Unit. 3 (Ch-7)
Process Synchronization

Cooperating processes can affect or be affected by each other. Cooperating processes may directly share a logical address space to share data and code through light weight processes or threads. They may share the data through files also.

Several processes need to communicate with one another simultaneously. Concurrent access to share data may result in data inconsistency. When 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 avoid the race condition, only one process should be allowed to manipulate the shared variable and data. This requires proper synchronization while using the shared data.

22

Critical Section problem:

Critical section is a segment of code, in which a process may be changing variables, updating a ~~variable~~ table, writing a file and so on. When one process is executing in its critical section, no other cooperating process is to be allowed to execute in its critical section. Thus, the execution of critical section by the cooperating processes is mutually exclusive. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section is followed by an exit section. The remaining code is the remainder section.

Nilam ...

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

Structure of a process ~~Pi having Critical Section~~

Requirements for a solution to Critical Section Problem:

Solutions should satisfy three main requirements:

If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

1) Mutual Exclusion: Mutual exclusion must be enforced by allowing one process at a time in its critical section, among all processes that have critical sections for the same resource or shared object.

2) Progress: When no process is executing in its ~~shared~~ critical section, any process that requests entry to its critical section must be permitted to enter without delay.
No process should be delayed indefinitely i.e. no starvation and deadlock is allowed.

3) Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section & before that request is granted. ~~after a process has made~~

Other requirements are:

- No assumptions are made about relative speeds or number of processors.
- Basic machine-language instructions such as load, store, test etc. are executed atomically. Thus if two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order.

Two process Solutions : Software approaches can be implemented for concurrent processes that execute on a single processor or a multiprocessor machine ~~which~~ with shared main memory.

Algorithm 1: This algorithm is for two processes. Two cooperating processes (P_0 & P_1) share a common variable (turn) initialized to 0 (or 1).

A process (P_0 or P_1) wishing to execute its critical section first ~~examines the~~ content of turn. If the value of turn is equal to the number of process then process may execute in its critical section Otherwise it is forced to wait.

Waiting process repeatedly reads the value of turn until it is allowed to enter its critical section. This procedure is known as busy waiting.

do {	do {
$\boxed{\text{while (turn != 0); /* Do nothing, keep testing */}}$	$\boxed{\text{while (turn != 1); /* do nothing, keep testing */}}$
Critical Section	Critical section
$\boxed{\text{turn = 1;}}$	$\boxed{\text{turn = 0;}}$
remainder Section	remainder Section
} while (1);	} while (1);
Structure of process P_0	Structure of process P_1

Drawbacks :

1) Processes must strictly alternate in their use of their critical section; thus the pace of execution is governed by the slower process.

3) If one process fails before exit section, other process is permanently blocked.

Thus, this algorithm follows mutual exclusion but does not follow progress requirement of solution.

Algorithm 2: Algorithm 1 does not give information about the state of each process, it remembers only which process is allowed to enter its critical section.

Variable turn of algorithm 1 is replaced by a boolean array flag [2]. The elements of array are initialized to false. Flag [0] corresponds to P_0 and flag [1] corresponds to P_1.

In this algorithm, P_0 sets flag [0] to be true signaling that it is ready to enter its critical section. Then P_0 checks to verify that process P_1 is not also ready to enter its critical section. If P_1 were ready, then P_0 would ~~enter the critical section~~ wait until P_1 indicated about no need of critical section. (until flag [1] was false)

At this point, P_0 would enter the critical section and set flag [0] to be false in exit section allowing the process P_1 to enter its critical section.

```
do{
    flag[0] = true;
    while (flag[1]);
        /* do nothing, keep testing */;

        critical section

    flag[0] = false;

        remainder section
} while (1);
```

Structure of process P_0

```
do{
    flag[1] = true;
    while (flag[0]);
        /* do nothing, keep testing */;

        critical section

    flag[1] = false;

        remainder section
} while (1);
```

Structure of process P_1

Drawbacks: In this solution, mutual exclusion is satisfied because before entering in its critical section, each process checks other's flag to be false.

But progress requirement is not satisfied.

If P_0 sets flag [0] = true
P_1 sets flag [1] = true
P_0 checks flag [1] to be false
P_1 checks flag [0] to be false

Now P_0 and P_1 are looping forever in their respective while statements.

3) Algorithm 3:

By combining the key ideas of algorithm 1 and algorithm 2, a correct solution of critical-section problem is obtained, where all three requirements are met. The processes share two variables: boolean flag [2];
int turn;

Initially flag [0] = flag [1] = false
turn = 0 (or 1)

- First, P_0 sets flag [0] to be true.
- Then, P_0 sets ~~flag~~ turn to be 1, to ensure that if other process wishes to enter the critical section it can do so.
- When process P_0 finishes its critical section, P_1 sets flag [0] to be false.

This algorithm is Peterson's algorithm.

```
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1)
    /* do nothing; keep testing */;

    critical section

    flag[0] = false;

    remainder section
} while(1);
```

Structure of process P_0

```
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0)
    /* donothing, keep testing */;

    Critical section

    flag[1] = false;

    Remainder section
} while (1);
```

Structure of process P_1

Correctness of Solution

1. Mutual exclusion : A process P_0 can enter its critical section only if either flag[1] = false or turn = 0. If P_0 & P_1 processes set the flag[0] & flag[1] to be true. But turn is a shared variable, it can be either 0 or 1. Thus, at a time only one process can ~~eter~~ enter its critical section.
2. Progress requirement : A process P_0 can stuck in while loop only when flag[1] = true and turn = 1, otherwise ~~it~~ P_0 enters its critical section. If flag[1] = true and turn = 1, P_1 enters its critical section.
3. Bounded Waiting : Once P_1 exits its critical section, it sets flag[1] to be false. If P_1 resets the flag[1] to be true, turn also is set as 0. Thus, P_0 gets chance to enter its critical section.

48

Multiple-Process Solution:

The algorithm for solving the critical section problem for n processes is known as the bakery algorithm.

The common data structures are:

```
boolean choosing [n];
    int number [n]
```

Initially, these data structures are initialized to false and 0 respectively.

```
do {
    choosing [i] = true;
    number [i] = max (number [0], number [1], ... number[n-1]) + 1;
    choosing [i] = false;
    for (j=0; j<n; j++) {
        while (choosing [j]) /* do nothing */;
        while ((number[j] != 0) && (number [j], j] < number [i], i]))
                      /* do nothing */;

      critical section

    number [i] = 0;

      remainder section

} while (1);
```

A process requesting entry to critical section is given a numbered token such that the number of the token is larger than the maximum number issued earlier. The algorithm permits the processes to enter the critical section in the order of their token numbers.

If two or more processes choose their tokens concurrently & After choosing a token, a pair of Pi (number[i], i) is compared with similar pairs for other processes using the precedes relation as:

(number[j], j) < (number [i], i) if

number[j] < number [i], or

number [j] == number [i] and j < i

Thus, if more than one process has obtained the same token number, the request of the process with the smaller process id is considered to be earlier. A process may enter the critical section if its pair is the 'smallest' in the system, else it gets into a busy wait repeatedly checking for this condition. Thus, processes enter the critical section in the order in which they raise their requests.

