DATE :

Unit - III

* Concurrency : Mutual Exclusion And Synchronization *

The central themes of operating system design are all concerned with the management of processes and threads.

* Multiprogramming –

The management of multiple processes within a uniprocessor system.

* Multiprocessing –

The management of multiple process within a multiprocessor.

* Distributed processing –

The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Fundamental to all of these areas, and fundamental to OS design, is concurrency. Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files and I/O access), synchronisation of the activities of multiple processes and allocation of processor time to processes.

* Some key terms related to Concurrency *

Atomic operation –

A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.

Critical section –

A section of code within a process that requires access to shared resources and that must not be

executed while another process is in a corrosponding section of code.

Deadlock -

A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

Livelock -

Mutual Exclusion -

The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

Race Condition -

A situation in which multiple threads or processes read & write a shared data item and the final results depends on the relative timing of their execution.

Starvation -

A situation in which a runnable process is over-looked indefinitely by the scheduler, although it is able to proceed, it is never chosen.

* Process Interaction *

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence.

* Competition among Processes for Resources -

In the case of competing processes three control problems must be faced. First is the need for mutual exclusion. Suppose two or more processes require access to a single nonsharable resource, such as printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data and/or receiving

data. We will refer to such a resource as a critical resource, and the portion of the program that uses it a critical section of the program.

* Illustration of Mutual Exclusion *

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

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

...

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

* Requirements for Mutual Exclusion *

Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1] Mutual exclusion must be enforced. Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.

2] A process that halts in its noncritical section must do so without interfacing with other processes.

3] It must not be possible for a process requiring access do a critical section to be delayed indefinately: no deadlock or starvation.

4] When no process is in critical section, any process that requests entry to its critical section must be permitted to enter without delay.

5] No assumptions are made about relative process speeds or number of processors.

6] A process remains inside its critical section for a finite time only.

* Mutual Exclusion: Hardware Support *

Interrupt Disabling –

In a uniprocessor system, concurrent processes can not have overlapped execution; they can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted. Therefore to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted.

A process can then enforce mutual exclusion in the following way –

```
while (true) {
    /* disable interrupts */;
```

```
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

* Special Machine Instructions –

In a multiprocessor configuration, several processors share access to a common main memory. In this case, there is not a master/slave relationship; rather the processors behave independantly in a peer relationship. There is no interrupt mechanism between processors on which mutual exclusion can be based.

* Compare and Swap Instruction –

The compare & swap instruction, also called a compare & exchange instruction, can be defined as follows.

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

* Exchange Instruction –

The exchange instruction can be defined as follows;

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

```
    memory = register;
    register = temp;
}
```

The instruction exchanges the contents of a register with that of a memory location. Both the Intel IA-32 architecture (Pentium) and the IA-64 architecture (Itanium) contain an XCHG instruction.

* Semaphores *

* Common Concurrency Mechanism —

1) Semaphore —

An integer value used for signaling among processes. only three operations may be performed on a semaphore, all of which are atomic; initialize, decrement and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore.

2) Binary Semaphore —

A semaphore that takes on only the values 0 and 1.

3) Mutex —

Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).

The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals. For signaling

special variables called semaphores are used. To transmit a signal via semaphore s, a process executes a premitive semSignal(s). To receive a signal via semaphore s, a process executes the primitive semWait(s); if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined.

1] A semaphore may be initialized to a nonnegative integer value.

2] The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.

3] The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

* A defination of Semaphore Primitives -

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait (semaphore s)
{
    s.count --;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
```

```
void semSignal (semaphore s)
{
    s.count ++;
    if (s.count <= 0) {
     /* remove a process P from s.queue */;
     /* place process P on ready list */;
     }
}
```

