relationship is required.

· Once a named pipe is established, several processes can use it for communication. A named pipe has several writers.

· Named pipes continue to exist after communicating processes have finished. ·
Both UNIX and Windows systems support named pipes.

Named pipes are referred to as FIFOs in UNIX systems.

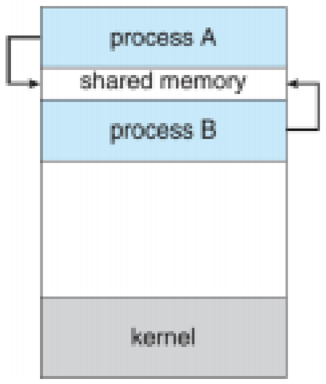
· Once Named pipes are created, they appear as typical files in the file system. · FIFO is created with the mkfifo( ) system call and manipulated with the ordinary open( ), read( ), write( ) and close( ) system calls.

· It will continue to exist until it is explicitly deleted from the file system. · Although FIFOs allow bidirectional communication, only one-way transmission is permitted. If data must travel in both directions, two FIFOs are used. The communicating processes must reside on the same machine.

· If inter-machine communication is required, sockets must be used.

Shared Memory Systems

In the shared-memory model, a region of memory will be shared by cooperating processes. · Processes can exchange information by reading and writing data to the shared region. · A shared-memory region (segment) resides in the address space of the process. · Other processes that wish to communicate using this shared-memory segment must attach it to their address space.



· Normally, the operating system tries to prevent one process from accessing another process's memory.

· Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. · The form of the data and the location are determined by these processes and are not under the operating system's control.

· The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Producer-Consumer Problem in Cooperative process

A producer process produces information that is consumed by a consumer process. Example: A compiler may produce assembly code that is consumed by an assembler. The assembler may produce object modules that are consumed by the loader.

One solution to the producer–consumer problem uses shared memory.

· To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. · This buffer will reside in a region of memory that is shared by the producer and consumer processes.

· A producer can produce one item while the consumer is consuming another item. · The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. Unbounded Buffer and Bounded Buffer · **Unbounded Buffer:** The size of the buffer is not limited. The consumer may have to wait for new items, but the producer can always produce new items.

· **Bounded Buffer:** The buffer size is limited. The consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Code for Producer and Consumer Process in Bounded Buffer IPC

Following variables reside in shared memory region by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
............
}item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The code for the **Producer process** can be modified as follows:

```
while (true)
{
        /* produce an item in next_produced */
while (counter == BUFFER_SIZE) ; /*do nothing Buffer full */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

The code for the **Consumer process** can be modified as follows:

```
while (true)
{
    while (counter == 0); /* do nothing Buffer Empty */
 next_consumed = buffer[out];
 out = (out + 1) % BUFFER SIZE;
    counter--; /* consume the item in next_consumed */
 }
```

The shared buffer is implemented as a circular array with two logical pointers: in and out. ·

The variable **—in‖** points to the next free position in the buffer and **—out‖** points to the first full position in the buffer.

· An integer variable counter is initialized to 0. Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

· The buffer is empty when counter== 0 and the buffer is full when counter== Buffer_size. · The producer process has a local variable next_produced in which the new item to be produced is stored.

· The consumer process has a local variable next_consumed in which the item to be consumed is stored.

Note: Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.

Example: Let us consider the **counter =5**. Producer and consumer processes concurrently execute the statements **—counter++‖** and **—counter--‖**.
Let R1 and R2 are two registers. the statement —counter++‖ may be implemented in machine language as follows:

T0: R1 = counter
T1: R1 = R1 + 1
T2: counter = R1

where register1 is one of the local CPU registers. Similarly, the statement —counter--‖ is implemented as follows:

T3: R2 = counter
T4: R2 = R2 – 1
T5: counter = R2

We execute the statements in the order **T0, T1, T2, T3, T4, T5,T6** then we get the accurate counter value=5.

Now if these statements are executed concurrently by interleaving as follows: **T0:** R1 = counter {R1 = 5, counter=5}
T1: R1 = R1 + 1 {R1 = 6, counter=5}
T2: R2 = counter {R2 = 5, counter=5}
T3: R2 = R2 − 1 {R2 = 4, counter=5}
T4: counter = R1 {counter = 6, R1=6}
T5: counter = R2 {counter = 4, R2=4}

Note that we have arrived at the incorrect state —counter == 4‖, indicating that four buffer locations are full but actually five buffer locations are full.
If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state —counter == 6‖.

Race condition

· Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a Race Condition.

· To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized.

THE CRITICAL-SECTION PROBLEM

Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}.

· Each process has a segment of code, called a **Critical Section**, in which the process may be changing common variables, updating a table, writing a file and so on. · When one process is executing in its critical section, no other process is allowed to execute in its critical section.

```
do {
        Entry section

        Critical Section
        Exit section

        Remainder section
} while (true);
```

· Each process must request permission to enter its critical section. The section of code implementing entering request is the **Entry section**.

· The critical section may be followed by an **Exit section**.

· The remaining code is the **Remainder section**.

· The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements: 1. **Mutual exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next and this selection cannot be postponed indefinitely.

3. **Bounded waiting**. After a process has made a request to enter its critical section and before that request is granted, there exists a limit on the number of times that other processes are allowed to enter their critical sections..

Two general approaches are used to handle critical sections in operating systems: 1. **Preemptive Kernel** allows a process to be preempted while it is running in kernel mode. 2. **Non-preemptive Kernel** does not allow a process running in kernel mode to be preempted.

PETERSON’S SOLUTION

Peterson’s solution is a classic **Software-Based Solution** to the critical-section problem. Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered ***P0*** and ***P1***. Let ***Pi*** represents one process and ***Pj*** represents other processes (i.e. j = i-1)

```
do
{
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        Critical Section
    flag[i] = false;