Synchronization hardware

Some hardware instructions are available on many systems which can be used effectively used in solving the critical section problem. Different methods are:

(1) Interrupt disabling

(2) Special Machine Instructions

(a) Test and set instructions

(b) Exchange Instructions.

1) Interrupt disabling.

In a uniprocessor environment interrupts must be avoided to occur while a shared variable is being modified.

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

Because the critical section can not be interrupted, mutual exclusion is guanteed. Drawbacks ① The efficiency of execution is degraded because the processor is limited in its ability to interleave program.
② Disabling interrupts on a multiprocessor can be time-consuming as message should be passed whenever a process enters its critical section

2. special Machine Instructions

In a multiprocessor environment, several processors share access to a common main memory. At a hardware level, a access to a memory location excludes any other access to that same location. Many machines provide special machine instructions that carry out two actions atomically, i.e. as an uninterruptible ~~unit~~ unit. Because these actions are performed in a single instruction cycle, they are not subject to interference from other instructions.

a) Test and Set Instructions : The instruction tests the value of its argument i. If the value is 0, then it replaces it by 1 and returns true. Otherwise the value is not changed and false is returned. The entire testset function is carried out atomically, i.e. it is not subject to interrupts.

```
boolean testset (int i)
{ if (i==0)
  {
    i=1;
    return true;
  }
  else
  { return false;
  }
}
```

```
do
{
  while (!Testset (bolt))
      /* do nothing */;

  critical section

  bolt = 0;

  remainder section
} while (1);
```

Test and Set instruction

A shared variable bolt is initialized to 0. The only process that may enter its critical section is one that finds bolt equal to 0. All other processes attempting to enter their critical section go into a busy-waiting mode. When a process leaves its critical section, it resets bolt to 0; at this point one and only one of the waiting processes is granted access to its critical section.) The choice of process depends on which process executes the testset instruction next.

b) Exchange Instruction : (or Swap Instruction)

The exchange instruction which is performed atomically, can be defined as follows:

```
void exchange (int keyi, int bolt)
{ int temp;
  temp = bolt;
  bolt = keyi;
  keyi = temp;
}
```

A shared variable bolt is initialized to 0. Each process uses a local-variable keyi that is initialized to 1. The only process that may enter its critical section is one that finds bolt equal to 0. It excludes all other processes from the critical section by setting bolt to 1. When a process leaves its critical section, it resets bolt to 0, allowing another process to gain access to its critical section. If bolt = 1 then exactly one process ~~is~~ is in its critical section whose key value equals to 0.

```
do
{
    keyi = 1;
    while (keyi = 0)
      exchange (keyi, bolt);

        critical section

    bolt = 0;

        remainder section
} while (1);
```

Exchange Instruction

Bounded waiting mutual exclusion with Test & Set

Test Set and Exchange Instruction methods do not satisfy bounded waiting requirement. This algorithm satisfies all the critical section requirements. The common data structures are:

boolean waiting[n]
boolean lock;

These data structures are initialized to false.

Process P_i can enter its critical section only if either waiting[i] == false or key == false. key is a local variable which was initialized to true. The value of key can become false only if the Test and set is executed and lock is false, which means no other process is executing its critical section. The waiting [i] can become false only if process comes out from while loop after having the key value false due to lock value as false. This proves mutual exclusion and progress requirements.

- 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.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestSet(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 (1);
```

Semaphores :

For complex problem, a synchronization tool named semaphore is used. A semaphore S is an integer variable is accessed only through two standard atomic operations: wait (or P) and signal (or V). The definitions of wait and signal are:

```
wait (S)
{ while ( S <= 0)
      ; // no-operation
    S-- ;
}
```

wait definition

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

Signal definition

Modifications to the value of semaphore in the wait and signal operations must be executed indivisibly.

```
do {
    wait(mutex);

      Critical section

    Signal(mutex);

      Remainder section
} while (1);
```

Mutual exclusion with semaphore

Disadvantage :

Semaphore solution requires 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 entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is called a spinlock. Spinlocks are useful in multiprocessor system. The advantage of a spinlock is, that no context switch is required when a process is waiting. Spinlocks are useful when processes have to wait for a short period.

Solution for busy waiting:

When a process executes the wait operation and finds that the semaphore value is not positive. Instead of 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.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation by a wakeup operation. Wakeup operation changes the process from waiting state to the ready state and places it in the ready queue]

Semaphore is defined as a "C" struct here

```
typedef struct
  { int value;
    struct process *L;
  } semaphore;
```

Each semaphore has an integer value and a 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. These two operations are provided by the OS as system calls

```
void wait (semaphore S)
{ S.value --;
  if (S.value <0)
  { add this process to S.L;
    block();
  }
}

void signal (semaphore S)
{   S.value ++;
    if (S.value <=0)
    { remove a process P from S.L;
      wakeup (P);
    }
}
```

In classical definition of semaphore with busy waiting, the semaphore value is never negative. But, this implementation (waiting queue) may have negative semaphore value. This is due to the decrement result of wait operation. Semaphore list is implemented as FIFO queue which ensures the bounded waiting. Wait and signal operations on the same semaphore should not be executed by two processes at ~~same~~ same time. In a uniprocessor environment, interrupts can be inhibited at the time of wait and signal operations. In a multiprocessor environment, inhibiting interrupts does not work, we can employ any of the correct software solutions (Algorithm 1, 2 or 3) for critical section problem.

But complete elimination of busy waiting is not possible. Here, busy waiting is limited to only wait/signal sections to be executed.

Drawbacks of waiting queue Implementation

1) Deadlock: A situation may occur when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

Example: A system has two process P_0 and P_1 each accessing two semaphores, S and Q, set to the value 1:

P_0	P_1
wait (S);	Wait (Q);
Wait (Q);	wait (S);
.	.
.	.
.	.
Signal (S);	signal (Q);
Signal (Q);	signal (S);

If P_0 executes wait (S), and then P_1 executes wait (Q) ① When P_0 executes wait(Q), it is blocked in semaphore queue Q. Then, P_0 has to wait until P_1 executes signal (Q). ② when P_1 executes wait(S) it is blocked in semaphore queue S. Then, P_1 has to wait until P_0 executes signal (S).

Since these signal operations can not be executed, P_0 and P_1 are deadlocked. Various mechanism can be performed to handle deadlock.

2) Starvation: A situation where processes may wait indefinitely within the semaphore, may occur due to LIFO implementation of queue

Binary Semaphores :

General semaphore (counting semaphore) can have the integer value over an unrestricted domain. A binary semaphore is a semaphore with an integer value of 0 or 1.

Implementation of counting semaphore using binary semaphore.