* A defination of Binary semaphore primitives -

```
struct binary - semaphore {
        enum {zero, one} value;
        queueType queue;
};
void semWait B (binary - semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void sem signal B (semaphore s)
{
    if (s.queue is empty ( )}
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

The two types of semaphores, the nonbinary semaphore is often referred to as either a counting semaphore or a general semaphore.

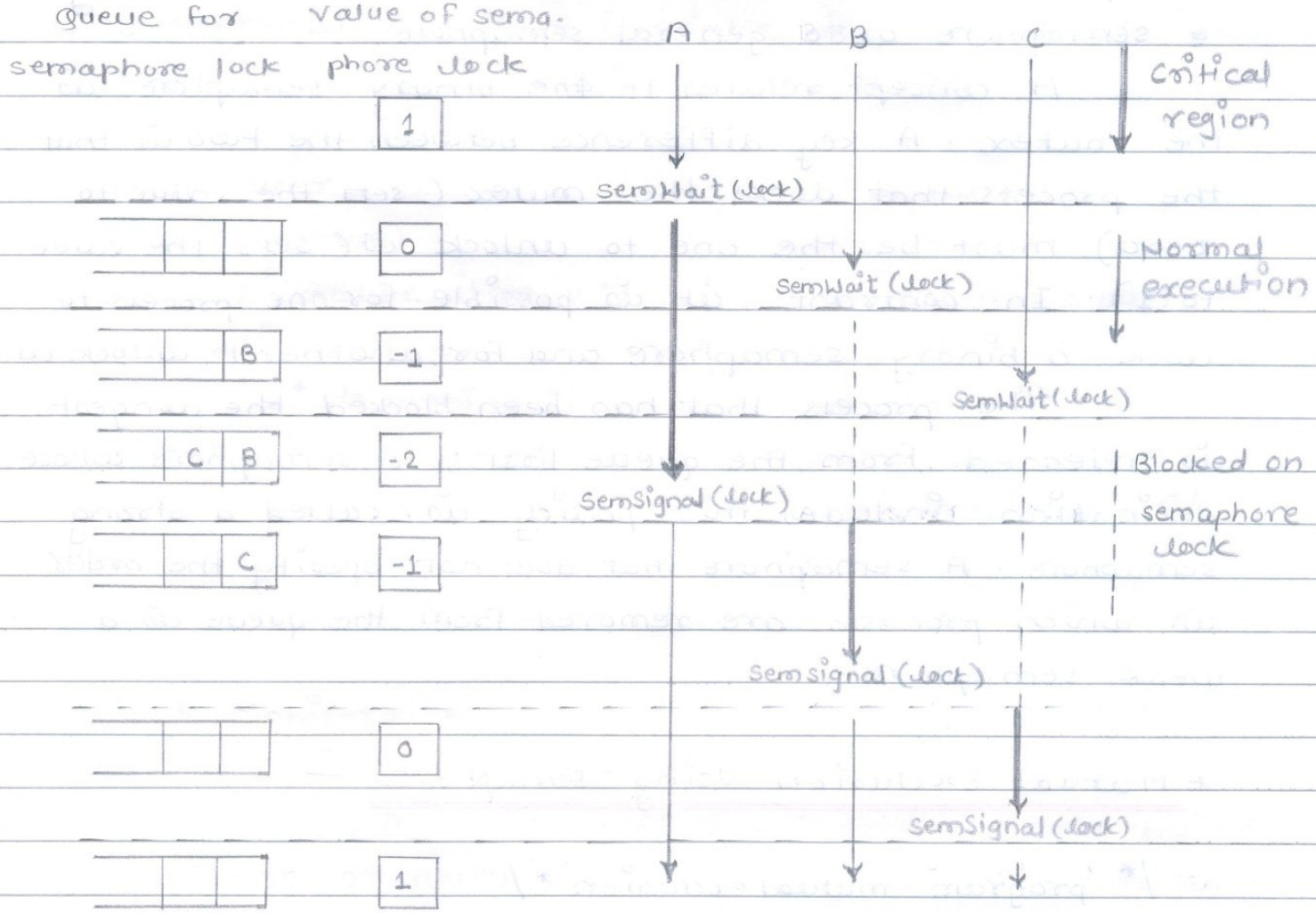
A concept related to the binary semaphore is the mutex. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.

The process that has been blocked the longest is released from the queue first; a semaphore whose defination includes this policy is called a strong semaphore. A semaphore that does not specify the order in which processes are removed from the queue is a weak semaphore.

* Mutual Exclusion Using semaphores —

```
/* program mutualexclusion */
const int n = /* numbers of proceses */;
semaphore s = 1;
void P (int i)
{
  while (true) {
     semWait (s);
     /* critical section */;
     semSignal (s);
     /* remainder */;
  }
}
void main ( )
{
   parbegin ( P(1), P(2), ...., P(n));
}
```

* Mutual Exclusion *



* Note that normal execution can proceed in parallel but that critical regions are serialized.

Fig – Processes Accessing Shared Data Protected by a Semaphore

* The Producer / consumer Problem –

The general statement is this : There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a signal consumer that is taking items out of the buffer one at a time.

