UNIT-3

SYLLABUS:

Process Synchronization: The Critical-Section Problem, Peterson's Solution, Synchronization Hardware, Semaphores, Classic Problems of Synchronization, Monitors.
Deadlocks: System Model, Deadlock Cbaracte1izatiou, Deadlock Prevention, Deadlock Avoidauce, Deadlock Detection, Recove1y from Deadlock.

Process Synchronization:

Process Synchronization:

- There are several sinllltions, where different processes need to interact with each other to achieve a common goal.
- The interaction can be done by sharing data or by sending messages.
- When the data is shared by several independent processes, then there is a chance of data becoming data inconsistent.
- ConcmTent access toshared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Example:
Let us consider two processes Pl and P2, both share a variable named 'counter'.
If both Pl and P2 execute simultaneously, with Pl incrementing the counter while P2 is decrementing it. Then, at the end the result of this variable will be such that, it is not expected by either P1 or P2 because it became inconsistent. This is oft.en called as race condition.
Hence a synchronization mechanism is needed to avoid inconsistency among several processes.

Basically, process synchronization is implemented by making a process to wait for another process perfotms an appropriate action on shared data. It is also called as signaling where one process waits for notification of an event that all occtu· in another process.

Race Condition: A race condition refers to the situation that results when many processes or tlu·eads reads and writes data items in a way that the final result generated is in accordance with the order of instructions execution in multiple processes.

- \1/hen several processes access and manipulate the san1e data concmTently and the outcome of the execution depends on the particular order in which the access takes place is called **race condition.**
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable colll1ter. Hence processes must be synchronized.

Example:
Let PI and P2 be two processes that shares a global variable 'x'. During execution, if at some point process **PI** updates some value of 'x' to 5 and at some other point updates it to 10. Thus, a race among two processes starts for changing the value of 'x' and a process that performs an update operation last determines the final value of 'x'.

Consider another situation in which two processes P3 and P4 shares two global variables 'y' and 'z' whose initial values are 5 and IO respectively. Dming some point in execution,
if process P3 executes an assigmnent statement y = y + z and
if process P4 executes an assigmnent statement z = y + z
The final values of y and z depends on the order in which the two assignment statements are executed if process P3 is executed prior to P4, then the final values of y and z are 15 and 25 respectively, on the other hand if execution of P4 precedes P3, then the result values ofy and z are 20 and 15 respectively.

Prncess Competition for Resources: Conflict arises among the various concmmntly executing processes when they are competing for the same resomce.

Consider a siniation in which two or more processes want to access a resource. Each of these concmrnntly executing processes is tmaware of tlie presence of other processes and the execution of one process does not cause any effect on the execution of the other process. Hence, the state of the resomces used, remains tmaffected.

For instance, consider that inf01mation is not exchanged between these processes and the execution of one process causes a significant effect on the behavior of the other competing processes, i.e., if two processes want to access a single resource then the OS grants the resom·ce access to only one process and let the other process to wait, as a result of which the blocked process may never get the resource and terminates in an inapproptiate manner.

Three problems dominate in case of competing process.

I) The need for nmnllll exclusion.
2) The occmrnnce of deadlock.
3) The problems of starvation.

1) The need for Mutual Exclusion: Consider a sinllltion in which nvo or more processes need access to a single non-sharable resource (Example printer). Dming the execution process, each process sends the commands to *VO* devices or sends and receives data or receives status infotmation etc. Such an *VO* device is said to be ctitical resource and a pottion of a program tliat uses is called a **ctitical section.** An important point to be considered is that there is only one program is pe1mitted to enter into the critical section at anytime.

2) The occulTence of deadlocks: The major cause for the occtmence of a deadlock is the imposition of the mutual exclusion.

Consider an example, granting of nvo resources **RI** and R2 to two processes Pl and P2. Further suppose that each of these processes want to access bot11 the resources in order to execute some function. A sitllltion may occur in which an operating system assigns the resource RI to process P2 and resource R2 to process PI. Hence, each process is waiting for one of tlie resources and will not release the acquired resource till it gets the othei· resource that is a process needs both these resources in order to proceed. This leads to deadlock.

3) **The prnblem of Starvation:** Let Pl, P2 and P3 be the three processes, each of which requires a petiodical access to resource R. If access to resource 'R' is granted to the process P1, then the other two processes P2 and P3 are delayed as they are waiting for the resource 'R'. Now let the access is granted to P3 and if Pl again needs 'R' prior to the completion of its critical section. If the OS permits Pl to use 'R' after P3 has completed its execution, then these ultimate access permissions provided to Pl and P3 causes P2 to be blocked.

This competition among the processes can be controlled by involving an OS which is responsible for allocating the resotu·ces to all processes in the system.

The Critical-Section Problem:

The resource that cannot be shared between two or more processes at the same time is called as a **critical 1·esource**.
There may be a situation where more than one process requires to access the critical resource. Then, during the execution of these processes they can send data to the critical resource, receive data from the ctitical resource or they can just get the information about the status of the critical resource by sending related commands to it. An example of a source by sending related commands to it. An

example of a critical or a non-sharable resource is "printer". A critical resource can be accessed only from the critical section of a program.

C1itical section

A critical section is a segment of code present in a process in which the process may be modifying or accessing common vaiiables or shared data items. The mo0st important thing that a system should control is that, when one process is executing its critical section, it should not allow other processes to execute in its critical sections.

Before executing critical section the process should get permission to enter its c1itical section from the system. This is called an entJ.y section. After that process executes its critical section and comes out of it this is called exit section. Then, it executes the remaining code called remainder section.

Starvation

Two ai·e more processes ai·e said to be in staivation, if they ai·e waiting perpenially for a resource which is occupied by another process. The process that has occupied the resource may or may not present in the list of processes that ai·e sta1ved.