```
Void Wait (semaphore s)
{
  waitB(mutex);
  s--;
  if (s<0) {
      SignalB(mutex);
      waitB(delay);
      }
      SignalB(mutex);
}
```

```
Void Signal (semaphore s)
{
  waitB (mutex);
  s++;
  if (s<=0)
      SignalB (delay);
  SignalB (mutex);
}
```

Initially s is set to the desired semaphore value. Each Wait operation decrements s and each signal operation increments s.

The binary semaphore mutex which is initialized to 1, assures that there is mutual exclusion for the updating of s. The binary semaphore delay, which is initialized to 0, is used to suspend the processes.

5A

Classic problems of synchronization (by semaphore)

1. Producer - Consumer problem : or Bounded Buffer Problem.

Problem : " A producer process produces information that is consumed by a consumer process. e.g a print program produces characters that are consumed by the printer driver".

To allow producer and consumer processes to run concurrently, there should be a buffer of items that can be filled by the producer and emptied by the consumer. The producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced. The consumer must wait until an item is produced.

The unbounded-buffer producer - consumer problem places no limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

The bounded - buffer producer-consumer problem assumes a fixed buffer size. In this case the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Solution for bounded-buffer producer-consumer problem :

Buffer pool consists of n buffers, each capable of holding one item. There are three semaphore variables in solution.

1) The semaphore mutex provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.

2) The semaphore empty count the number of empty buffer and initialized to the value n.

3, The semaphore ~~emp~~ full count the number of full buffer and initialized to the value 0.

```
semaphore mutex = 1 ;
semaphore full = 0 ;
semaphore empty = n ; /*size of buffer*/
do {
    ......
    produce an item in nextp
    ......
    wait (empty) ;
    wait (mutex) ;
    ......
    add nextp to buffer
    ......
    signal (mutex) ;
    signal (full) ;
} while (1) ;
```

The structure of the producer process

```
do {
    wait (full) ;
    wait (mutex) ;
    ......
    remove an item from buffer to nextc
    ......
    signal (mutex) ;
    signal (empty) ;
    consume the item in nextc
} while (1) ;
```

The structure of consumer process

Wait and signal operations should be atomic (i.e. these operations should run as a ~~sig~~ single instruction). Only one process at a time may manipulate a semaphore with either a wait or signal operation. Thus, any of software scheme such as Peterson's algorithm can be used.

One of hardware - supported scheme for mutual exclusion also can be used.

2. The Readers - Writers Problem :

Problem :

Several concurrent processes may share a file or database. Some processes may read the content and some may update (read & write) the content. The process that only read the content is known as reader and process that read & write is known as writer.

If the two readers access the shared data no adverse effect occurs. If a writer and some other process (whether a reader or a writer) access the shared data, inconsistency may occur.

Thus, writers should have exclusive access to the shared object. This synchronization problem is referred to as the readers - writers problem.

The readers - writers problem has several variations due to priorities.

reader priority — First readers - writers problem - No reader will be kept waiting unless a writer has already obtained permission to use the shared object.

writer priority — Second readers - writer problem - If a writer is waiting to access the object, no new readers may start reading.

Solution : A solution in both the cases may result in starvation. In first case, writer may starve, in second case, reader may starve.

In the solution to the first readers-writers prob the reader processes share the following data structures:

semaphore, mutex, wrt; initialized to 1

int readcount; initialized to 0.

1) The semaphore mutex used to ensure the mutual exclusion when the variable readcount is updated

2) The semaphore wrt is common to both reader and writer. It is used to ensure the mutual exclusion for writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

3) The ~~semaphore~~ variable readcount keeps track of how many processes are currently reading the object.

```
wait (mutex);
readcount++;
if (readcount == 1)
    wait (wrt);
signal (mutex);
....
 reading is performed
....
wait (mutex);
readcount--;
if (readcount == 0)
    signal (wrt);
signal (mutex);
```

The structure of a reader process

```
wait(wrt);
.....
 writing
signal(wrt);
```

writer process

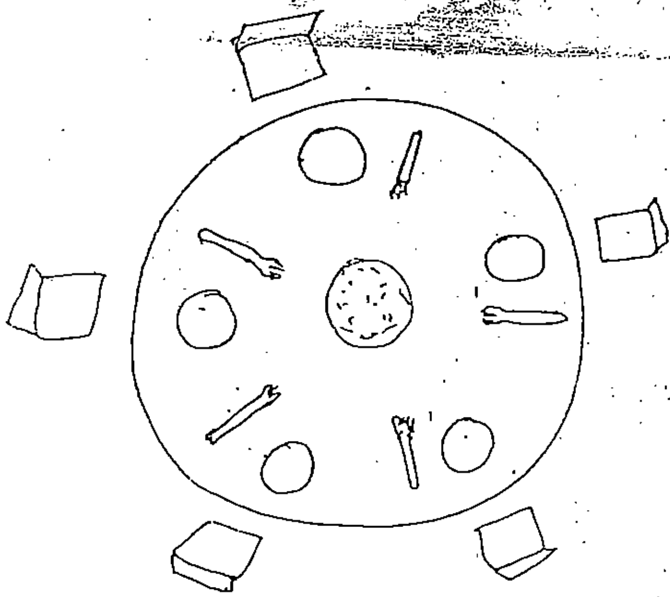
If a writer is in the CS and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex.

When a writer executes signal (wrt), either execution of a writer or a reader (waiting) is resumed by scheduler.

3. The Dining - Philosophers Problem : 56

Problem :

There are five philosophers who spend their lives thinking and eating. The philosophers share a common 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 with five single chopsticks and five plates.



The Dining arrangement of five philosophers

When a philosopher thinks, she does not eat. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are between her and her left and right neighbors. A philosopher may pick up only one chopstick at a time. Philosopher cannot pick up a chopstick 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 her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The dining-philosophers problem is a classic synchronization because it is an example of a large class of concurrency-control problems. It is a representation of the need to allocate several resources among several processes without the occurrence of deadlock and starvation.

using semaphore

Solution: Each chopstick can be represented by a semaphore. A philosopher can pick up the chopstick by executing a wait operation on that semaphore, she releases her chopstick by executing the signal operation on the appropriate semaphore. The shared data structure is

semaphore chopstick [5];

element of array belongs to one chopstick and initialized to 1.

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

This solution guarantees about mutual exclusion i.e. two neighbours can not eat simultaneously.

Drawbacks:

1) If all five philosophers become hungry simultaneous and each picks up her left chopstick.

All the elements of array chopstick will become 0. No philosopher can pick up right chopstick, and deadlock occurs.

Solutions to deadlock:

- Allow at most four philosopher to be sitting simultaneously at the table.
- Allow a philosopher to pick her chopsticks only if both chopsticks are available i.e. chopsticks should be picked up in critical section.
- Asymmetric Solution: An odd philosopher picks up first her left chopstick and then right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Problems with Semaphores: (mutual exclusive Problem)

All the processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait (mutex) before entering the critical section, and the signal (mutex afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

1. If a process interchanges the order in which the wait and signal operations on the semaphore mutex are executed, resulting in the following section

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

In this situation, many processes may be exe... in their C.S simultaneously, violating the mutual exclusion requirement.

2, If a process replaces signal (mutex) with wa... (mutex) :

wait (mutex);

...

Critical section

Wait (mutex);

In this case, a deadlock occurs.

3, A process omits the wait (mutex) or the signal (mutex), or both. Either deadlock occurs or mutual exclusion is violated.

<u>Critical region.</u>

The critical region (CR) is a control structure for implementing mutual exclusion over a shared variable.

The critical-region (high-level language synchronization construct) requires that a variable v of type T, which is to be shared among many processes be decalared as,

var V: shared T

The variable v can be accessed only inside a region statement. This constructs means when S statement is being executed, no other process can access the variable v.

region v do S

Sometimes condition is also associated with region construct. Then, it is known as conditional critical region.

Region v when B do S

The expression B is a boolean expression if this B is true, then only S statement can be executed.

When a process tries to enter the critical section, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process undergoes mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v.

The critical-region construct can solve the simple errors associated with the semaphore solution to the critical-section problem that may be made by a programmer.

Bounded Buffer problem Solution through Critical region:

```
struct buffer {
        int pool [n];
        int count, in, out ;
    };
```

```
region buffer when (count < n) {
        pool [in] = nextp ;
        in = (in+1) % n ;                code for
        count ++ ;                       producer
    }
```

```
region buffer when (count > 0) {
        next c = pool [out] ;
        out = (out +1) % n ;
        count -- ;                       code for
    }                                    consumer
```

The producer process inserts a new item nextp into the shared buffer by executing the segment code for producer. The consumer process removes an item from the shared buffer and puts in the nextc by executing the segment code for consumer.

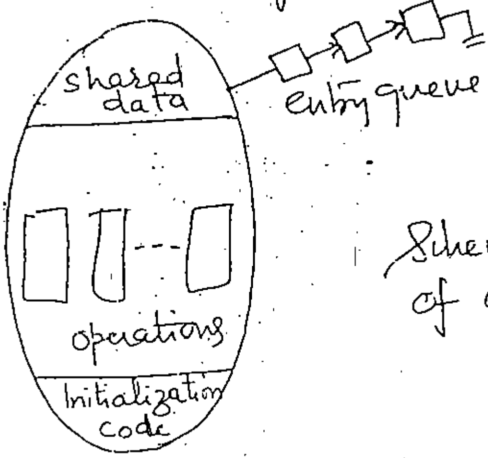
Monitors : A monitor is a high-level synchronizat construct. It is a characterized by a set of program defined operators. The representation of a monitor can not be used directly by the various processes. It consists of one or more procedures, an initializat sequence and local data.

The features of monitor are :

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 invoking one of its procedure.)

3) Only one process may be executing in the monitor at a time ; any other process that has invoked the monitor is suspended, waiting for the monitor to become available. The programmer does not need to code this synchronization constraint explicitly.

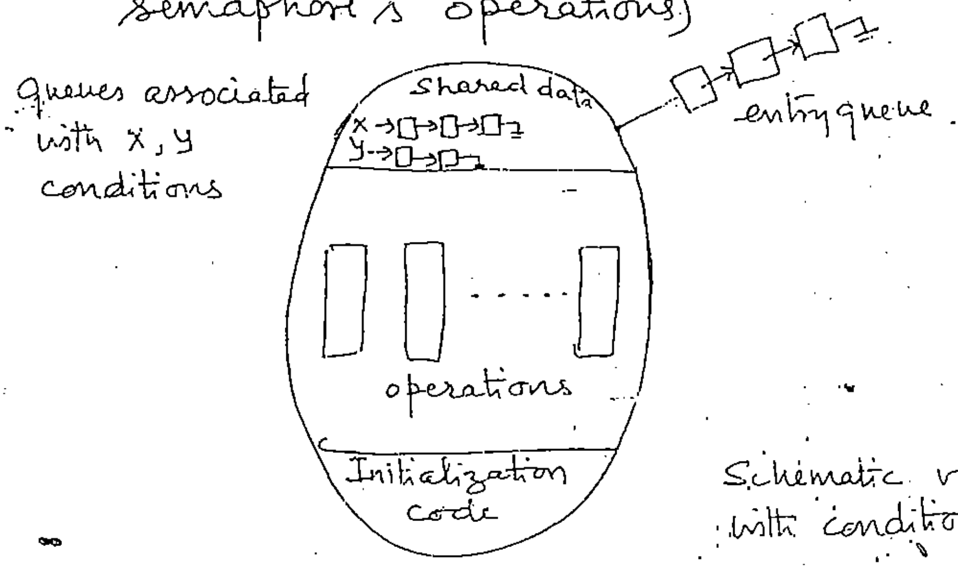


Schematic representation of a monitor.

To make monitor more powerful, additional synchronization mechanisms can be added to the monitor. A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Two functions operate on condition variables :

- wait() : suspend execution of the calling process on condition if condition is not satisfied. The monitor is now available for use by another process.
- signal() : Resume execution of some process suspended after a wait() on same condition. If there are several such processes, one is chosen. If there is no such process, do nothing.

(These wait() and signal() are different from semaphore's operations)



Schematic view of a monitor with condition variables

Dining-Philosopher problem : (Solution through monitor)

"A philosopher is allowed to pick up her chopsticks only if both of them are available."

For this solution, following data structure is declared:

① enum {thinking, hungry, eating} state [5];

Philosopher i can set the variable state [i] = eating only if her two neighbours are not eating:

(state [(i+4) % 5] != eating) &

(state [(i+1) % 5] != eating)

2) condition self [5];
where philosopher i can delay herself when she is hungry but is unable to obtain the chopstick she needs.

```
dp.pickup (i);
    ....
    eat
    ....
dp.putdown (i);
```

```
monitor dp
{  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();
       }
     }

   void init () {
     for (int i=0; i<5; i++)
         state [i] = thinking;
   }
}
```