Remainder Section
} while (true);
```

Peterson’s solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section. At any point of time the turn value will be either 0 or 1 but not both.

· if turn == i, then process *Pi* is allowed to execute in its critical section. · if turn == j, then process *Pj* is allowed to execute in its critical section. · The flag array is used to indicate if a process is ready to enter its critical section. Example: if flag[i] is true, this value indicates that *Pi* is ready to enter its critical section. · To enter the critical section, process *Pi* first sets **flag[i]=true** and then sets **turn=j**,

thereby **Pi** checks if the other process wishes to enter the critical section, it can do so. · If both processes try to enter at the same time, turn will be set to both i and j at the same time. Only one of these assignments will be taken. The other will occur but will be overwritten immediately.

· The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

The above code must satisfy the following requirements:

1. Mutual exclusion
2. The progress
3. The bounded-waiting

Check for Mutual Exclusion

· Each ***Pi*** enters its critical section only if either **flag[j] == false** or **turn == i**. · If both processes can be executing in their critical sections at the same time, then **flag[0] == flag[1] == true**. But the value of turn can be either **0** or **1** but cannot be both. · Hence ***P*0** and ***P*1** could not have successfully executed their **while** statements at about the same time.

· If ***Pi*** executed **―turn == j‖** and the process ***Pj*** executed **flag[j]=true** then Pj will have successfully executed the while statement. Now Pj will enter into its **Critical section**.

· At this time, **flag[j] == true** and **turn == j** and this condition will persist as long as ***Pj*** is in its critical section. As a result, mutual exclusion is preserved.

Check for Progress and Bounded-waiting

· The while loop is the only possible way that a process ***Pi*** can be prevented from entering the critical section only if it is stuck in the while loop with the condition **flag[j] == true** and **turn == j**.

· If ***Pj*** is not ready to enter the critical section, then **flag[j] == false** and ***Pi*** can enter its critical section.

· If ***Pj*** has set **flag[j] == true** and is also executing in its while statement, then either **turn == i** or **turn == j**.

· If **turn == i**, then ***Pi*** will enter the critical section. If **turn == j**, then ***Pj*** will enter the critical section.

· Once ***Pj*** exits its critical section, it will reset **flag[j]** to false, allowing ***Pi*** to enter its critical section.

· If ***Pj*** resets **flag[j]** to true, it must also set **turn == i**. Thus, since ***Pi*** does not change the value of the variable **turn** while executing the while statement, ***Pi*** will enter the critical section (**Progress**) after at most one entry by ***Pj*** (**Bounded Waiting**).

Problem with Peterson Solution

There are no guarantees that Peterson's solution will work correctly on modern computer architectures perform basic machine-language instructions such as load and store.

· The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. This is an Non-preemptive kernel approach.

· We could be sure that the current sequence of instructions would be allowed to execute in order without preemption.

· No other instructions would be run, so no unexpected modifications could be made to the shared variable.

This solution is not as feasible in a multiprocessor environment.

· Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors.

· This message passing delays entry into each critical section and system efficiency decreases and if the clock is kept updated by interrupts there will be an effect on a system's clock.

SYNCHRONIZATION HARDWARE

Modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically (i.e.)** as one uninterruptible unit.

There are two approaches in hardware synchronization:

1. test_and_set function
2. compare_and_swap function

test_and_set function

The test_and_set instruction is executed atomically (i.e.) if two test_and_set( ) instructions are executed simultaneously each on a different CPU, they will be executed sequentially in some

arbitrary order.
If the machine supports the test_and_set( ) instruction, then we can implement mutual exclusion by declaring a boolean variable **lock.** The lock is initialized to false.

The definition of test_and_set instruction for process **P$_i$** is given as:

```
boolean test_and_set(boolean *target)
{
boolean rv = *target;
*target = true;
return rv;
}
```

Below algorithm satisfies all the requirements of Critical Section problem for the process **P$_i$** that uses two Boolean data structures: waiting[ ], lock.

```
boolean waiting[i] = false;
boolean lock = false;
do
{
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
        key = test and set(&lock);
        waiting[i] = false;

        /* critical section */
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
        j = (j + 1) % n;
        if (j == i)
        lock = false;
        else
         waiting[j] = false;

        /* remainder section */
} while (true);
```

Mutual Exclusion
· Process Pi can enter its critical section only if either **waiting[i] == false** or **key == false.** · The value of key can become false only if the **test_and_set( )** is executed. · The first process to execute the **test_and_set( )** will find **key == false** and all other processes must wait.
· The variable **waiting[i]** can become false only if another process leaves its critical section.
· Only one **waiting[i]** is set to **false**, maintaining the mutual-exclusion requirement.

Progress
· A process exiting the critical section either sets **lock==false** or sets **waiting[j]==false**. · Both allow a process that is waiting to enter its critical section to proceed. · This requirement ensures progress property.

Bounded Waiting

· When a process leaves its critical section, it scans the array waiting in the cyclic ordering (**i+1, i+2, ..., n−1, 0, ..., i−1**).

· It designates the first process in this ordering that is in the entry section (**waiting[j] == true**) as the next one to enter the critical section.

· Any process waiting to enter its critical section will thus do so within **n−1** turns. · This requirement ensures the bounded waiting property.

<u>compare_and_swap function</u>

compare_and_swap( ) is also executed atomically. The compare_and_swap( ) instruction operates on three operands. The definition code as given below:

```
int compare_and_swap(int *value, int expected, int new_value)
{
        int temp = *value;
        if (*value == expected)
        *value = new_value;
        return temp;
}
```

Mutual-exclusion implementation with the compare and swap( ) instruction: do

```
{
   while (compare_and_swap(&lock, 0, 1) != 0); /* do nothing */
/* critical section */
lock = 0;
/* remainder section */
} while (true);
```

The operand value is set to new value only if the expression (*value == expected) is true. Regardless, compare_and_swap( ) always returns the original value of the variable value.

Mutual Exclusion with compare_and_swap( )

· A global variable lock is declared and is initialized to 0 **(i.e. lock=0**). · The first process that invokes compare_and_swap( ) will set **lock=1**. It will then enter its critical section, because the original value of lock was equal to the expected value of 0. · Subsequent calls to compare_and_swap( ) will not succeed, because lock now is not equal to the expected value of 0. (**lock==1**).

· When a process exits its critical section, it sets lock back to 0 (**lock ==0**), which allows another process to enter its critical section.

Problem with Hardware Solution:

The hardware-based solutions to the critical-section problem are complicated and they are inaccessible to application programmers.

<u>MUTEX LOCKS</u>

Mutex Locks are short for Mutual Exclusive Locks.

· Mutex locks are software solution to the critical section problem.

· Mutex lock are used to protect critical regions and thus prevent race conditions. · In Mutex locking, a process must acquire the lock before entering a critical section and it releases the lock when it exits the critical section.

· The acquire( )function acquires the lock and the release( ) function releases the lock.

Critical section solution through Mutex locks:

```
do
{
    acquire( )
    {
     while (!available); /* busy wait */
    available = false;
     }

        Critical Section
    release( )
     {
     available = true;
     }

        remainder section
} while (true);
```

Explanation of Algorithm:

· A mutex lock has a boolean variable **available** whose value indicates if the lock is available or not.

· If the lock is available, a call to acquire( ) succeeds and the lock is then set **unavailable** to other processes.

· A process that attempts to acquire an unavailable lock is blocked until the lock is released.

· Calls to either acquire( ) or release( ) must be performed atomically.

Disadvantage with Mutex Locks: Busy Waiting

Mutex locks implementation leads to Busy Waiting.

· While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire( ).

· This type of mutex lock is also called a **Spinlock** because the process ―spins‖ while waiting for the lock to become available.

·This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes a lot of CPU cycles.

SEMAPHORES

Semaphores provides solution to the critical section problem.

A **semaphore** S is an integer variable that is accessed only through two standard atomic

operations: wait( ) and signal( ).
The definition of wait( ) and signal( ) are as follows:

```
wait(S)
{
        while (S <= 0); // busy wait
        S--;
}
signal(S)
{
        S++;
}
```

All modifications to the integer value of the semaphore in the wait( ) and signal( ) operations must be executed all at once.

· When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

· In wait(S) function, the statements ($S \leq 0$) and its possible modification (S--) must be executed without interruption.

Semaphore Usage

Operating system provides two types of semaphores: Binary and Counting Semaphore. **Binary Semaphore**

· The value of a **binary semaphore** can range only between 0 and 1.

· Binary semaphores behave similarly to mutex locks.

Counting Semaphore

· The value of a **counting semaphore** can range over an unrestricted domain. · Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

· The semaphore is initialized to the number of resources available.

· Each process that wishes to use a resource performs a wait( ) operation on the semaphore and thereby decrements the count value **(S--)**.

· When a process releases a resource, it performs a signal( ) operation and increments the count value **(S++)**.

· When the count for the semaphore goes to 0 **(S<=0)**, all resources are being used. · After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphores provides solution for Synchronization problem

Consider two concurrently running processes: *P*1 with a statement *S*1 and *P*2 with a statement *S*2. Suppose we require that *S*2 be executed only after *S*1 has completed. We can implement this scheme by letting *P*1 and *P*2 share a common semaphore synch, initialized to 0 **(i.e. synch==0).**

In process P1, we insert the statements.

```
S1;
signal(synch);
```

In process P2, we insert the statements

```
wait(synch);
S2 ;
```

Because synch is initialized to 0 **(i.e. synch==0)**, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Semaphore Implementation

Definitions of the wait( ) and signal( ) semaphore operations leads to busy waiting problem. To overcome the problem of busy waiting, the definitions of wait( ) and signal( ) must have to modified as follows:

· When a process executes wait( ) operation and finds that the semaphore value is not positive, it must wait. (i.e.) Rather than engaging in busy waiting, the process can block itself.

· The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state.

· Now control is transferred to **CPU scheduler**, which selects another process to execute. · A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal( ) operation.

· The process is restarted by a wakeup( ) operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

The semaphore definition has been changed to single integer variable to a Structure type with two variables (i.e) Each semaphore has an integer value and list of processes. When a process must wait on a semaphore, it is added to the list of processes. A signal( ) operation removes one process from the list of waiting processes and awakens that process.

```
typedef struct {
int value;
struct process *list;
} semaphore;
…………

wait(semaphore *S)
{
 S->value--;
 if (S->value < 0)
 {
        add this process to S->list;
        block( );
 }
 }
…………..



signal(semaphore *S)
{
 S->value++;
 if (S->value <= 0)
{
        remove a process P from S->list;
        wakeup(P);
}
}
```

· It is critical that semaphore operations wait( ) and signal( ) be executed atomically. This ensures **mutual exclusion** property of Critical Section problem.

· The block( ) operation suspends the process that invokes it.

· The wakeup(P) operation resumes the execution of a blocked process P. · The list of waiting processes can be easily implemented by a link field in each process control block (PCB).

· Each semaphore contains an integer value and a pointer to a list of PCBs. · A FIFO queue is used to add and remove a process from the list. It ensures the Bounded Waiting property.

· The semaphore contains both head and tail pointers to the FIFO queue. **Note:** The above algorithm does not eliminate the **busy waiting** problem completely instead it limits the **busy_waiting** problem to very short amount of time.

Deadlocks and Starvation

A set of processes is in a deadlocked 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 may be either resource acquisition or resource release.

Note: 1. Deadlocks leads to a problem of indefinite blocking and starvation. 2. The implementation of a semaphore with a waiting queue may result in Deadlock.

Example: Consider a system consisting of two processes P0 and P1. Both are accessing two semaphores S and Q. and **S==Q==1**.

P0 P1

```
wait(S); wait(Q);
wait(Q); wait(S);
. .
. .
. .
signal(S); signal(Q);
signal(Q); signal(S);
```

· Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q).

· When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). · Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). · Since these signal( ) operations cannot be executed, *P*0 and *P*1 are deadlocked.

It is a scheduling problem. It occurs only in systems with more than two priorities. Consider there are three processes L, M, H whose order of priorities are given as **L< M < H** and all are wanted to access the resource R.

· When a higher-priority process (H) needs to read or modify kernel data resource R that are currently being accessed by a lower-priority process (L).

· Since kernel data are protected with a lock, the higher-priority process (H) will have to wait for a lower-priority (L) to finish with the resource.

· The situation becomes more complicated if the lower-priority process (L) is preempted in favor of another process (M) with a higher priority.

· Now the higher priority process (H) must have to wait until the process M has to release the lock and then L has to release the lock then only the process H can access the kernel data resource **R**.

· This problem is called **Priority Inversion**.

One solution for priority inversion is that the system will have only two priorities. · These systems solve the priority inversion problem by implementing a **Priority Inheritance Protocol**.

· In this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. · When they are finished, their priorities revert to their original values.

Example:

· A priority-inheritance protocol would allow process **L** to temporarily inherit the priority of process H, thereby preventing process M from preempting process **L's** execution. · When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority.

· Because resource R would now be available, process H will get the resource instead of process M.

Note: This solution is inefficient for systems with more than one priority given for processes.

CLASSIC PROBLEMS OF SYNCHRONIZATION

There are three classic problem that are related synchronization when processes are executing concurrently.

1. The Bounded-Buffer Problem
2. The Readers–Writers Problem
3. The Dining-Philosophers Problem

The Bounded-Buffer Problem

The major problem in Producer-consumer process is The Bounded-Buffer problem. Let the Buffer pool consists of **n** locations, each location stores one item.
The Solution for Producer-Consumer Process can be given as:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Producer process:
```
do {
        /* produce an item in next produced */
         wait(empty); /* If any one location on buffer is empty */
        wait(mutex);

                /* add next produced to the buffer */

        signal(mutex);
        signal(full);
} while (true);
```

Consumer process:
```
do {
        wait(full); /* If any one location in buffer is full */
wait(mutex); /*remove an item from buffer to next consumed */
…………
signal(mutex);
        signal(empty); /* consume the item in next consumed */
} while (true);
```

Explanation of above algorithm

There is one integer variable and three semaphore variables are declared in the definition. · Integer variable n represents the size of the buffer.

· The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. Mutex variable allows only either producer process or consumer process allows to access the buffer at a time.

· The **empty** semaphore counts the no. of free location in buffer, it is initialized to **n**. · The **full** semaphore counts the number of full locations buffers, is initialized to **0**. · **empty=n** and **full=0** represents that initially all locations in the buffer are empty.

Producer process

· **wait(empty):** Any location in the buffer is free, then wait(empty) operation is successful and producer process will put an item into buffer. If wait(empty) is false then the producer process will be blocked.

· **wait(mutex):** Here it allows the producer to use the buffer and produce an item into buffer.

· **signal(mutex):** Buffer will be released by producer process.

· **signal(full):** It indicates one or more item added to the buffer if signal(full) is successful.

Consumer process

· **wait(full):** If wait(full) **< 0** then the buffer is empty and there are no item in the buffer to consume. Hence the consumer process will be blocked. Otherwise the buffer is having some items the consumer process can consume an item.

· **wait(mutex):** It allows consumer to use buffer.

· **signal(mutex):** Buffer is released by consumer.

· **signal(empty):** Consumer consumed an item. Hence one more location in buffer is free.

<u>The Readers–Writers Problem</u>

The Reader-Writer problem occurs in the following situation:

· Consider there is a shared file that is shared by two processes called Reader process and Writer process.

· The reader process will read the data from the file and the writer process will modifies the contents of the file.

· A reader process does not modify the shared file contents. Hence if two reader processes access the shared data simultaneously then there will be no problem of inconsistency. · If a writer process is writing the data and any other process wants to read or write the shared file simultaneously that may result the inconsistency of data. Hence we can’t allow more than one writer process to access the shared file.

· If one writer process is modifying the contents of the shared file, no other reader or writer process will read or write the data of shared file.

· Hence the writer process has exclusive access to the shared file while performing write operation.

· This synchronization problem is referred to as the **Readers–Writers problem**.

Solution to Reader-Writer Problem

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

Code for Reading the data: do