Let Pl,P2 and P3 be the three processes, each of which requires a periodical access to resource R. if access to resource 'R' is granted to theprocessP1, then the other two processesP2 and P3 are delayed as they are waiting for the resource 'R'. Now, let the Access is granted to P3 and if PI again needs 'R' prior to the completion of its critical section. If the OS permits P1 to use "R" after P3 has completed its execution, then these alternate access permissions provided to Pl and P3 causes P2 to be blocked.

Here, we need to illustrate where sta1vation is possible or not in algorithms like FCFS, SPN, SRT and priority. Consider FCFS (First Come First Se1ved) Algorithm, in this staivation is not possible. The reason is the CPU picks the process according to anival of its burst time and run the process till its completion.

Consider SPN (Sho1test Processing Next) Algo1ithm, in this staivation is possible with the process that has long burst time. The reason is the CPU picks the process that has sh01test next burst time. Here, we can overcome sta1vation problem by using primitive SPN algorithm, which prompts the cmTently running process.

Next SRT (shortest remaining time) algoritlnn, in this sta1vation is possible with the processes that has sho1test remaining time. The reason is CPU picks the process that has shmtest remaining time. Here, we can overcome the problem of staivation by giving chance to processes that are waiting for a long period of time. Finally, consider pri01ity algorithm, in this sta1vation is possible with low priority processes. The reason is, CPU picks the process with highest priority.

We can overcome sta1vation problem by a technique called aging. This technique increases the priority of the processes that waiting for longperiod of time.

Requirements for Mutual Exclusion?

Mutual Exclusion must meet the following requirements.

1) Mutual exclusion must be enforced only one process at a time is allowed into its critical section, ainong all processes that have critical sections for the saine resource or the shared object.
2) A process that halts in its non-critical section must do so ,vithout interfering ,vith other processes.
3) It must not be possible for a process requiring access to a critical section to be delayed indefinitely, no deadlock or sta1vation.
4) When no process is in a c1itical section, any process that requests entty to its critical section must be pennitted to enter without delay.
5) No assmnptions are made about relative process speeds or nlll11ber of processors.
6) A process remains inside its critical section for a finite time only.

C1itical section problem:

- Each process has a segment of code called a **nitical section.** C1itical section is used to avoid race conditions on data items.
- In critical section, the process may be changing common vai·iables, updating a table, writing a file and so on.
- At any moment at **most one process can execute** in critical section.
- A **critical section** is a piece of code that accesses a shared resource (data stt1Jcnire or a device) that must not be concmrnntly accessed by more than one thread of execution.
- The critical-section problem is to design a protocol that the processes can use to cooperate. A Critical Section Environment contains:
  1. ***Entry Section:*** Code requesting entty into the critical section.
  2. ***Critical Section:*** Code **in**which only one process can execute at any one time.
  3. ***Exit Section:*** The end of the critical section, releasing or allmving others in.
  4. ***Remainder Sectio11:*** Rest oftl1e code after the critical section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE) ;
```

Ge11eral Structure of a Typical Process

```
do {
    while ( tum "= I );          Entry    ton
    r crItlcal section ·1        CrIt     ct on
    turn = f;
    r remainder section·1 ------ Exit  ct on
} whIle(TRUE);                   --..........
                                 Rm  nd r S ct on
```

A solution to the c1itical-section problem must satisfy the following requirements:

1) **Mutual Exclusion:** If process *P;* is executing in its critical section, then no other processes can be executing in their critical sections.

2) **Pmgress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their reminder section can paiticipate In the decision on which will enter its critical next. This selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3) **Bounded ,vaiting:** When a process requests access to a critical section, a decision that grants its access may not be delayed indefinitely. A process may not be denied access because of starvation or deadlock. A bolllld must exist on the number of times that other processes ai·e allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (i.e. All requesters must evennially be let into the critical section).

There are two general ways for handling critical sections in the operating systems. They are:

1) **Preemptive Kernel:** It allows the Kernel model process to be preempted (i.e., intenupted) during execution.
2) **Non-preemptive Kernel:** It does not allow a Kernel mode process to be preempted during execution, the process will execute lllltil it exits Kemel mode or voluntai·ily leaves control of the CPU. This approach is helpful in avoiding race conditions.

Peterson's Solution:

- **A classic software based solution to the c1itical section pmblem is known as Peterson's solution.**
- It provides a good algorithmic description of solving the critical section problem and illustrates some of the complexities involved in designing software that addresses the **requirements of mutual exclusion, pmgress and bounded waiting requirements.**
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

Peterson's solution requires two data items to be shared between the two processes:

The variable nun indicates whose nun it is to enter its critical section.

The flag anay is used to indicate if a process is ready to enter its critical section, flag[i] = trne indicates that process **P;** is ready.

```
do
    flag [i] = TRUE;
    t.urn = j;
    while (flag[j] && turn     j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

The strveture of process fl in Peterson's solution

The algorithm does satisfy the tln·ee essential c1ite1ia to solve the critical section problem. For two processes Po and P1:

1) ***Mutual exclusion:*** Po and P1 can never be in the critical section at the same time: If Po is in its critical section, then **flag [0] is true** and either **flag [1] is false** (meaning P1 has left its critical section) or nun is 0 (meaning P1 is just now ttying to enter the critical section, but graciously waiting).
In both cases, P1 cannot be in critical section when PO is in critical section.

2) ***Progress:*** Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = ttue), AND it is the other process's nun to use the critical section (nun== j).
Ifboth of those conditions ai·e 1Iue, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i.
The shai·ed vai·iable nun assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.

3) ***Bounded waiting:*** As each process enters their entty section, they set the nun vai·iable to be the other processes nun.
Since no process ever sets it back to their own nun, this ensures that each process will have to let the other process go first at most one time before it becomes their nun again.