```
monitor monitor-name
{ shared variable declarations
  procedure body P1 (----)
  {
    ----
  }
  procedure body P2 (----)
  {
    ----
  }
   .
   .
   .
  procedure body Pn (---)
  {
    ---
  }
  { Initialization code
  }
}
```

General syntax of a monitor.

OS 1.

Unit 4 (Chapter-8)

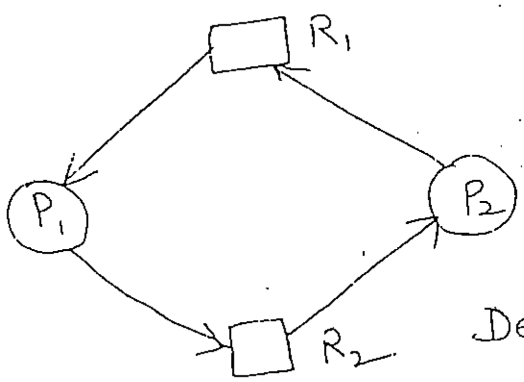
Deadlocks

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 deadlock is a situation where a group of processes are permanently blocked because each process holds a subset of resources needed to complete the execution and waiting for release of the remaining resources held by others in the same group."

A deadlock can occur in concurrent environment as a result of uncontrolled granting of system resources to requesting processes.

Example: Two processes P_1 & P_2 are running simultaneously. P_1 requests for resource R_1 and P_2 requests ~~for requests~~ for resource R_2. Both requests are granted. Now, P_1 requests for R_2 and P_2 requests for R_1. Both processes can not finish the execution because P_1 is asking for R_2 which is held by P_2 and P_2 is asking for R_1 which is held by P_1



Deadlock Condition

Neelam Bawane

- System Model:
  - A system consists of a finite numbers of different types of resources (physical or logical) for the competing processes. e.g. memory space, CPU cycl files, I/O devices.
  - Each resources may have some number of identical ~~resources~~ instances e.g. 2 CPUs means resource type CPU has two instances.
  - Identical instances are those instances when allocation of any (resource type) instance will satisfy the request.
  - For non-identical instances, separate resource classes should be defined.
  - A process must request a resource before using it and must release the resources after using it. Thus, following sequence of operations occur to use any resource:

(i) Request - A process requests a resource, if resource is not available, process has to wait.

(ii) Use - The process uses the resource, if resource is available.

(iii) Release - The Process releases the resource when work is over.

- A process may request for any numbers of resources but it should not exceed the total number of resources.
- The request and release of resources are carried out through system calls e.g.

request/release device (for I/O devices)
open/close files
allocate/free memory.

Deadlock Characterization : A deadlock situation can arise if the following four conditions hold simultaneously in a system. If any of these condition is not true, deadlock is not possible.

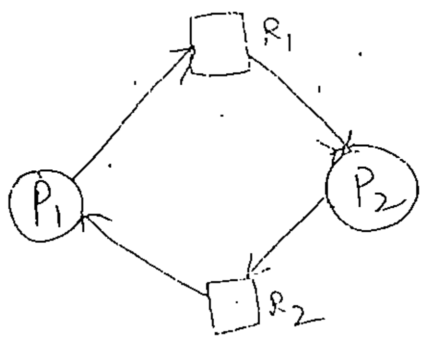
1) **Mutual Exclusion** : Only one process may use a resource at a time. The shared resources are used in a mutually exclusive manner. No two processes can use the ~~so~~ a single resource at a time.

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** : A resource can be released only voluntarily by the process, system can not preempt the resource.

4) **Circular Wait** : A closed (circular) chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

The first three conditions are necessary but not sufficient for a deadlock to exist. The fourth condition is must.



Resource - Allocation Graph: A process allocation graph is a directed graph which is used to represent or describe the deadlocks. Graph is consisting of following components:

1. Set of edges
2. Two Sets of nodes (i) Set of all active processes
   $P = \{P_1, P_2, \ldots P_n\}$
   (ii) Set of all resources types
   $R = \{R_1, R_2, \ldots R_n\}$

There are two types of edges
(i) Request edge $\underbrace{P_i \rightarrow R_j}$ (Process P_i is requesting for resource R_j)
(ii) Assignment edge $\underbrace{R_j \rightarrow P_i}$ (Process P_i is holding the resource R_j)

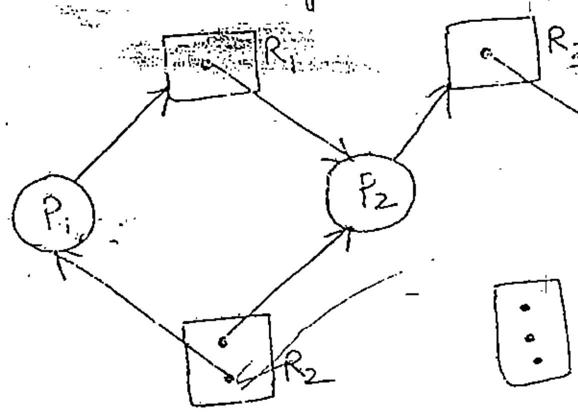
When a process requests for a resource, a request edge is inserted. When the resource is available, request edge is tranformed to assignment edge. When process does not need resource any more, resource is released and assignment edge is deleted.

Example 1: $P = \{P_1, P_2, P_3\}$
$R = \{R_1, R_2, R_3, R_4\}$
$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2$
$R_2 \rightarrow P_1, R_3 \rightarrow P_3$
Resources instances $R_1 = 1$ instance
$R_2 = 2$ "
$R_3 = 1$ "
$R_4 = 3$ "

Draw resource allocation graph stating process states. Whether system is deadlocked or not.

Process States:

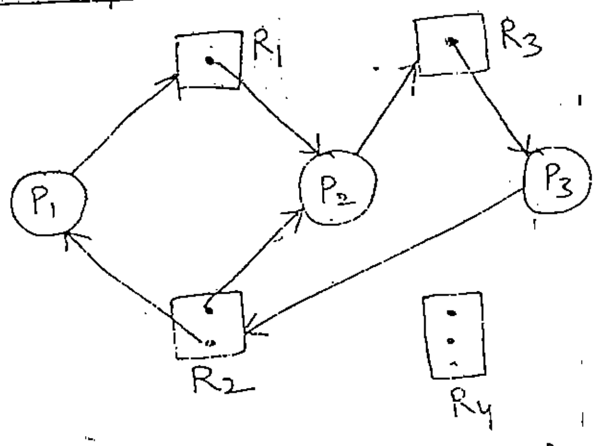
- P_1 is holding an instance of R_2
  P_1 is waiting for an instance of R_1
- P_2 is holding an instance of R_1 and R_2
  P_2 is waiting for an instance of R_3
- P_3 is holding an instance of R_3



Resource allocation graph

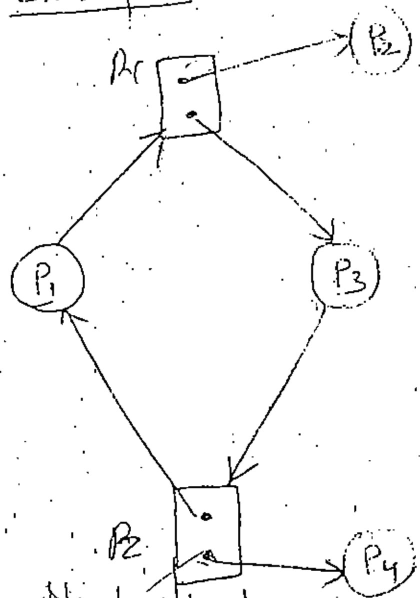
Since graph contains no cycle, deadlock is not there.

Example 2:



Resource allocation graph with a deadlock
(Cycle is present)

Example 3:



No deadlock
(Cycle is present)

Thus, if graph contains cycle – deadlock may or may not be there depending upon processes taking part in cycle.

Methods for handling Deadlocks

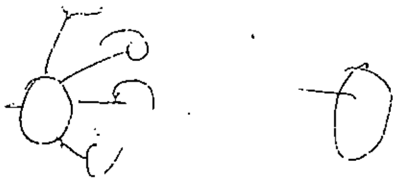
1. Ignore
2. Prevention
3. Avoidance
4. Detection & Recovery

1) Ignore: No technique is applied to avoid or detect the deadlock. We pretend that deadlock never occur in system. If system comes under deadlock condition, system performance will come down. More and more processes, as they make requests for resources, enter a deadlock state. The system will stop functioning and will need to be restarted manually.

2) Deadlock Prevention: The OS must eliminate one of the four conditions.

1. Mutual Exclusion: Sharing of resources should be allowed. All the resources can not be shared in any system such as tape drive, disk space, I/O devices or write permission to file. Thus, this method is not appropriate. Some resources like read permission of file can be shared. Printer can not be shared but spooling technique can help in sharing printer.