```
{
    wait(mutex);
    read_count++;
    if (read_count == 1)
    wait(rw_mutex);
    signal(mutex);
    . . .
    /* reading is performed */
    . . .
    wait(mutex);
    read_count--;
    if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Coder for Writing data: do

```
{
    wait(rw_mutex);
    . . .
    /* writing is performed */
    . . .
    signal(rw_mutex);
} while (true);
```

· The semaphores mutex and rw_mutex are initialized to 1. read_count is initialized to 0. · The semaphore rw_mutex is common to both reader and writer processes. · The mutex semaphore is used to ensure mutual exclusion when the variable read_count is updated.

· The read_count variable keeps track of how many processes are currently reading the object.

· The semaphore rw_mutex functions as a mutual exclusion semaphore for the writers. · rw_mutex is also used by the first reader process that enters the critical section or last reader process that exits the critical section.

· rw_mutex is not used by readers who enter or exit while other readers are in their critical sections.

Reader-Writer process code Explanation

· Let us consider the process P1 that is executing **wait(mutex)** operation that increments the

read_count to **1** (i.e. now **read_count=1**).

· The **if condition** has been satisfied by P1 and it can invoke **wait(rw_mutex)** and P1 will enter into the critical section and modify (write) the contents of the shared file. · Later P1 executes **signal(rw_mutex)** operation and the control pointer returns to readering process and executes **signal(mutex)**.

· The **signal(mutex)** operation allows another process P2 to enter into reader process and executes **wait(mutex)** and increments **read_count** value by **1** (i.e. now **read_count=2**). · Now P2 fails to satify the **if condition** because the **read_count** value. · Hence P2 can only reads the file but not writes on the file because P1 is still in critical section and reading the updated data.

· If P1 coming out of the critical section and executes **wait(mutex)** and decrements the **read_count** value by **1**. (now **read_count =1**) and executes **signal(mutex)** to allow P2. · If P2 now executes **if condition**, then the condition is satisfied and P2 will enter into writing process and writes the contents of the file.

· This is how the semaphore variable solves the synchronization problem of critical section.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating.

· The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

· In the center of the table is a bowl of rice and the table is laid with five single chopsticks. · When a philosopher thinks, she does not interact with her colleagues. · From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her. The chopsticks are placed between her and her left and right neighbors.

· A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up achopstick that is already in the hand of a neighbor.

· When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks.

· When she is finished eating, she puts down both chopsticks and starts thinking again.

Solution with semaphores:

```
semaphore chopstick[5];
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);

        /* eat for a while */

        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);

        /* think for a while */
} while (true);
```

· Each chopstick is represented with a semaphore and all the elements of chopstick are initialized to 1.

· A philosopher tries to grab a chopstick by executing a wait( ) operation on that semaphore.

· A philosopher releases chopsticks by executing the signal( ) operation on the appropriate semaphores.

Note: Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.

1. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
2. All the elements of chopstick will now be equal to 0.
3. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Possible remedies to the Deadlock problem:

· Allow at most four philosophers to be sitting simultaneously at the table. · Allow a philosopher to pick up her chopsticks only if both chopsticks are available. To do this, she must pick them up in a critical section.

· An Asymmetric solution will be used, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

PROBLEM WITH SEMAPHORES

Semaphores provide a convenient and effective mechanism for process synchronization but using semaphores incorrectly can result in timing errors that are difficult to detect. These errors happen only if particular execution sequences take place and these sequences do not always occur.

Examples: 1

Suppose that a process interchanges the order in which the wait( ) and signal( ) operations on the semaphore mutex are executed, resulting in the following execution: signal(mutex);

```
...
critical section
...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.

Example:2

Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes wait(mutex);

```
...
critical section
...
wait(mutex);
```

In this case, a deadlock will occur.

Example:3

Suppose that a process omits the wait(mutex) or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

MONITORS

Monitors are developed to deal with semaphore errors. Monitors are high-level language synchronization constructs.

· A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.

· The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. **Monitor Syntax:**

```
monitor monitor_ name
{
 /* shared variable declarations */
        function P1 ( . . . )
        {
        . . .
        }
        function P2 ( . . . )
        {
        . . .
        }
        . . .
        function Pn ( . . . )
        {
        . . .
        }
         initialization code ( . . . )
        {
        . . .
        }
}
```

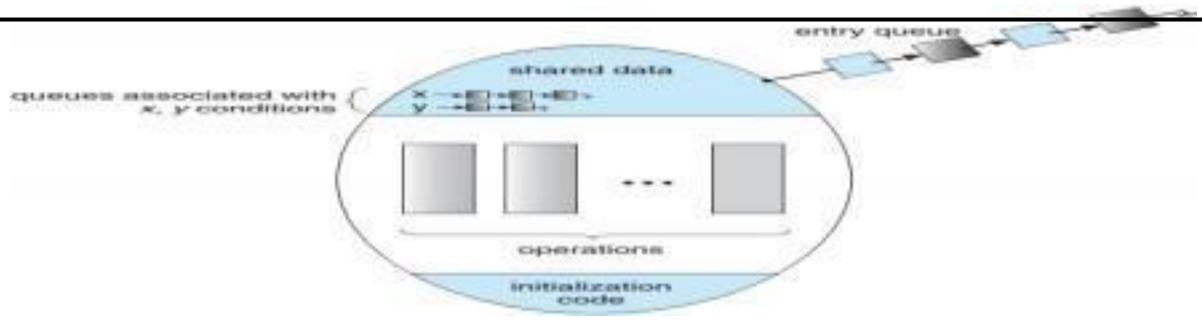
The representation of a monitor type cannot be used directly by the various processes. · A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

· Similarly, the local variables of a monitor can be accessed by only the local functions. · The monitor construct ensures that only one process at a time is active within the monitor. · the programmer does not need to code this synchronization constraint explicitly

To solve the problem of synchronization problem along with monitors a construct called **Condition** has been implemented. Condition construct defines one or more variables of type **"condition".** The syntax of condition is:

condition x,y;

The only operations that can be invoked on a condition variable are wait( ) and signal( ).
x.wait( ): Process invokes x.wait( ) is suspended until another process invokes **x.signal( )**.
x.signal( ): It resumes exactly one suspended process.



Dining-Philosophers Solution Using Monitors

By using monitor we can have a deadlock free solution for dining philosopher's problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

There are 3 states for each philosopher: Thinking, Hungry, Eating.

enum { THINKING, HUNGRY, EATING } state[5];

Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating: **(state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING)**

We also need to declare: **condition self[5];**

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Solution through monitor:

```
monitor Dining_Philosophers {
 enum {THINKING, HUNGRY, EATING} state[5];
 condition self[5];
     void pickup(int i)
     {
         state[i] = HUNGRY;
         test(i);




         if (state[i] != EATING)
         self[i].wait( );
     }
   void putdown(int i)
   {
       state[i] = THINKING;
       test((i + 4) % 5);
       test((i + 1) % 5);
   }
   void test(int i)
   {
       if ((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING))
       {
        state[i] = EATING;
```

```
                        self[i].signal( );
                }
                }
        initialization_code( ) {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
        }
        }
```

The distribution of the chopsticks is controlled by the monitor **Dining_Philosophers**. · Each philosopher before starting to eat, must invoke the operation pickup( ). This act may result in the suspension of the philosopher process.
· After the successful completion of the pickup( ) operation, the philosopher may eat. · After pickup( ) operation the philosopher invokes the putdown( ) operation.

Thus, philosopher i must invoke pickup( ) & putdown( ) operations in following sequence:

DiningPhilosophers.pickup(i);

Eat

DiningPhilosophers.putdown(i);

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Implementing a Monitor Using Semaphores

Monitor uses three variables: semaphore mutex=1; semaphore next=0; int next_count; · For each monitor, a semaphore mutex is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor. · The signaling processes can use **next** variable to suspend themselves, because signaling process must wait until the resumed process exit or leave.
· **next count** is an integer variable used to count the number of processes suspended on next.

Each external function F is replaced by:

```
wait(mutex);
...
body of F
...
if (next count > 0)
signal(next);
else
signal(mutex);
```

The above code ensures the mutual exclusion property.

Implementing condition variables

For each condition x, we introduce a semaphore x_sem and an integer variable x_count. semaphore x_sem=0;

```
int x_count=0;
```

The operation x.wait( ) can now be implemented as:

```
x_count++;
```

```
if (next_count > 0)
signal(next);
else
signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal( ) can be implemented as:

```
if (x_count > 0)
{
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Resuming Processes within a Monitor

Consider a situation, if several processes are suspended on condition x, and an x.signal( ) operation is executed by some process, then how do we determine which of the suspended processes should be resumed next?

We have two solutions: FCFS and Priority mechanism.

1. We use first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first.
2. In priority mechanism the conditional-wait construct can be used.
   - **x.wait(c);**

   where c is an integer priority numbers that is evaluated when the wait( ) operation is executed. The **c** value is then stored with the name of the process that is suspended. When x.signal( ) is executed, the process with the smallest priority number is resumed next.

Consider the below code the **Resource_Allocator monitor** that controls the allocation of a single resource among competing processes.

```
monitor Resource_Allocator
{
    boolean busy;
    condition x;
    void acquire(int time)
    {
        if (busy)
        x.wait(time);
        busy = true;
    }
    void release( )
    {
        busy = false;
        x.signal( );
    }
```

```
                initialization_code( )
                {
                 busy = false;
                 }
            }
```

· Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource.

· Monitor allocates the resource to the process that has the shortest time-allocation request.

The process that needs to access the resource must observe the following sequence:

```
R.acquire(t); /* R is an instance of type ResourceAllocator */
 ...
access the resource;
 ...
R.release( );
```

Problems with monitor

1. A process might access a resource without first gaining access permission to the resource.
2. A process might never release a resource once it has been granted access to the resource.
3. A process might attempt to release a resource that it never requested. 4. A process might request the same resource twice (without first releasing the resource).

DEADLOCKS

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 resource release.

SYSTEM MODEL

A system consists of a finite number of resources to be distributed among a number of competing processes.

Resources are categorized into two types: Physical resources and Logical resources ·
Physical resources: Printers, Tape drives, DVD drives, memory space and CPU cycles ·
Logical resources: Semaphores, Mutex locks and files.

Each resource type consists of some number of identical instances. (i.e.) If a system has two CPU's then the resource type CPU has two instances.

A process may utilize a resource in the following sequence under normal mode of operation:

1. **Request**: The process requests the resource. If the resource is being used by another process then the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. **Use**: The process can operate on the resource.

Example: If the resource is a printer, the process can print on the printer. 3. **Release**: The process releases the resource.

System calls for requesting and releasing resources:
· Device System calls: request( ) and release( )
· Semaphore System calls: wait( ), signal( )
· Mutex locks: acquire( ), release( ).
· Memory System Calls: allocate( ) and free( )
· File System calls: open( ), close( ).

A **System Table** maintains the status of each resource whether the resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. 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.

FOUR NECESSARY CONDITIONS OF DEADLOCK

A deadlock situation can arise if the following **4** conditions hold simultaneously in a system:

1. **Mutual exclusion**. Only one process at a time can use the resource. If other process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**. If a process holding a resource and the resource cannot be preempted until the process has completed its task.
4. **Circular wait**. A set *{P*0, *P*1, ..., *Pn}* of waiting processes must exist such that *P*0 is waiting for a resource held by *P*1, *P*1 is waiting for a resource held by *P*2, ..., *Pn*−1 is waiting for a resource held by *Pn* and *Pn* is waiting for a resource held by *P*0.

RESOURCE-ALLOCATION GRAPH

The resource allocation graph is used for identification of deadlocks in the system. A **System Resource-Allocation Graph G={V,E}** is a directed graph that consists of a set of vertices *V* and a set of edges *E*.

The set of vertices *V* is partitioned into two types of nodes: Processes and Resources.

1. **Process set** P= *{P*1, *P*2, ..., *Pn}* consisting of all the active processes in the system.
2. **Resource set** R= *{R*1, *R*2, ..., *Rm}* consisting of all resource types in the system.

The set of Edges E is divided into types: Request Edge and Assignment Edge. 1. **Request Edge (Pi → Rj):** It signifies that process *Pi* has requested an instance of resource type *Rj* and **Pi** is currently waiting for the resource **Rj**.
2. **Assignment edge (Rj → Pi):** It signifies that an instance of resource type *Rj* has been allocated to process *Pi*.

Processes can be represented in **Circles** and **Resources** can be represented in **Rectangles**. **Instance** of resource can be represented by a **Dot**.

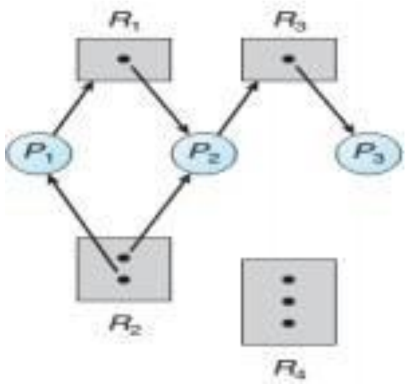
Note:

1. When process *Pi* requests an instance of resource type *Rj*, a request edge is inserted in the Resource-allocation graph.
2. When this request can be fulfilled, the request edge is transformed to an assignment edge.

3. When the process no longer needs access to the resource, it releases the resource and the assignment edge is deleted.

Resource allocation graph shows three situations:
1. Graph with No deadlock
2. Graph with a cycle and deadlock
3. Graph with a cycle and no deadlock

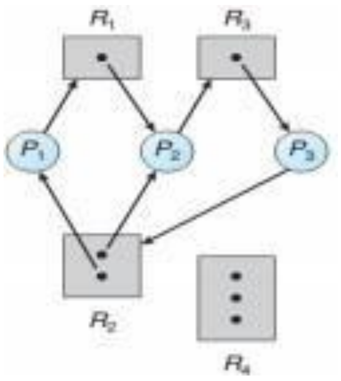
Resource Allocation Graph without Deadlock
The below graph consists of three sets: Process **P**, Resources **R** and Edges **E**.



· Process set P= {P1, P2, P3}.
· Resources set R= {R1, R2, R3, R4}.
· Edge set E= E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}. Resource type R1 and R3 has only one instance and R2 has two instances and R4 has three instances.

The Resource Allocation Graph depicts that:
· **R2 → P1, P1 → R1:** P1 is holding an instance of resource type R2 and is waiting for an instance of R1.
· **R1 → P2, R2 → P2, P2 → R3:** Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
· **R3 → P3:** Process P3 is holding an instance of R3.
The above Resource allocation graph does not contain any cycle then there is no process in the system is deadlocked.
Note:
1. If each resource type has exactly one instance then a cycle implies a deadlock. 2. If each resource type has several instances then a cycle does not necessarily imply that a deadlock has occurred.

Resource Allocation Graph with a Cycle and Deadlock



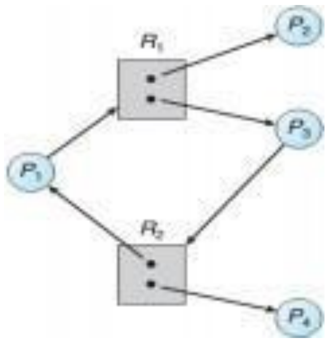
Consider the above graph, with processes and Resources and have some edges: P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

· Process P2 is waiting for the resource R3, which is held by process P3. · Process P3 is waiting for either process P1 or process P2 to release resource R2. · Process P1 is waiting for process P2 to release resource R1.

Hence the Processes *P*1, *P*2 and *P*3 are deadlocked.

Resource Allocation Graph with a Cycle and No Deadlock



The graph has a cycle: P1 → R1 → P3 → R2 → P1.

· This cycle does not lead to deadlock, because the process P4 and P2 is not waiting for any resource.

· Process *P*4 may release its instance of resource type *R*2. That resource can then be allocated to *P*3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

The Deadlock can be handled by 3 methods:

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection and Recovery

DEADLOCK PREVENTION

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. (i.e.) Deadlock can be prevented if any of Mutual Exclusion, Hold and wait, No preemption and Circular wait condition cannot hold.

Mutual Exclusion

The mutual exclusion condition must hold when at least one resource must be non-sharable. · We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources by default are nonsharable.

Example 1: A mutex lock cannot be simultaneously shared by several processes. Example 2: Printer is a resource where only one process can use it.

· Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Example: Read-only files.

· If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. · **Protocol 1:** Each process can request the resources and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

Example: Consider a process that copies data from a **DVD drive** to a file on **Hard disk**, sorts the file and then prints the results to a **Printer**.

If all resources must be requested at the beginning of the process, then the process must initially request the **DVD drive**, **disk file** and **Printer**. It will hold the printer for its entire execution, even though it needs the printer only at the end.

· **Protocol 2:** A process can be allowed to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Example: Consider a process that copies data from a **DVD drive** to a file on **Hard disk**, sorts the file and then prints the results to a **Printer**.

The process to request initially **only** the **DVD drive** and **Hard disk** file. It copies from the **DVD drive** to the **Hard disk** and then releases both the DVD drive and the disk file. The process must then request the **Hard disk** file and the **Printer**. After copying the disk file to the printer, it releases these two resources and terminates.

Problem: Starvation and Low Resource utilization

· Resource utilization is low, since resources may be allocated but unused for a long period. · A process that needs several resources may have to wait indefinitely leads to starvation.

No Preemption

To ensure that No preemption condition does not hold, we can use the following protocol: · If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted (i.e.) resources are implicitly released.

· The preempted resources are added to the list of resources for which the process is waiting.

· The process will be restarted only when it can regain its old resources as well as the new resources that it is requesting.

Note: This protocol is often applied to resources whose state can be easily saved and restored later such as CPU registers and memory space. It cannot be applied to resources such as mutex locks and semaphores.

Circular Wait

One way to ensure that circular wait condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Consider the set of resource types R={R1, R2, ..., Rm} and **N** be the set of natural numbers. we define a one-to-one function **F: R → N**.

· The function assigns each resource type to a unique integer number, which allows us to compare two resources and to determine whether one resource precedes another resource in our ordering.

Example: If the set of resource types R includes tape drives, disk drives and printers, then the function **F: R → N** might be defined as follows:

F (Tape drive) = 1 (F: Tape drive → 1)
F (Disk drive) = 5 (F: Disk drive → 1)
F (Printer) = 12 (F: Printer → 1)

We can now consider the following protocol to prevent deadlocks:

· Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type Ri. · After that, the process can request instances of resource type Rj iff **F(Rj) > F(Ri)** · Example: A process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

· Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that **F(Ri) ≥ F(Rj)**.

Note: If several instances of the same resource type are needed, a **single** request for all of them must be issued.

Disadvantage of Deadlock Prevention

Deadlock-prevention algorithms leads to low resource utilization and the system throughput will be reduced.

DEADLOCK AVOIDANCE

In Deadlock avoidance the processes first informs the operating system about their maximum allocation of resources to be requested and used during its life time.

· With this information, the operating system can decide for each request whether the resource will be granted immediately or the process should wait for resources. · To take this decision about decision about the resource allocation, the operating system must consider the resources currently available, the resources currently allocated to each process and the future requests and releases of each process.

Algorithms for Deadlock Avoidance

A deadlock-avoidance algorithm dynamically examines the **Resource-Allocation State** to ensure that a circular-wait condition can never exist.

The Resource Allocation **State** is defined by the number of available resources and allocated resources and the maximum demands of the processes.

There are three algorithms are designed for deadlock avoidance:

1. Safe State
2. Resource Allocation Graph Algorithm
3. Bankers Algorithm

Safe State Algorithm

If the system can allocate resources to each process up to its maximum in some order and still avoid a deadlock then the state is called Safe state.

· A system is in a safe state only if there exists a **Safe sequence**.

· A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state, if for each process **Pi**, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i.

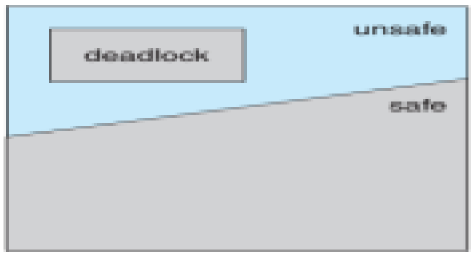
· In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished.

· When Pj have finished its task, Pi can obtain all of its needed resources and after completing its designated task Pi can return its allocated resources and terminate. · When P_i terminates, P_{i+1} can obtain its needed resources and so on.

· If no such sequence exists, then the system state is said to be unsafe.

Note:

1. A safe state is not a deadlocked state and a deadlocked state is an unsafe state. 2. An unsafe state ***may*** lead to a deadlock but **not all** unsafe states are deadlocks. 3. As long as the state is safe, the operating system can avoid unsafe and deadlocked states. 4. In an unsafe state, operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs. Behavior of the processes controls unsafe states.



Example: Consider a system with 12 magnetic tape drives and 3 processes: *P*0, *P*1 and *P*2.

Process	Maximum Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

The above table describes as follows:

· Process **P0** requires 10 tape drives, **P1** needs 4 tape drives and **P2** need 9 tape drives. · At time t0, process P0 is holding 5 tape drives, P1 and P2 is holding 2 tape drives each. · Now there are 3 free tape drives.

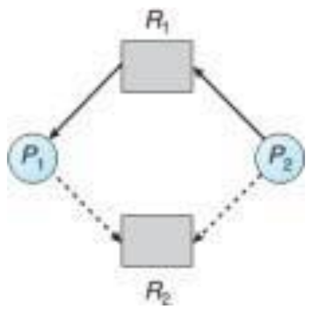
At time **t0**, the system is in a safe state. <*P*1, *P*0, *P*2> sequence satisfies the safety condition.
· Process *P*1 can immediately be allocated all its tape drives and then return all 4 resources. (i.e.) P1 currently holding 2 tape drives and out of 3 free tape drives 2 tape drives will be given to P1. Now P1 is having all 4 resources. Hence P1 will use all of its resources and after completing its task P1 releases all 4 resources and then returns to the system. Now the system is having **5 available** tape drives.

· Now process P0 needs 5 tape drives and the system has 5 available tape drives. Hence **P0** can get all its tape drives and it reaches its maximum 10 tape drives. After completing its task P0 returns the resources to the system. Now system has 10 available tape drives.

· Now the process P2 needs 7 additional resources and system have 10 resources available. Hence process *P*2 can get all its tape drives and return them. Now the system will have all 12 tape drives available.

Problem: Low Resource utilization
If a process requests a resource that is currently available, it may still have to wait. Hence there exist a low resource utilization is possible.

Resource-Allocation-Graph Algorithm
In this algorithm we use three edges: request edge, assignment edge and a **claim edge**. · Claim edge **Pi** → **Rj** indicates that process **Pi** may request resource **Rj** at some time in the future.
· Claim edge resembles a request edge in direction but is represented by dashed line. · When process **Pi** requests resource **Rj**, the claim edge **Pi** → **Rj** is converted to a request edge.
· When a resource **Rj** is released by **Pi**, the assignment edge **Rj** → **Pi** is reconverted to a claim edge **Pi** → **Rj**.
· The resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.



Now suppose that process Pi requests resource Rj.
· The request can be granted only if converting the request edge **Pi** → **Rj** to an assignment edge **Rj** → **Pi** does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm.
· If no cycle exists, then the allocation of the resource will leave the system in a safe state. · If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process Pi will have to wait for its requests to be satisfied.

Example: consider the above resource-allocation graph. Suppose that P2 requests R2. · R2 is currently free still we cannot allocate it to P2, since this will create a cycle in graph. · A cycle indicates that the system is in an unsafe state.
· If P1 requests R2 and P2 requests R1, then a deadlock will occur.
Problem: The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

BANKER's ALGORITHM
Banker's algorithm is used in a system with multiple instance of each resource type. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Banker's algorithm uses two algorithms:

1. Safety algorithm
2. Resource-Request algorithm

Process of Banker's algorithm:

· When a new process enters the system, the process must declare the **Maximum** number of instances of each resource type that it may need.

· The **Maximum** number may not exceed the **Total** number of resources in the system. · When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.

· If the system is in safe state then the resources are allocated.

· If the system is in unsafe state then the process must wait until some other process releases enough resources.

Data structures used to implement the Banker's algorithm

Consider the system with **n** number of processes and m number of resource types: · **Available$_m$:** A vector of length m indicates the number of available resources of each type.

· **Max$_{n \times m}$:** An n × m matrix defines the maximum demand of each process. · **Allocation $_{n \times m}$:** An n × m matrix defines the number of resources of each type currently allocated to each process.

· **Need $_{n \times m}$:** An n × m matrix indicates the remaining resource need of each process.

$$\textbf{Need[i][j] = Max[i][j]−Allocation[i][j].}$$

· **Available[j] = k** means then k instances of resource type Rj are available. · **Max[i][j] = k** means process Pi may request at most k instances of resource type Rj.

· **Allocation[i][j]=k** means process Pi is currently allocated k instances of resource type Rj. · **Need[i][j]=k** means process Pi may need k more instances of resource type Rj to complete its task.

Each row in the matrices **Allocation $_{n \times m}$** and **Need $_{n \times m}$** are considered as vectors and refer to them as **Allocation$_i$** and **Need$_i$.**

· The vector **Allocation$_i$** specifies the resources currently allocated to process Pi. · The vector **Need$_i$** specifies the additional resources that process Pi may still request to complete its task.

Safety algorithm

Safety algorithm finds out whether the system is in safe state or not. The algorithm can be described as follows:

```
1. Let Work and Finish be vectors of length m and n, respectively. We initialize
                         Work = Available
                    Finish[i] = false for i = 0, 1, ..., n − 1.
2. Find an index i such that both
                    Finish[i] == false
                    Need_i ≤ Work
   If no such i exists, go to step 4.
3. Work = Work + Allocation_i
   Finish[i] = true
   Go to step 2.
4. If Finish[i] == true for all i, then the system is in a safe state.
```

Note: To determine a safe state, this algorithm requires an order of **m×n^2**operations.

Resource-Request Algorithm
This algorithm determines whether requests can be safely granted.
Let **Request$_i$** be the request vector for process Pi. If **Request$_i$ [j] == k**, then process Pi wants k instances of resource type Rj.

When a request for resources is made by process Pi, the following actions are taken: 1. If **Request$_i$** ≤ **Need$_i$**, go to step 2.
Otherwise, raise an error condition, since the process has exceeded its maximum claim. 2. If **Request$_i$** ≤ **Available,** go to step 3.
Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = **Available**– **Request$_i$** ;
Allocation$_i$ = **Allocation$_i$** + **Request$_i$**;
Need$_i$ = **Need$_i$** – **Request$_i$** ;

4. If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources.
If the new state is unsafe, then Pi must wait for **Request$_i$** and the old resource-allocation state is restored.

Example for Banker's Algorithm
Consider a system with 5 processes: **P0, P1, P2, P3, P4** and 3 resource types **A, B** and **C** with **10, 5, 7** instances respectively. (i.e.) . Resource type **A=10, B= 5** and **C=7** instances. Suppose that, at time T0, the following snapshot of the system has been taken:

	Allocation	Max	Available
Process	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

The Available vector can be calculated by subtractring total no of resources from the sum of resources allocated to each process.

Available resources of A= Total resources of A – Sum of resources allocated to Process P1 to P4

The Need matrix can be obtained by using **Need[i][j] = Max[i][j]−Allocation[i][j]**

	Max	Allocation	**Need**
	A B C	A B C	**A B C**
P0	7 5 3	0 1 0	**7 4 3**
P1	3 2 2	2 0 0	**1 2 2**
P2	9 0 2	3 0 2	**6 0 0**
P3	2 2 2	2 1 1	**0 1 1**
P4	4 3 3	0 0 2	**4 3 1**

By using the banker's algorithm we can decide whether the state is safe or not. After solving the above problem by using bankers algorithm we will get to a safe state with safe sequence <P1,P3,P4,P0,P2>.
Now we get a safe state, the resources will be granted immediately for requested process P1.

DEADLOCK DETECTION ALGORITHM

If a system does not employ either a Deadlock-Prevention or a Deadlock-Avoidance algorithm then a deadlock situation may occur. In this environment, the system may provide: · An algorithm that examines the state of the system to determine whether a deadlock has occurred

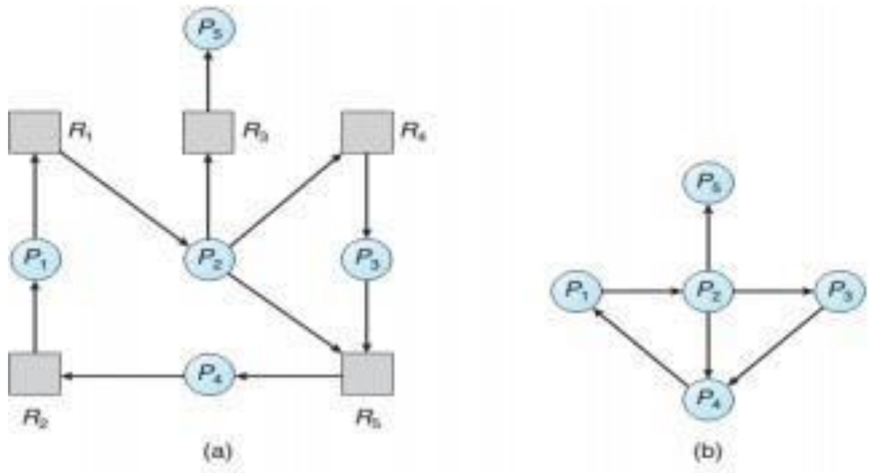
· An algorithm to recover from the deadlock.

Deadlock Detection in Single Instance of Each Resource Type

If all resources have only a single instance then we can define a Deadlock-Detection algorithm that uses a variant of the resource-allocation graph called a **wait-for** graph. We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

· An edge from **Pi** to **Pj** in a wait-for graph implies that process **Pi** is waiting for process **Pj** to release a resource that **Pi** needs.

· An edge **Pi** → **Pj** exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges **Pi** → **Rq** and **Rq** → **Pj** for some resource **Rq** .



· In above figure we present a resource-allocation graph and the corresponding wait-for graph. A deadlock exists in the system if and only if the wait-for graph contains a cycle. · To detect deadlocks, the system needs to **maintain** the wait-for graph and periodically **invoke an algorithm** that searches for a cycle in the graph.

· An algorithm to detect a cycle in a graph requires an order of $\mathbf{n}^2$operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

We will implement a Deadlock Detection algorithm that is similar to the Banker's algorithm. The data structures used in Deadlock Detection algorithm is:

· **Available:** A vector of length **m** indicates the number of available resources of each type. ·

Allocation: An **n × m** matrix defines the number of resources of each type currently allocated to each process.

· **Request:** An **n × m** matrix indicates the current request of each process. If **Request[i][j]==k**, then process Pi is requesting **k** more instances of resource type **Rj**.

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