Synchronization Hardware:

- One simple solution to the critical section problem is to simply prevent a process from being intern.1pted while in their c1itical section, which is the approach taken by non-preemptive kernels.
- Unfmtunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling intenupts on all processors.
- There is also a question as to how this approach affects timing if the clock interrupt is disabled.
- Another approach is for hardwai·e to provide ce1tain ***atomic*** operations.
- These operations ai·e guaranteed to operate as a single ins1I1.1ction, without intenuption.
- One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and renuns its previous value as shown below:

```
boolean TestAndSet(boolean *target)
    boolean rv = *target;
    *target= TRUE;
    return rv;
```

The definition of the TestAndSet () instruction.

If the machine suppo1ts the TestAndSet instruction, then we can implement mun1al exclusion by declaring a Boolean vai·iable lock, initialized to false. The st11.1cn1re of process **P**; is shown:

```
do {
    while (TestAndSetLock(&lock))
        // do nothing

    // critical section

    lock= FALSE;

    // remainder section
}while (TROE);
```

Mutual-exclusion implementation with TestAndSet ()

The **Swap** instruction, defined as sho,vn below operates on the contents of two words; like the TestAndSet instruction, it is executed atomically.

```
void Swap{boolean *a, boolean *bl
   boolean temp= *a;
   *a   *b;
   *b = temp;
```

The definition of the **Swap**() instruction.

If the machine suppo1ts the Swap instmction, then mutual exclusion can be provided as follows.
A global Boolean variable lock is declared and is initialized to false.
In addition, each process also has a local Boolean variable key.
The stmcn1re of process Pi is shown below:

```
do (
   key  = TRUB;
   while (key== TRUE)
      Swap(&lock, &key);

      // critical section

     lock  FALSE;

     // remainder section
}while (TRUE);
```

Mutual-exclusion Implementation with the Swap() instruction.

But these algorithms do not satisfy the bounded - waiting requirement. The below algorithm satisfies all the critical section problems. Common data stn1cn1res used in this algorithm are:

Boolean waiting[n];
Boolean lock;

Both these data strnctures are initialized to false.
For proving that the munJal exclusion requirement is met, we must make sure that process Pi can enter its critical section only if either waiting[i] = false or key= false.
The value of key can become false onlv if the TestAndSet() is executed.

```
do  {
   waiting[i] = TRUE;
   key= TROE;
   while (waiting[il && key)
      key= TestAndSet(&lock);
   waiting[i) = FALSE;

      // critical section

   j = (i + 1.) l n;
   while {(j !• i) && !waiting[j)l
      j = (j + 1.) \ n;

   if  (j ..,  i)
      lock= FALSE;
   else
      waiting(j) = FALSE;

      // remainder section
}while (TRUE);
```

Bounded-waiting mutual exclusion with TestAlldSet ()

Semaphores:

- The various hardware based solutions can be difficult for application programmers to in1plement.
- **Semaphores** are most often used to synchronize operations (to avoid race conditions) when multiple processes access a common, non-shareable resource.
- *Semaphores* are integer variables for which only two (atomic) operations are defined, the **wait (P)** and **signal** operations, whose definitions in pseudocode are shown in the following figure.

Wait:

```
wait(S) {
    while S <= O
       ; II no-op
    s--;
```

Signal:

```
signal(S)
    S++;
```

P(S) or S.waitO: decrement or block if already 0
V(S) or S.sigoal(): increment and wake.up process if any

- To indicate a process has gained access to the resource, the process decrements the semaphore.
- Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.
- \1/hen one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- Access to the semaphore is provided by a series of semaphore system calls.
- Semaphores can be used to deal with the II-processes critical section problem, where the *n-processes* share a semaphore **mutex** (mutual exclusion) initialized to 1. Each process Pi is organized as shown:

```
do {
   waiting(mutex);

     // critical section
   signal(mutex);

     // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores.

Impleme11tntio11

- The big problem with semaphores described above is the busy loop in the wait call **(busy waiting),** which consumes CPU cycles without doing any useful work.
- This type of lock is known as a ***spi11lock,*** because the lock just sits there and spins while it waits.
- While tl1is is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be shmt - One thread spins on one processor while another completes their critical section on another processor.
- An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU.
- In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem).
- The new defmition of a semaphore and the conesponding wait and signal operations are sho,vn as follows:

Semaphore Structure:

```
typedef suuct {
    int value;
    struct process •list;
}semaphore;
```

Wait Operation:

```
wait(semapbore *S){
  S->value--;
  if (S->value < 0){
        add this process to S->list;
        block();
```

Signal Operation:

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

- OS's distinguish between **counting and binary semaphores.**
- The value of a **counting semaphore** can range over an umestricted domain.
- The value of a **binary semaphore** can range only between O and I.
- A binary semaphore must be initialized with 1 or 0, and the completion of P and V operations must alternate.
- If the semaphore is initialized with I, then the first completed operation must be **P.**
- If the semaphore is initialized with 0, then the first completed operation must be V.
- Both P and V operations can be blocked, if they are attempted in a consecutive mam1er.
- **Binar·y semaphores** are known as **mutex locks** as they are locks that provide munial exclusion. Binary semaphores are used to deal with the critical section problem for multiple processes.
- **Counting semaphores** can be used to cortrol access to a given resource consisting of a finite mm1ber of instances. Cmmting semaphores maintain a cotmt oftJ:te number of times a resource is given.
- The semaphore is initialized to the numbetr resources available.
- Each process that wishes to use a resource rfonns a waitO operation on the semaphore.
- When a process releases a resource, it perfo s a signalO operation.

Deadlocks and Starvation:

The· ... aiting queue may result in a siniation where two or more processes are waitmg indef ...)) that can be caused only by one of the waiting processes. When such a state is reach

Po