2. Hold and Wait: The hold and wait condition can be eliminated by forcing a process to release all resources held by it whenever it requests a resource that is not available. Thus, waiting processes will not hold any resources. There are two possible implementation:

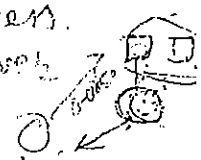


(1) The process requests all required resources prior to commencement of execution. Resource utilization may be low, since many of the resources may be allocated but unused for a long period.

(2) A process can request resources only when it has none. A process can request resources, use them, for additional resources it has to release resources.

If resources are popular, possibility of starvation is more.

3. No preemption Resources allocated to a process should be taken away forcibly from it whenever required. Two implementations are possible.



(i) If a process is holding some resources and requests another resource that can not be immediately allocated to it, then all the resources currently being held are preempted.

(ii) When a process requests some resources

- Check whether resources are available or not
- If resources are available, allocate them
- If not, check whether resources are allocated to other processes which are waiting for additional resources

so preempt the desired resources from waiting process and allocate them to the requesting process.

method is applicable only when state of resources can be saved and restored back if necessary e.g. CPU registers. But with tape drives, it is not possible.

64

4. Circular Wait : The circular-wait condition can be prevented by defining a linear ordering of resource type following steps can be ~~fo~~ implemented.

- Impose a total ordering of all resources types.
- Each process should request resources in an increasing order. e.g. tape drive = 0
  disk drive = 1
  printer = 2
  plotter = 3
- ~~If P1 holds tape~~ drive (0), needs printer (2)
  P_2 holds printer (2), needs tape drive (0).

→ Only P_1 is elligible to request for printer because tape drive (0) < printer (2)

→ P_2 can not request for tape drive (0) because printer (2) ≮ tape drive (0).

Drawbacks of deadlock prevention

1. All the resources whether external or internal need to be numbered which is a tedious job.
2. Low device utilization
3. Reduced system throughput

3) Deadlock avoidance : Avoidance allows the three conditions mutual exclusion, hold and wait, no preemption. A decision is made dynamically whether the current resource allocation request will lead to deadlock.

Two approaches can be followed :

- Do not start a process if it demands may lead to deadlock.
- Do not grant an incremental resource req to a process if this allocation may lead to deadl

Safe State: A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock.

Or a system is in a safe state only if there exists a safe sequence (a possible sequence of processes for execution)

e.g. $P_1, P_2 \dots P_n$ has a safe sequence if for each process (P_i), required resources can be granted in some order.

An unsafe state may lead to deadlock. Not all unsafe states ~~the~~ ~~OS~~ are deadlocks. In an unsafe state, the OS cannot prevent processes from requesting resources such that a deadlock occurs.

Avoidance algorithm:

- System should always be in safe state to ensure no deadlock occurs.
- Request can be granted if the allocation leaves the system in a safe state.
- Thus, if a process requests a resource that is currently available, it may still have to wait.

Drawback — * Low resource utilization
* Less throughput

Example 1: A system has 12 magnetic tape drives and three processes.

	Max Needs	Drives held at t_0	future Need	Available drives in system = 3
P_0	[illegible]	5	5	
P_1	4	2	2	
P_2	9	2	7	

P_0 need (5) > Available (3), it can not execute

P_1 need (2) < Available (3), it can execute it releases 2 drives after finish Available becomes 5

P_2 need (7) > Available (5), it can not execute

Again P_0 need (5) = Available (5), it executes and releases 5 and available becomes 10

P_1 need (7) < Available (10), it also executes

thus safe sequence $= \langle P_1, P_0, P_2 \rangle$

Example 2: If system has 12 magnetic tape drives and three processes

	Max Need	Drives held	future Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

Available drives in system = 2

P_0 need (5) > Available (2), P_0 can not executes

P_1 need (2) = Available (2), P_1 executes, & release and available = 4

P_2 need (7) > Available (4)

P_0 need (5) > Available (4) — Execution not possible

No safe sequence, thus system is unsafe.

66

Resource - Allocation Graph Algorithm

for one instance of each resource type, a variant of the resource - graph can be used for deadlock avoidance. There are following components

- a set of resource nodes
- a set of process nodes
- a set of edges — claim edge $P_i \dashrightarrow R_j$
  request edge $P_i \longrightarrow R_j$
  assignment edge $P_i \longleftarrow R_j$

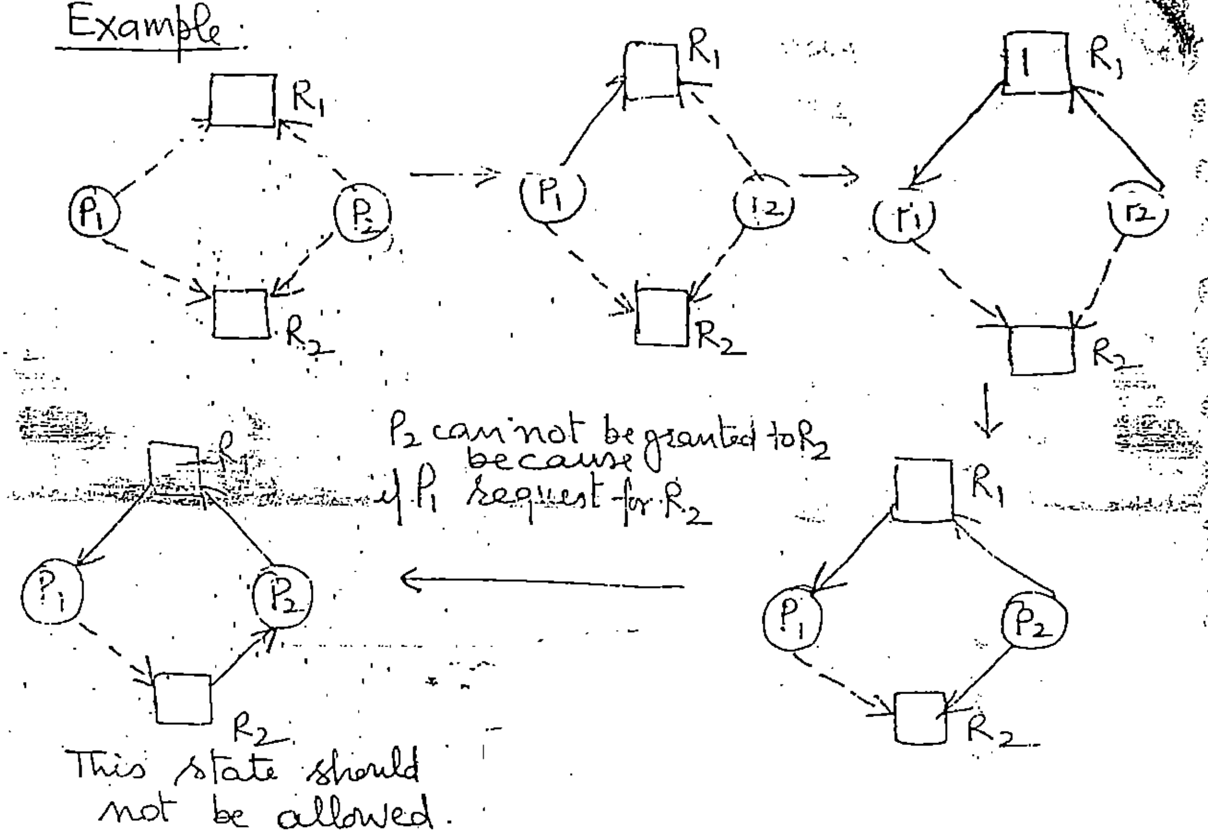
Steps to draw graph:

* first, draw all claim edges
* Convert claim edge into request edge whenever request is made such that no cycle should occur
* Convert request edge into assignment edge whenever resource is free ~~and no~~ such that no cycle should occur.
* When resource is released, assignment edge is reconverted into claim edge.

Two conditions:

1) When a process starts execution, its all claim edges should appear
2) New claim edge can be added if all edges are claim edges. (No request & assignment edge has been entered)

Example.



Banker's Algorithm