```
1. Let Work and Finish be vectors of length m and n, respectively. We Initialize
            Work = Available. For i = 0, 1, ..., n–1.
            if Allocation_i != 0, then Finish[i] = false.
            Otherwise, Finish[i] = true.
2. Find an index i such that both
        a. Finish[i] == false
        b. Request_i ≤ Work
   If no such i exists, go to step 4.
3. Work = Work + Allocation_i
        Finish[i] = true
        Go to step 2.
4. If Finish[i] == false for some i, 0 ≤ i < n, then the system is in a deadlocked state.
   Moreover, if Finish[i] == false, then process Pi is deadlocked.
```

This algorithm requires an order of **m × n**^{2}operations to detect whether the system is in a deadlocked state.

Example:

Consider a system with 5 processes: **P0, P1, P2, P3, P4** and 3 resource types **A, B** and **C** with **10, 5, 7** instances respectively. (i.e.) . Resource type **A=7, B= 2** and **C=6** instances. Suppose that, at time T0, we have the following resource-allocation state:

	Allocation	Request	**Available**
	A B C	A B C	**A B C**
P0	0 1 0	0 0 0	**0 0 0**
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Initially the system is not in Deadlock State. If we apply the Deadlock Detection algorithm we will find the sequence < P0, P2, P3, P1, P4 > results in **Finish**[i] == true for all i. The system is in safe state hence there is no deadlock.

RECOVERY FROM DEADLOCK

There are two options for breaking a deadlock.

1. Process termination
2. Resource Preemption

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

· **Abort all Deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time and the results of these partial computations must be discarded and probably will have to be recomputed later.

· **Abort one process at a time until the Deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Many factors may affect which process is chosen for preempting
includes: 1. Priority of the process.
2. How long the process has computed and how much longer the process will compute before completing its designated task.
3. How many and what types of resources the process has used.
4. How many more resources the process needs in order to complete.
5. How many processes will need to be terminated?
6. Whether the process is Interactive or Batch.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. There are 3 issues related to Resource Preemption:

1. **Selecting a victim**. As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback**. If we preempt a resource from a process then the process cannot continue with its normal execution. It is missing some needed resource. We must do total roll back of the process and restart it from that state: abort the process and then restart it.
3. **Starvation**. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? In a system where victim selection is based on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its task which leads to starvation. Hence we must ensure that a process can be picked as a victim only a finite number of times. The solution is to include the number of rollbacks in the cost factor.