```
i.rait (S) ;        wait(Q);
wait(Q);            wait(S);

signal($);          sign.al (Q):
signal(Q);          sign.al(S) ;j
```

Po

```
vait(S);        vait(Q);
vait(Q);        vait(S);

sigJ1al(S):     aignal(Q):
sigllal(Q);     aignal(S);
```

- Another problem related to deadlocks is indefinite blocking or struvation, a siniation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur ifwe add and remove processes from the list associated with a semaphore in LIFO order

Drawbacks of Semaphore:

1) They ru·e essentially shared global variables.
2) Access to semaphores can come from anywhere in a progran1.
3) There is no control or guarantee of proper usage.
,!) They serve two purposes, mutual exclusion and scheduling constraints.

lassie Problems of S nchronization:

he Bounded Buffer· rnblem or Producer·-Consumer·Prnblem:

Here the pool consists ofn buffers, each capable of holding one item. The mutex semaphore provides muUial exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores cotmt the number of empty and full buffers. The semaphore empty is initialized to the value n, the semaphore foll is initialized to value 0. The code below can be interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {

   // produce: an :.tem l.n next.p

   vait (elOJ)tyl,
   vait lmut..xl;

   II ad!! nextp to butler

   ai9nal (mutex) ;
   811J!>•l (full):
}"hila (TRUS);
```

The:ruc:lure ot 1IMI ""proceos.

```
do {
   wait(lull);
   wait (mut x);

   // re-:nov• an tcer.i !t'Cal butter LO nexte

   aigMl (a,tex) ;
   aignal (e1"ptyl;

   // consume the 1tel'l in next.e

)vhilo (TRUil) ;
```

TheSttuctlft o!the CO!>!l<.IT1.. _,i.

he Readers-\Vriters Prnblem:

the readers-writers problem there a.re some processes (tenned readers), who only read the shared data, and never change it, d there are other processes (tenned writers), who may change the data in addition to or instead of reading it. There is no limit o how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access. This ynchronization problem is refened to as the **rnadel'S-w1iters pl'Oblem.** There are several variations to the readers-writers roblem, most centered around relative priorities of readers versus writers.

The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to struvation of the writers, as there could always be more readers corning along to access the data.
The *second* readers-writers problem gives priority to the ,,niters. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be struved bya steady stream of writers.

The following code is an exan1ple of the first readers-writers problelll, and involves an impottant counter and two binaty semaphores:

```
do {
   wait(wrt);

   II wril1.ng is perforr.ed

   signal (wrt) :
}while (TRUE);
```

The s1ructire ot a v«iler J)IOC8$$.

```
cl <{>
   wait(mutex);
   readcount++;
   if (readcount l)
      wait(wrt);
   signal(mutex);

   // reading is performed

   wait (mutex);
   readcount--;
   , lreadcount 0)
      aignd (wrt);
   signol {mutex) ;
)while (TRUE) ;
```

Th<!structu re of a reader process.

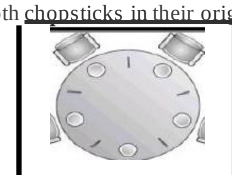
- *r_eadcount* is used by the reader processes, to count the number ofreaders cunently accessing the data.
- *mulfx* is a semaphore used only by the readers for controlled access to readcount.
- *wrt* is a semaphore used to block and release the ,:vriters. The first reader to access the data will set this lock and the last reader to ex.it will release it; The remaining readers do not touch wit.
- Note that the first reader to come along will block on wit if there is cunently a wiiter accessing the data, and that all following readers will only block on mutex for their tum to increment readcount

Some hardware implementations provide specific reader-writer locks, which ru·e accessed using an argument specifying whether access is requested for reading or writing. The use of reader-writer locks is beneficial for siniation in which: (1) processes can be easily identified as either readers or writers, and (2) there are significantly more readers than wiiters, making the additional overhead of the reader-wi·iter lock pay off **in** tenns of increased conctmency of the readers.

The Dining-Philosophers Problem:

The dining philosopher's problem is a classic synchronization problem involving the allocation of limited resources an10ng a group of processes in a deadlock-free and sta1vation-free manner: Consider five philosophers sitting around a table, in which there are five chopsticks evenly distiibuted and an endless bowl of1ice in tile center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)

- These philosophers spend their lives alternating between two activities: eating and thinking.
- When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.
- When a philosopher thinks, it puts down both chopsticks in their original locations.



The situation of the dining philosophers

One simple solution is to represent each chop stick with a semaphore. A philosopher hies to grab a chop stick by executing a wait O operation on that semaphore; she releases her chop sticks by executing tl1e signal O operation on the appropriate semaphores. Thus, the shai·ed data are *Semaphore chopstick [5};* where all elements of chopstick ru·e initialized to l. This solution is rejected as it could create a dead lock. Suppose that all five philosophers become hungi.y simultaneously and each gi.·abs her left chop stick. All the elements of chop stick will now be equal to 0. When each philosopher tJ·ies to gi.·ab her right chopstick, she will be delayed forever.

```
do {
   wait(chopstick[11);
   wait(chopstick[fi•ll % SJ);

   //  eat

   signal (chopstick[il);
   signal{chopstick  [(i+l)  % 5] l ;

   II  think

]while   (TRUE);
```

Tl"e structure of philosopher i.

Some potential solutions to the problem include:

o Only allow four philosophers to dine at the same time. (Limited simultaneous processes.

o Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)

o Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first.

A deadlock-free solution to the dining philosophers problem does not necessarily guarantee a stmvation-free one.