```
producer :
while (true) {
    /* produce item v */;
    b [in] = v;
    in ++;
}
```

```
consumer :
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out ++;
    /* consumer item w */;
}
```

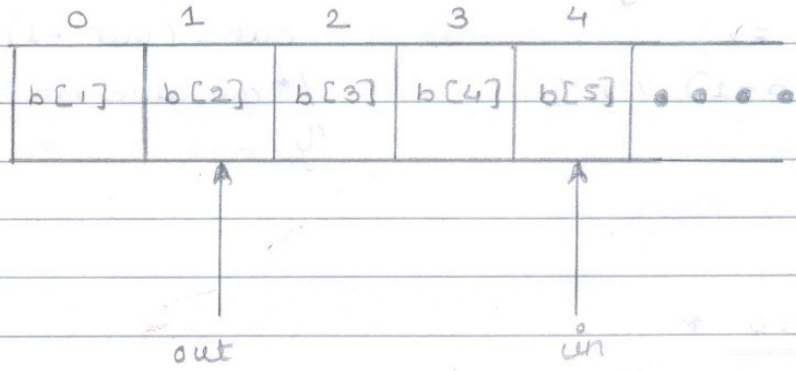
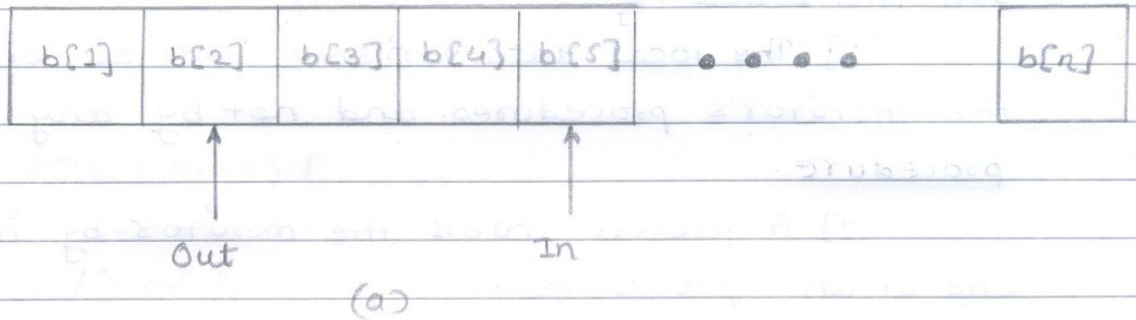
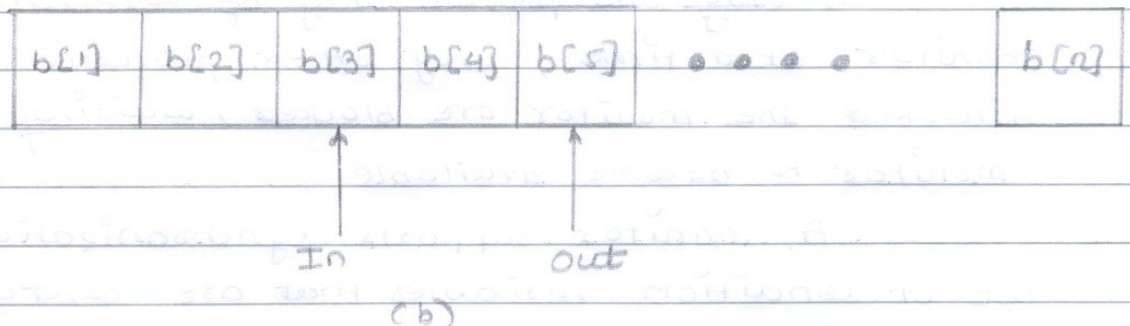


Fig - Infinite Buffer for the Producer/Consumer Problem



(a)



(b)

Fig - Finite Circular Buffer for the Producer / consumer problem

The producer and consumer functions can be expressed as follows (variable in and out are initialized to 0 and n is the size of the buffer):

```
producer:
while (true) {
   /* produce item v */
   while ((in + 1) % n == out)
      /* do nothing */;
   b[in] = v;
   in = (in + 1) % n;
}
```