1. When new process enters the system, it must declare the maximum numbers of instances of each resource type that it may need.
2. This number should not exceed the total no. of resources in the system.
3. When a request is made, system must determine whether allocation of this request leaves the system in safe state or not. This can be checked through two algorithm:
(i) Resource - request algorithm (to update data structure
(ii) Safety algorithm (to check system is safe or not

Implementation of Banker's algorithm

Different data structures

If no. of processes in system = n
no. of resource types = m

- **Max** → $n \times m$ matrix (maximum demand of each process)

e.g.

	R_0	R_1	R_2
P_0	3	1	4
P_1	2	4	3

If $Max[i,j] = k$
then P_i can request at most k instances of resource type j

- **Allocation** → $n \times m$ matrix (No. of resources of each type currently held by different processes)

e.g.

	R_0	R_1	R_2
P_0	2	0	2
P_1	1	3	2

If $Allocation[i,j] = k$
then P_i ~~now~~ is holding k instances of resource type j

- **Need** → $n \times m$ matrix (remaining requirement of each process)

If $Need[i,j] = k$
then P_i needs k instances of resource type j

e.g.

	R_0	R_1	R_2
P_0	1	1	2
P_1	1	1	1

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

- **Available** → A Vector length of m (no. of available resources of each type)

If $available[j] = k$
then there are k instances of resource type j

$Max_i \rightarrow$ Maximum requirement of process i for resources of different type.

$Allocation_i \rightarrow$ Resources currently held by process i

$Need_i \rightarrow$ Resources currently needed by process i

A) Resource-Request Algorithm

$Request_i \rightarrow$ Request vector for P_i

When a request for resources ~~is made~~ by process i the following actions are to be taken:

1) If $Request_i \leq Need_i$, go to step 2 otherwise raise error

2) If $Request_i \leq Available$, go to step 3 otherwise P_i must wait

3) If system allocate the request i to process i following states should be modified:

$Available := Available - Request_i$;

$Allocation_i := Allocation_i + Request_i$;

$Need_i := Need_i - Request_i$;

4) Check the data structures through Safety algorithm

If resulting data structures can produce safe sequence, process i gets allocation of request i resources.

→ If new state does not have a safe sequence, process i must wait for Request i resources and old state of data structures (before allocation) is restored.

(B) Safety Algorithm

Work ⇒ Vector of length m (resources)
Finish ⇒ Vector of length n (Processes)

1) Initialize
Work := Available
Finish[i] := false for i = 1, 2 . . . . n

2. For an i (process) such that both a and b should be true

(a) finish [i] = false
(b) $Need_i \le Work$

If no such i exists go to step 4
If i exists go to step 3

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
Otherwise system is unsafe

Tip:

To solve the problems:

- Need = Max − Allocation
- 1) check $Need_i \le Available$
  2) Execute or $finish_i$ = true,
  Available = Available (old) + $Allocation_i$
  3) Find all processes finish = true or not and generate safe sequence.

Example 1:

for Resource types A B C maximum instances present in the system are A = 10, B = 5, C = 7

(1) Calculate need matrix

(2) Whether system is safe or not

(3) If a request from a process P_1 arrives for (1,0,2). Can the request be granted immediately

(4) If a request from a process P_4 arrives for (3,3,0) (in original data structures). Can the request be granted immediately?

~~Answer~~

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Ans 1) Need Matrix

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

request for Process P_1 (1 0 2)

Available - request of P_1

332 - 102 = 230

Key

Ans(2) P_0 (Need$_0$) $743 \nleq 332$ (Available)

finish$_0$:= false

P_1 (Need$_1$) $122 \leq 332$ (Available)

finish$_1$:= true

Available := $332 + 200$ (Allocation)

$= 532$

P_2 (Need$_2$) $600 \nleq 532$ (Available)

finish$_2$:= false

P_3 (Need$_3$) $011 \leq 532$ (Available)

finish$_3$:= true

Available := $532 + 211$ (Allocation$_3$)

$= 743$

P_4 (Need$_4$) $431 \leq 743$ (Available)

finish$_4$:= true

Available := $743 + 002$

$= 745$

P_0 (Need$_0$) $743 \leq 745$ (Available)

finish$_0$:= true

Available := $745 + 010$

: 755

P_2 (Need 2) $\cancel{122}$ 600 ≤ 755 (Available)

finish$_2$:= true

Available := $755 + 302$

$= 10\ 5\ 7$

Safe Sequence $= \langle P_1, P_3, P_4, P_0, P_2 \rangle$

System is safe.

Ans (3)

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Safe sequence: $< P_1, P_3, P_4, P_0, P_2 >$

Ans (4)

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	0 0 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	3 3 2	4 3 3	1 0 1	

With Available 0 0 2 No process can execute because for all i Need $\nleq$ Available

No safe sequence occurs which implies ~~A~~ System is not safe.

Deadlock detection

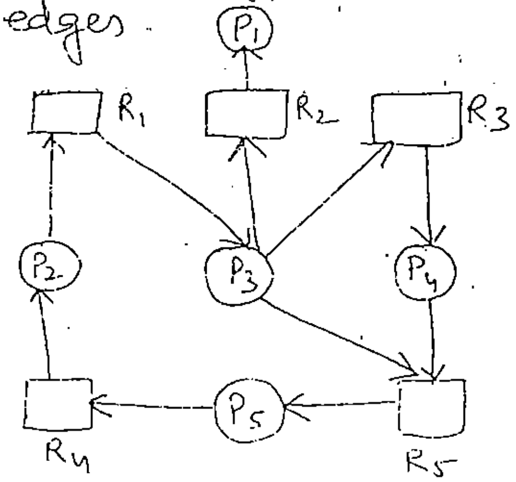
If the system does not use a fully effective deadlock prevention or avoidance policy, Then a deadlock may occur. A deadlock detection strategy accepts the risk of a deadlock occurring and periodically executes a procedure to detect it. Hence the system must provide:

- An algorithm to check the occurrence of deadlock.
- An algorithm to recover from deadlock.

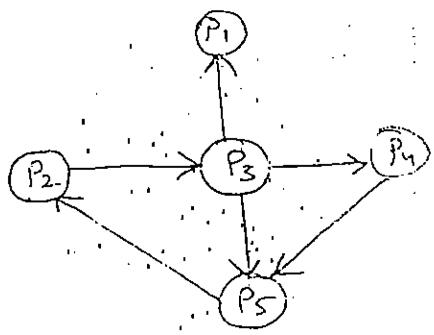
1) Single Instance of Each Resource type

A deadlock detection uses a slightly modified version of resource allocation graph, known as wait-for graph.

This graph is obtained from the resource allocation graph, ~~known as~~ by removing the resource type nodes and collapsing the appropriate edges.



Resource-allocation graph



Corresponding Wait-for graph

An edge $P_i \rightarrow P_j$ implies that process P_i is waiting for P_j to release a resource for which P_i is waiting and P_j is holding. In this example cycle exists in wait-for graph, Thus deadlock occurs.