Deadlocks:

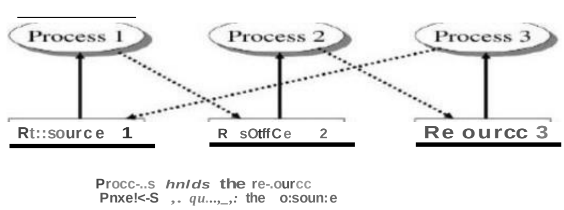
System Model:

- Finite number of resources is available in the system. These resources m·e distributed among a number of competing processes.
- Two general categories ofresources can be distinguished.
  - i. Reusable Resources
  - ii. Consumable Resource

i. **Reusable Resource:** A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use. Processes obtained resource mlits that they later release for reuse by other processes.
Exan1ple: processors, 1/0 channels, 1/0 devices, primary and seconda1y memo1y, files, database, semaphores etc.

ii **Consumable Resource:** A consumable resomce is one that can be created and destroyed. There is no limit on the number of variable resom·ces of a particular type.
Exan1ple: intenupts, signals messages and information in 1/0 buffers.

- A process requests resources before using it, and it must release the resource after using it.
- The number of resomces requested may not exceed the total nmnber of resources available in the system.
- If the system have 4 printers, then the request for printer is equal to or less than 4.
- A process may utilize the resomce if only the following sequence:
  1) **Request:** If the request is not guaranteed inunediately, then the requesting process may wait tu1til it acquire the resources.
  2) **Use:** The process can operate the resomce.
  3) **Release:** The process can release the resources.
- Process 1 is holding resource 1 and requesting resource 2; Process 2 is holding resomce 2 and requesting resource 3; Process 3 is holding resomce 3 and holding resource I.
- None of the process can proceed because all are waiting for a resource held by another blocked process.
- Unless one of the process detects the siniation and is able to withdraw the request for a resomce and release the one resource allocated to it, none of the processes will be ever be ahie to 11111.
- The below diagram shows the deadlock with three processes. Pictorially process is represented by circle and resom·ce by square.



Three deadlocked processes

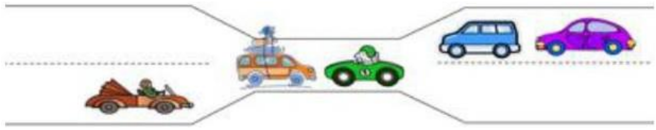
- Request is shO\vn by dotted mrnw from process to resomce. Holding resom·ce by process is shown by anow.
- Deadlock is global condition rather than local one. Deadlock condition must be handled by operating system.

.Deadlock Characterization:

Necessarv Conditions for deadlock:

A deadlock is a condition in a system where a process cannot proceed because it needs to obtain a resomce held by another process but it itself is holding a resomce that the other process needs. More fmmally, four conditions have to be met for a deadlock to occm· in a system:

1) **Mutual exclusion:** A resource can be held by at most one process.
2) **Hold and wait:** Processes that already hold resomces can wait for another resource.
3) **Non-preemption:** A resource, once granted, cannot be taken away.
4) **Circular wait:** Two or more processes are waiting for resomces held by one of the other processes.

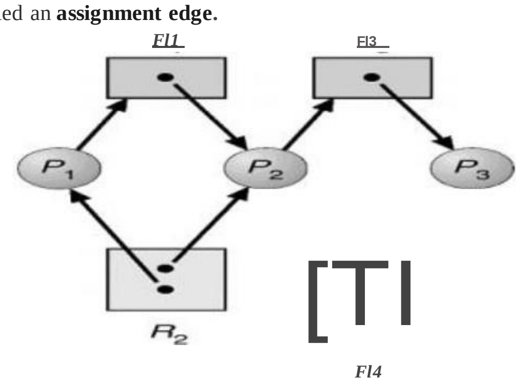


Resource-Allocation Graph:
Deadlocks can be described in terms of a directed graph called a **system rnsource allocation graph.** This graph consists of a set of veJtices Vand set of edges E. The set of vertices Vis paititioned into two different types of nodes:

- P -the setconsisting of all the active processes in the system and
- R -the setconsisting of all resotu·ce types in the system.

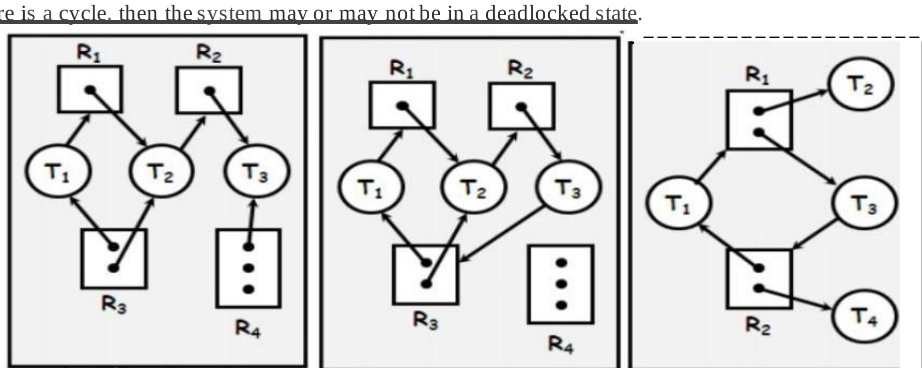
A directed edge from resource type Rj to process **Pi** is denoted by Rj -> Pi ; it signifies that an instance of resotuce type Rj has been allocated to process Pi.

- A directed edge **Pi** -> Rj is called a **1·equest edge.**
- A directed edge Rj->Pi is called an **assignment edge.**



Resource-Allocation Graph

- A process is represented using circle and resotuce type is represented using a rectangle.
- Since resotuce type may have more than one instance, each instance is represented using a dot within the rectangle.
- A request edge points to the rectangle where as an assignment edge must also designate one of the dots in the rectangle.
- When a process requests an instance of resotuce type, a request edge is inse1ted in the resotuce allocation graph.
- When this request can be folfilled, the request edge is instantaneouslytransf01med to an assignment edge.
- When the process no longer needs access to the resource, it releases the resom·ce; as a result, the assignment edge is deleted.
- If a resotuce allocation graph does not have a cycle, then the system is not in a deadlocked state.
- If there is a cycle, then the system may or may not be in a deadlocked state.