```
consumer:
while (true) {
   while (in == out)
      /* do nothing */;
   w = b[out];
   out = (out + 1) % n;
   /* consume item w */;
}
```

* Monitors *

Monitor with signal –

A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data. The chief characteristics of a monitor are the following:

1] The local data variables are accessible only by the monitor's procedures and not by any external procedure.

2] A process enters the monitor by invoking one of its procedures.

3] Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.

A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors,

which are operated on by two functions.

1) cwait (c) :-

Suspend execution of the calling process on condition c. The monitor is now available for use by another process.

2) c signal (c) :-

Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

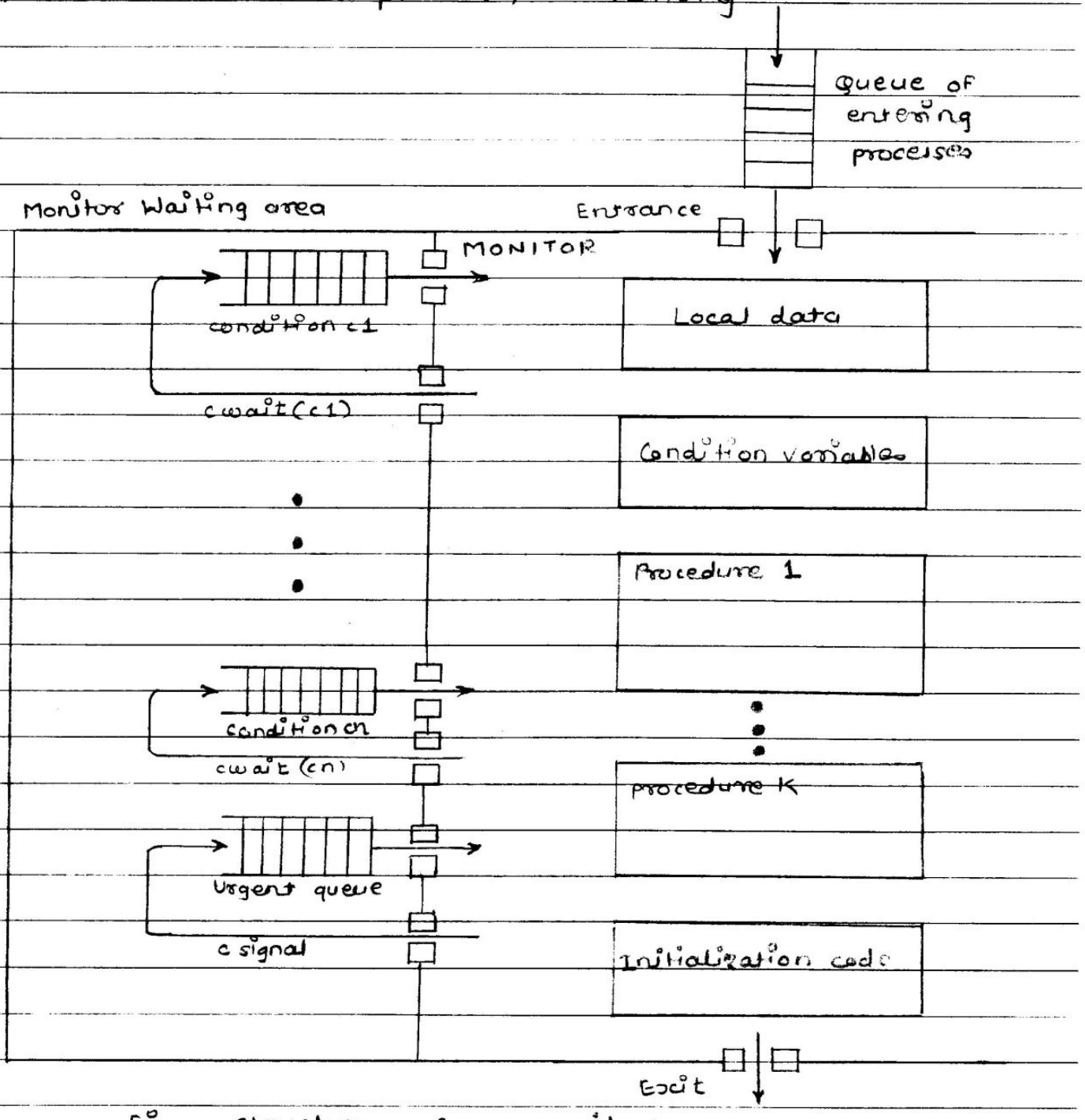


Fig — Structure of a monitor

* Readers / Writers Problem —

The readers / writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

The conditions that must be satisfied are as follows:

1] Any number of readers may simultaneously read the file.

2] Only one writer at a time may write to the file.

3] If a writer is writing to the file, no reader may read it.

* Concurrency : Deadlock And Starvation *

* Principles of Deadlock —

Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked processes in the set. Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case.

* The conditions for Deadlock —

Three conditions of policy must be present for a deadlock to be possible.

1) Mutual Exclusion –

Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.

2) Hold and wait –

A process may hold allocated resources while awaiting assignment of other resources.

3) No preemption –

No resource can be forcibly removed from a process holding it.

4) Circular wait –

A closed chain of processes exists, such that each process holds atleast one resource needed by the next process in the chain.

Possibility of Deadlock	Existence of deadlock
1] Mutual exclusion	1] Mutual exclusion
2] No preemption	2] No preemption
3] Hold and wait	3] Hold and wait
	4] Circular wait

* Deadlock prevention *

Mutual Exclusion —

In general, the first of the four listed conditions can not be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion may must be supported by the OS. Some resources,

such as files, may allow multiple accesses for reads but only exclusive access for writers. Even in this case, deadlock can occure if more than one process requires write permission.

Hold and wait –

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.

No preemption –

This condition can be prevented in several ways: first, if a process holding certain resources is denied a further request, that process must release its original resources and if necessary, request them again together with the additional resource.

Circular Wait –

The circular wait condition can be prevented by defining a linear ordering of resource types.

* Deadlock Avoidance –

- Do not start a process if its demands might lead to deadlock.

- Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

* Determination of a safe state *

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

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