2) Several Instances of each Resource type

It is similar to Banker's algorithm

Datastructures

- $\boxed{\cdot \text{ Available}}$ ⇒ A vector of length m

e.g.

A	B	C
2	4	1

- $\boxed{\cdot \text{ Allocation}}$ ⇒ n x m matrix (no. of resources instances currently held by different processes)

e.g.

	A	B	C
P_0	3	1	0
P_1	1	2	3

- $\boxed{\cdot \text{ Request}}$ ⇒ n x m matrix (no. resources instances currently requested by different processes)

$Allocation_i$ ⇒ Vector, resources currently held by P_i

$Request_i$ ⇒ Vector, resources currently requested by P_i

Deadlock detection algorithm

1) Work = Vector of length m

Finish := Vector of length m

Initialize Work := Available

If Allocation ≠ 0 then Finish[i] := false

otherwise finish[i] := true

2) Find an i such that both a) & b)

a) finish[i] = false

b) $Request_i \leq Work$

If no such i exists, go to step 4

If any i exists go to step 3

3) Work := Work + Allocation i

Finish[i] := true

go to step 2

[Assumption: if $request_i \leq work$, process finishes allocation will be released, and work will be updated as $Work = Work + Allocation_i$]

4) If finish [i] = false for any i
then system is in a deadlock state
If finish[i] = true for all i
then system is in a safe state.
The processes for which finish[i] = false
is called deadlocked.

Example: A = 7, B = 2, C = 6

	Allocation A B C	Request A B C	Available A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Solution:

For $P_0 \rightarrow$ $Request_0(000) \leq Available(000)$

finish$_0$:= true

Available: 000 + 010 (Allocation$_0$) =

for $P_1 \rightarrow$ $Request_1(202) \nleq Available(010)$

finish$_1$:= false

for $P_2 \rightarrow$ $Request_2(000) \leq Available(010)$

finish$_2$:= true.

Available := 010 + 303 (Allocation$_2$)

= 313

for $P_3 \rightarrow$ $Request_3(100) \leq Available(313)$.

finish$_3$:= true

Available := 313 + 211 (Allocation

= 524

for $P_4 \rightarrow$ $Request_4(002) \leq Available(524)$

finish$_4$:= true

Available := 524 + 002 (Allocation$_4$

= 526

for $P_1 \rightarrow$ $Request_1(202) \leq 526$ Available.

finish$_1$:= true

Available := ~~526~~ 526 + 200 (Allocation$_1$)

= 726

Safe sequence = $\langle P_0, P_2, P_3, P_4, P_1 \rangle$ system is in safe state, not deadlocked.

Deadlock-detection Algorithm Usage

The detection algorithm depends on two factors — Time to invoke

1, frequency of deadlock occurrence
2, Number of processes affected by deadlock

~~At the deadlock time, resour~~

As we know:

- Deadlock makes the resources idle.
- Number of processes may grow in deadlock
- Any request may create deadlock condition

Thus, if deadlock occurs frequently and many processes are getting affected, detection-algorithm should be run frequently.

1) Whenever a process ~~~~ requests, we can invoke detection algorithm.

Advantages: Deadlock can be detected as soon as it occurs, thus, low resource utilization can be avoided.

Disadvantages: It will increase ~~in~~ overhead in computation time.

2) Algorithm can be invoked at less frequent intervals eg. once per hour, when CPU utilization drops down 40%

Advantages: Computation overhead will be less

Disadvantages: Low Resource utilization and less throughput will be caused. Partial computation finished by processes will be lost.

Recovery from Deadlock: Once a deadlock situation has been identified by the detection algorithm. Any of the following actions can be taken:

1. Recover manually.
2. System recovers automatically: There are two ways: a) Process termination
   b) Resource preemption

a) Process Terminations: The system reclaims all resources allocated to a process by aborting the process. Hence deadlock is eliminated.

(i) Abort all deadlocked processes: This is very expensive since processes may have computed for long time, and all the partial results are are discarded

(ii) Abort one process at a time: This process is repeated until the deadlock cycle is eliminated. This is also very expensive, since after each process termination, a deadlock detection algorithm must be invoked.

For a given set of deadlocked processes, we should find out which process should be terminated to break the deadlock. There are various factors which governs the choice of process to be aborted.

- priority of process
- How much computation is completed by process and how much remaining
- How many or what type resources are already used by process.
- How many more resources are needed
- Nature of process - Interactive or batch
- How many resources need to be terminated.

13

73

b) Resource Preemption By preempting some resources from processes and reallocating these resources to other processes, the deadlock can be eliminated. There are three main issues:

1) Select a Victim: which process and which resources need to be preempted.

2) Roll back: ~~when a~~ process is preempted, the process cannot continue its normal execution, hence, the process has to be rollbacked to some safe state and restart it from that state.
Since it is difficult to decide safe state, generally total rollback is allowed and process is restarted again.

3) Starvation: Resources may be preempted from same process again and again due to low priority. Thus, a particular process may feel starvation.

(for ex) instance of resource is 7,7,10.
& he has given current allocation & max.
we can find ~~Allocation~~ Available
is given below.

Available = Num of instance − sum of allocation.

	Current allocation R_1	R_2	R_3
P_1	2	2	3
P_2	2	0	3
P_3	1	2	4
	5	4	10 ← sum of allocation

Available = 7 7 10 − 5 4 10

Available = 2, 3, 0

current allocation R_1	R_2	R_3	Max			Available		
2	2	3	3	6	8	2	3	0
2	0	3	4	3	3			
1	2	4	3	4	4			