Simple Resource Allocation Graph | Allocation Graph With Deadlock | Allocation Graph With Cycle, but No Deadlock

Methods for handling deadlocks:
Deadlock problem can be dealt with in one of three ways -

I) Use a protocol to prevent or avoid deadlocks enstu"ing that the system will never enter a deadlock state
2) Allow the system to enter a deadlock state, detect it and recover.
3) Ignore the problem altogether and pretend that deadlocks never occur in the system.

- To enstue that deadlocks never occtu-, the system can use either deadlock prevention or a deadlock avoidance scheme.
- Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (listed under deadlock characterization) cannot hold.
- These methods prevent deadlocks by constraining how requests for resotuces can be made.
- Deadlock avoidance requires that the OS be given in advance additional inf01mation conceming which resotuces a process will request and use dtu"ing its lifetime. With this additional knowledge it can decide for each request whether or not the process should wait.
- To decide whether the ctuTent request can be satisfied or must be delayed, the system must consider the resotuces ctu1·ently available, the resotuces ctuTently allocated to each process and the fonu·e requests and releases of each process.
- Deadlock detection is fairly straightforwai·d, but deadlock recovery requires either abmting processes or preempting resotuces, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occtus the system will gradually slow down, as more and more processes become snick waiting for resotu·ces ctmently held by the deadlock and by other waiting processes.
- Unfo1tunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

Deadlock Prevention:

For a deadlock to occur, each of the four necessaiy conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occmTence of a deadlock.

I) Mutual Exclusion
2) Hold and Wait
3) No Preemption
4) Circular Wait

1) **Mutual Exclusion:**

- The munial exclusion condition must hold for non-shai·able resources (printer).
- Sharable resources do not require mutually exclusive access and thus caimot be involved in a deadlock (read only file).
- We cannot prevent deadlocks by denying the munial exclusion condition because some resom·ces are intrinsically non-sharable.

2) **Hold and Wait:**

- To ensure that the hold and wait condition never occurs in the system, we must guai·antee that whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process 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 cmTently allocated.

Both these protocols have two main disadvantages.

I. First, resource utilization may be low since resources may be allocated but unused for a long period.
2. Second, starvation is possible. A process that needs several populai· resources may have to wait indefinitely because at least one of the resources that it needs is always allocated to some other process.

3) **No Preemption:**

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
- One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
- Another approach is that when a resource is requested and not available, then the system looks to see what other processes ctmently have those resom·ces *and* are themselves blocked waiting for some other resom·ce.
- If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
- Either of these approaches may be applicable for resources whose states ai·e easily saved and restored, such as registers and memo1y, but are generally not applicable to other devices such as printers and tape drives.

4) **Circular Wait:**

- The fomih and final condition for deadlocks is the circular wait condition.
- One way to ensure that this 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.
- Let R ={RI, Rz, ..., Rm) be the set of resource types.
- We assign to each resource type a unique integer number, which allows us to compare two resotu·ces and to detennine whether one precedes another in our ordering.
- Fonnally, we defme a one-to-one ftmction F: RN, where N is the set ofnamral numbers.
- Each process can request resources only in an increasing order of enumeration.
- That is, a process can initially request any nmnber of instances of a resource type, Ri.
- After that, the process can request instances of resource type Ri if and only ifF(Rj) >F(Ri).
- If several instances of the same resource type are needed, a single request for all of them must be issued.
- Alternatively, whenever a process requests an instance of resource type Rj, it is required that, it has released any resources R.i such that F(Ri)2:F(Rj). If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Avoidance:

The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions. This requires more infonnation about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.) In some algorithms the scheduler only needs to know the *maximum* munber of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order. When a scheduler sees that staiting a process or granting resource requests may lead to furure deadlocks, then that process is just not sta1ied or the request is not granted. A resource allocation ***state*** is defined by the number of available and allocated resources, and the maximtmI requirements of all processes in tl1e system.

Safe state: A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximmns) without entering a deadlock state. More fonnally, a state is safe if there exists a ***safe sequence*** of processes {Po, P1, P2, ..., PN} such that all of the resomce requests for P; can be granted using the resources cunently allocated to P; and all processes P1 where j < i. (I.e. if all tl1e processes p1ior to Pi finish and free up tl1eir resources, then Pi will be able to finish also, using the resources that they have freed up.) If a safe sequence does not exist, then the system is in an m1safe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

As long as the state is safe, the OS can avoid m1safe (and deadlocked states). In an unsafe state, the OS cannot prevent processes from requesting resources such that a deadlock occms. The behavior of the processes controls unsafe states.

111e idea is simply to ensme that the system will always remain in a safe state. Initially, tl1e system is in a safe state. Whenever a process requests a resource tl1at is cunently available, the system must decide whether the resomce can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

There exist a total of 12 resources. Each resource is used exclusively by a process. The ctment state looks like this.

Po, P1, and P2 ai·e the processes. Process Po requires 10 tape drives, process P1may need as many as 4, and process P2 may need up to 9 tape drives. Suppose that, at tinle to, process Po is holding *5* tape drives, process P1is holding 2, and process P2 is holding 2 tape drives. At time to, the system is in a safe state.

Process	Max Needs	Allocated	Current Needs
PO	10	5	5
Pl	4	2	2
P2	9	3	7

The sequence < P1, Po, P2> satisfies the safety condition, since process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process Po can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P2could get all its tape drives and retum them (the system will then have all 12 tape drives available).