C-A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

a] Initial state

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

Claim matrix C

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

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

b) P_2 runs to completion

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

Claim matrix C

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

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

c) P1 runs to completion

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

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

d) P_3 runs to completion

Fig – Determination of a safe state

* Determination of an unsafe state *

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

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

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	P3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

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

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

b) P1 requests one unit each of R1 and R3

Fig – Determination of an Unsafe state

* Deadlock Detection –

Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes.

* Deadlock Detection Algorithm –

1] Mark each process that has a row in the Allocation matrix of all zeros.

2] Initialize a temporary vector W to equal the available vector.

3] Find an index i such that process i is currently unmarked and the ith row of Q is less than or equal to W. That is, $Q_{ik} \leq W_k$, for $1 \leq k \leq m$; if no such row is found, terminate the algorithm.

4] If such a row is found, mark process i and add the corresponding row of the allocation matrix to W. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

For Example –

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Fig – Example of Deadlock Detection

* An Integrated Deadlock strategy –

– Group resources into a number of different resource classes.

– Use the linear ordering strategy defined previously for the prevention of circular wait to prevent deadlocks between resource classes.

– Within a resource class, use the algorithm i.e. most appropriate for that class.

As an example of this technique, consider the following classes of resources.

1) Swappable space –

Blocks of memory on secondary storage for use in swapping processes.

2) Process resources –

Assignable devices, such as tape drives and files.

3) Main memory –

Assignable to processes in pages or segments.

4) Internal resources –

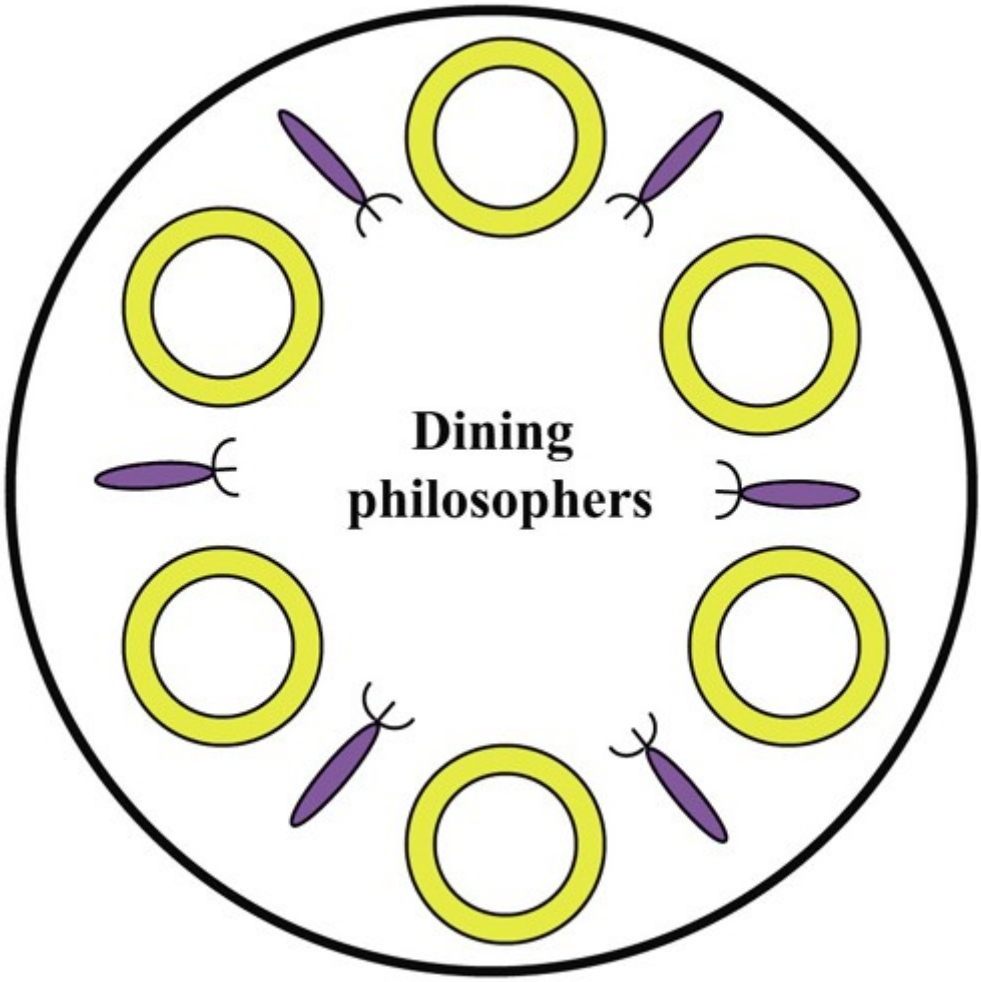
Such as I/O channels.

The order of the preceding list represents the order in which resources are assigned. The order is a reasonable one, considering the sequence of steps that a process may follow during its lifetime. Within each class, the following strategies could be used.

– Swappable space
– Process resources
– Main memory
– Internal resources

* Dining Philosophers Problem —

The eating arrangements are simple : A round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti. The problem : device a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock & starvation (in this case, the term has literal as well as algorithmic meaning.



DATE:

* solution using semaphores -

* A First solution to the Dining Philosophers Problem -