Suppose that, at time 11, process P2 requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape chives. When it renuns them, the system will have only 4 available tape d1ives. Since process Po is allocated 5 tape ch·ives, but has a maximum of 10, it may then request 5 more tape ch·ives. Since they are tmavailable, process Po must wait. Similarly, process P2 may request an additional 6 tape ch·ives and have to wait, resulting in a deadlock.

process	holding	max claims	outstanding requests
A	4	6	2
B	3	9	6
C	4	11	7

unallocated: 2

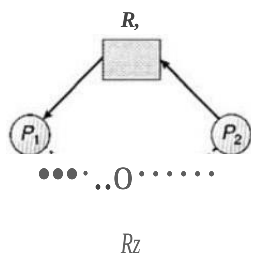
deadlock-free sequence: A,8,C

However, ifB should have 7 instead of 6 outstanding requests, this sequence is not safe: deadlock exists.

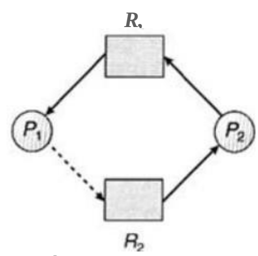
Resource Allocation Graph:

If there is a resot11·ce allocation system with only one instance of each resomce type, a vai.iant of the resomce allocation graph can be used for deadlock avoidance. Inaddition to the request and assignment edges, a **claim edge** is also used. This edge resembles a request edge in direction but is represented in the graph by a dashed gne. When a process makes a request, the request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph. An algorithm for detecting a cycle in this graph requires an order of $\mathbf{n}^2$ operations where **n** is the munber of processes in the system.

Consider the resomce-allocation graph shown below. Suppose that P2 requests R2. Although R2 is cunently free, we cannot allocate it to P2, since this action will create a cycle in the graph as shown below. A cycle indicates that the system is in an tmsafe state. If P1 requests R2, and P2requests R1, then a deadlock will occur.



Resource-allocationgraph for deadlock avoidance.



An unsafe state in a resource-allocation graph

Banker's algorithm:

Q) Explain in detail about Banker's algo1ithnL

For resomce categories that contain more than one instance the resomce-allocation graph method does not work. A less efficient scheme called the **Banke1·'s Algo1ithm,** which gets its name because it is a metliod that bankers could use to assme that when they lend out resot11·ces they will still be able to satisfy all their clients. (A banker won't loan out a little money to stait building a house tmless they are assmed that they will later be able to loan out the rest of the money to fmish the house.) When a process stai.ts up, it must state in advance the maximum allocation ofresomces it may request, up to the ai.notmt available on the system. When a request is made, the scheduler detern1ines whether granting the request would leave the system in a safe state. If not, then the process must wait tmtil the request can be granted safely.

The banker's algorithm relies on several key data st:ructmes: (where n is the nt11nber of processes and m is the number of resomce categories.)

- ***Available:*** vector of length m indicates the nlllllber of available resources of each type. If AvailableLi] = k, there are k instances ofresomce type Rj available
- ***Max:*** an nxm matrix defines the maximum demand of each process. If Max[i,j] = **k,** then process Pi may request at most k instances ofresomce type Ri.
- ***Allocatio11:*** an nxm matrix defines the m1111ber ofresomces of each type ct11rnntly allocated to each process. If Allocation[i,j] =k, then process Pi is cmTently allocated k instances ofresource type Rj.
- *Need:* an nxm matrix indicates the remaining resource need of each process. If Need[ij] = k, then process Pi may need k more instances ofresomce type **Ri** to complete its task. Note that Need[ij] = Max[ij] - Allocafion[ij],
- ***Request:*** It is a vector size m which indicates that the process Pi has requested for some resomce.

Each row in the matrices Allocation and Need ai.·e treated as vectors and refer to them as Allocationi and Needi, respectively. The vector Allocation; specifies the resources cunently allocated to process Pi; the vector Need, specifies the additional resources that process Pi may still request to complete its task.

Safety Algo1ithm: The Algorithm for finding out whether or not a system is in a safe state.

1) Let *Work* and *Finish* be vectors of length m and n respectively.

   Work is a working copy of the available resources, which will be modified during the analysis. Finish is a vector of Booleans indicating whether a particular process can finish. (Or has finished so far in the analysis.)Initialize Work to Available, and Finish to false for all elements.

   Work= Available

   Fi11ish Ii) = ***false* for *i* = 0, 1, ..., *11*- 1**

2) Find a process i such that,

   (a) ***Finish* ltl = *false***

   (b) ***Need;*** $. ***Work***

   Ifno such i exists, then go to step 4.

3) ***Work = Work+Allocatio11;***
   Fi11ish[i] = true
   Goto step 2.

4) If ***Fillish* ltl =** **true** for all i, then the system is in a safe state, because a safe sequence has been found.

This algorithm may require an order of m x n^2operations to decide whether a state is safe.

Resource - Request Algorithm: - alg01ithm which detennines if requests can be safely granted.

Let Requesti be the request vector for process Pi. If Request;[j] = k, then process Pi wants k instances of resoU1ce type Rj. When a request for resom·ces is made by process Pi, the following actions are taken:

1) If ***Request;***$. ***Need;*** go to step **2.** Othe1wise, raise enor condition, since process has exceeded its maximum claim.
2) If ***Request;***$. ***Available,*** go to step 3. Otherwise *P,*- must wait, since resources are not available.
3) Pretend to allocate requested resomces to *P,*- by modifying the state as follows:

   Available =Available -Request;
   Allocatio11; = Allocatio11; + Request;;
   Need; = Need;- Request;;

   If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. But, if the new state is unsafe, then Pi must wait for Requesti and the old resom·ce-allocation state is restored.