```
/* program dining philosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think ();
        wait (fork [i]);
        wait (fork [(i+1) mod 5]);
        eat ();
        signal (fork [(i+1) mod 5]);
        signal (fork [i]);
    }
}
void main ()
{
    parbegin (philosopher (0), philosopher (1),
        philosopher (2), philosopher (3),
        philosopher (4));
}
```

* Unix Concurrency Mechanism -

UNIX provides a variety of mechanisms for interprocessor communication and synchronization. Here, we look at the most important of these:

1) Pipes -

When a pipe is created, it is given a fixed size in bytes. When a process attempts to write into the pipe, the write request is immediately executed if there is sufficient room; otherwise the process is blocked. similarly, a reading process is blocked

DATE :

if it attempts to read more bytes than are currently in the pipe; otherwise the read request is immediately executed. The OS enforces mutual exclusion; that is, only one process can access a pipe at a time.

There are two types of pipes: named & unnamed. only related processes can shared unnamed pipes, while either related or unrelated processes can share named pipes.

2] Messages –

A message is a block of bytes with an accompanying type. UNIX provides msgsnd and msgrcv system calls for processes to engage in message passing. Associated with each process is a message queue, which functions like a mailbox.

3] Shared Memory –

The Fasted form of interprocess communication provided in UNIX is shared memory. This is a common block of virtual memory shared by multiple processes.

4] Semaphores –

The semaphore system calls in UNIX system V are a generalization of the semWait and semSignal primitives: several operations can be performed simultaneously and the increment & decrement operations can be values greater than 1.

5] Signals –

A signal is a software mechanism that informs a process of the occurrence of asynchronous events. A signal is similar to a hardware interrupt but does not employ priorities. That is, all signals are treated equally; signals that occure ~~that~~ at the

same time are presented to a process one at a time, with no particular ordering.