Example of Banker's Algorithm:

Five processes- Po, P1, P2, P3, P4 with three resource types: A, B, C with 10, 5, 7 instances. The content of the matrix Need is defined to be Max-Allocation (Max= Allocation+ Need).
A snapshot of the system taken at time To is shown below

	Allocation	Mnx	Avnilable	Need
	ABC	ABC	ABC	ABC
Po	010	753	332	743
I'1	200	322		122
I'2	302	902		600
P3	211	222		011
1'4	002	433		431

The system is in a safe state since the sequence< ***Pt, P3, P4, P2,* Po**>satisfies safety criteria

Now, if process Pl requests 1 instance of A and 2 instances ofC. (Request[!]= (1, 0, 2))
To decide whether this request can be immediately granted, we first check that
Request :S Available (that is, (1,0,2) :S (3,3,2) tme.
Now we anive at the following new state:

	Allocntio11	Need	Availnble
	ABC	ABC	I\.B C
Po	010	743	230
P1	302	020	
P2	302	600	
P3	21 l	011	
P,	002	-!31	

Executing safety algorithm shows that sequence < ***P1, P3, P4,* Po, *Pi***> satisfies safety requirement. When the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resoU1ces are not available. A request for (0,2,0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

Disadvantages of the Banker's Algorithm:

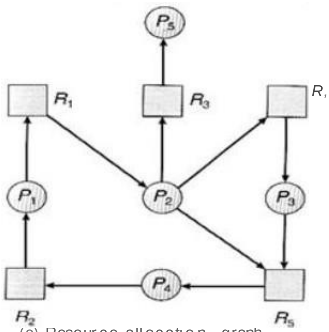
- It requires the number of processes to be fixed; no additional processes can stait while it is executing.
- It requires that the munber of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occuning.
- It allows all requests to be granted in finite time, but one year is a finite amount of time. Siniilai·ly, all of the processes gi.1ai·antee that the resoU1ces loaned to them will be repaid in a finite amolll!t of time. While this prevents absolute starvation, some pretty hungry processes might develop.
- All processes must know and state their maxim= resoU1ce need in advance.

Deadlock Detection:

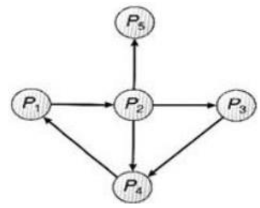
If deadlocks are not avoided, then another approach is to detect when they have occmTed and recover somehow. h1 addition to the perfo1mance hit of constantly checking for deadlocks, a policy/ algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be ab01ted or have their resources preempted.

Single instance of each resource type

If all resources have only a single instance, then we can define a deadlock detection algoritlun that uses a variant of the resource allocation graph called a wait for graph. A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.



(a) Resource-allocation graph.



(b) 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 the graph requires an order of n^2 operations where n is the number of ve1tices in the graph.

Several instances of a resource type:

The wait for graph scheme is not applicable to resource allocation system with multiple instances of each resource type. For several instances ofresource type, the algorithm employs several time varying data strucn1res. They are:

- Available - a vector oflength m indicates the number of available resources of each type
- Allocation - an nxm mahix defines the number of resources of each type cmTently allocated to each process
- Request - annxm mati·ix indicates the current request of each process.

The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:

Step l) Let *Work* and *Finish* be vectors oflength *m* and *n* respectively.

Initialize ***Work=Available.***

For i = *0, 1, ..., n-1,* if *Allocation -f- 0,* then *Finish{i] = false;*

otl1e1wise *Finish{i] = true..*

Step 2) Find a process *i* such that,

Finish*[zl = *false

Need;* :S *Work

Ifno such i exists, then go tostep 4.

Step 3) ***Work= Work+ Allocation;***

Finish[i] = true

Go to step 2.

Step 4) The basic Banker's Algorithm says that if *Finish[/} == true* for all i, that there is no deadlock.

This alg01ithm is more specific, by stating that if *Finish[ij == false* for any process P;, 0 i n then that process is specifically involved in the deadlock which has been detected.

This algonthm reqmres an order of **mxn**^{2} operations to detect whether the system 1s ma deadlocked state.

Five processes PO through P4 and three resource types A (7 instances), B (2 instances), and C (6 instances). Snapshot at time TO

	Allocation	Request	Total
	ABC	***ABC***	***ABC***
Po	0 1 0	000	726
P1	200	202	*Allocated*
P2	303	000	726
P3	2 1 1	100	*Available*
P4	002	002	**000**

Sequence O, 2, 3, 1, 4 will result in *Finish[t]* for all *i*.

Detection algorithm usage:

Two factors decide when to invoke the detection algorithm.

- ► How often is a deadlock likely to occur?
- ► How many processes will be affected by deadlock when it happens?

There are two obvious approaches, each with trade-offs:

1) Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. The down side of this approach is the extensive overhead and perf01mance hit caused by checking for deadlocks so frequently.
2) Do deadlock detection only when there is some clue that a deadlock may have occuued, such as when CPU utilization reduces to 40% or some other magic munber. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and dan1aging to more processes.

Recovery from Deadlock:

There are three basic approaches to recove1y from deadlock:

1) Inform the system operator, and allow him/her to take manual inte1vention.
2) Terminate one or more processes involved in the deadlock
3) Preempt resources

Process Ter111i11atio11:

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

Abort all deadlocked processes - Breaks the deadlock cycle, the deadlocked processes may have computed for a long time and the results of partial computations must be discarded and will have to be recomputed later.

Abort one process at a time until the deadlock cycle is eliminated - Incurs considerable overhead, since after each process is abo1ted, a deadlock detection algorithm must be invoked to detennine whether any processes are still deadlocked.

If the partial termination method is used, then we must determine which deadlocked process should be tenninated. Abmi those processes whose tennination will incur minimum costs.

Many factors may affect which process is chosen including:

1) What the priority of the process is?
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 type 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 tenninated?
6) Whether the process is interactive or batch?

Resource Preemption:

To eliminate deadlocks using resource preemption, preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, three issues need to be addressed:

- **Selecting a Yic.tim**- Which resources and which processes are to be preempted?
- **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfo1tunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning.
- **Starvation** - How do you guarantee that a process won't sta1ve because its resources are constantly being preempted? One option would be to us.

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. So this process never completes its designated task, a staivation sinIBtion that must be dealt with in any practical system. A process can be picked as a victim only a finite number of times.