Ch-6. Process & Threads.

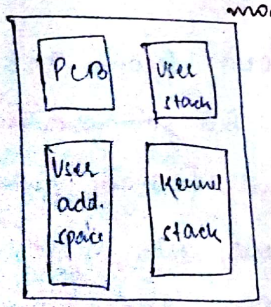
If our system support multithreading, then a process is fragmented into a no. of threads and then there is thread scheduling not process scheduling.

For a thread, generally 3 states — ready, running, blocked.

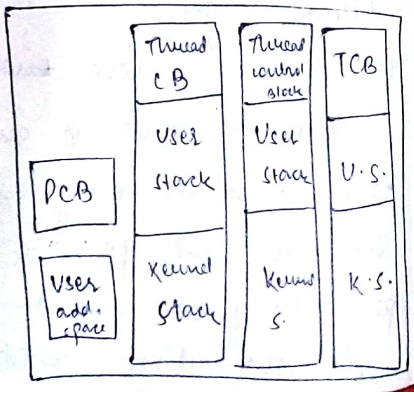
All the threads of a process share the address space of the process and the resources used by / alloted to the process.

Within a process, each thread has its own program counter, along with the process having its PC.

Single-thread process model



Multi-thread process model.



Operating System allows a thread utility which takes the backup of the thread currently running.

Multiprocessing and multithreading system increases the execution speed.

Date : 03/10/18.

Categories of Thread implementation. (User-level / Kernel-level)

- Thread shares user address space.
- Thread management is done by the application, not by the kernel.
- User-level impl.
- Kernel-Level implementation.

Date : 03/10/18

Process synchronization.

T_1 :	T_2 :
read (A) //1000	read (A) //1000
A = A - 50 //950	temp = A * 0.1 //100
write (A) //950	A = A - temp //900
read (B) //2000	write (A) //900
B = B + 50 //2050	read (B) //2000
write (B) //2050	B = B + temp //2100
	write (B) //2100

T_1	T_2
read (A) // 1000	
A = A - 50 // 950	
write (A) // 950	
read (B) // 2000	
B = B + 50 // 2050	
write (B) // 2050	
	read (A) // 950
	temp = A * 0.1 // 95
	A = A - temp // 855
	write (A) // 855
	read (B) // 2050
	B = B + temp // 2145
	write (B) // 2145

Assume that A and B are the amounts initially available in accounts 1 and 2 respectively. A = 1000, B = 2000

- Four properties of a database system —

① Atomicity
② consistency
③ Isolation
④ Durability

ACID

As in this case, before executing threads T_1 & T_2 A+B = 3000 and after executing threads, still A+B = 3000. So, the transaction is consistent.

In the 2nd case, threads T_1 and T_2 are in serial execution not in concurrent execution (as in 1st case). In the 2nd case, the threads are scheduled.

Here, in 2nd case: initially, A+B = 3000

finally, A+B = 855 + 2145
= 3000

Hence, this scheduling is also consistent here.

T_1	T_2	T_2
read(A)//1000		read(B)//2050
A=A-50 //950		B = B+temp//
write(A) //950		write(B) //2145
	read (A) //950	
	temp = A * 0.1 //95	
	A = A - temp // 855	
	write(A) //855	
read(B)//2000		
B = B+50 //2050		

A + B = 855 + 2145
= 3000

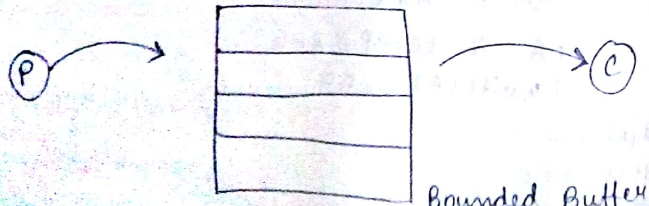
T_1	T_2
read (A) // 1000	
A = A - 50 // 950	
	read (A) // 1000 (जबतक update नहीं होगा तबतक old value ही रहेगी)
	temp = A * 0.1 // 100
	A = A - temp // 900
	write (A) // 900
	read (B) // 2000
write (A) // 950	
read (B) // 2000	
B = B + 50 // 2050	
write (B) // 2050	
	B = B + temp // 2000 + 100 = 2100
	write (B) // 2100

so, finally A+B = 950 + 2100 = 3050 ≠ 3000

so, not consistent.

- Synchronisation Problems.

① Producer - consumer problem.



Bounded Buffer (Unlimited capacity)

If unlimited capacity then, unbounded buffer.

Producer produces some items and puts them in the buffer from where the consumer consumes the item.

In case of bounded buffer, if production rate is more than consumption rate then producer may have to wait in case the buffer gets full. Similarly, if the consumption rate is more than the production rate then the consumer may have to wait. This is the producer - consumer problem.

- Solutions of this problem.

① Producer:

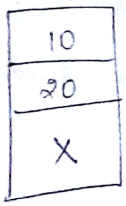
```
item nextProduced;
while (true)
{ while (((in+1) % Buffer.size)
  == out) ; // do nothing as the buffer is full
buffer[in] = nextProduced;
in = (in+1) % Buffer-size;
}
```

Consumer : item nextConsumed ;
while (true)
{ while (in == out) ; // do nothing
nextConsumed = buffer [out] ;
out = (out+1) % Buffer-size ;
}

• Limitation of this solution.

We are able to utilise only (Buffer-size - 1) capacity, not the complete Buffer-size.

eg.



→ this space can't be utilised as per the producer code.

Updated ~~Producer~~ code :—

Producer : item nextProduced ;
while (true)
{ while (counter == Buffer-size);
//do nothing.
buffer [in] = nextProduced ;
in = ((in+1) % Buffer-size;
counter ++ ; }

Consumer : item nextConsumed ;
while (true)
{ while (counter == 0) ; // do nothing
nextConsumed = buffer [out] ;
out = (out+1) % Buffer-size ;
counter -- ;
}

• counter variable is shared b/w producer & consumer.

• Limitation of this updated solution.

counter ++ and counter -- are high level instructions. Their machine-level implementations are —

counter ++		counter --	
I_1 :	R_1 = counter ;	I_4 :	R_2 = counter ;
I_2 :	R_1 = R_1 + 1	I_5 :	R_2 = R_2 - 1
I_3 :	counter = R_1	I_6 :	counter = R_2

If we assume the sequence of operation as → $\langle I_1^3, I_2^3, I_4^3, I_5, I_3, I_6 \rangle$

- counter is shared variable so, it should be like (y-1 = 3 in case of counter++ due to concurrency) But it is 2. so, ambiguity.

So, we need ~~contr~~ some control. One way of control is called critical-section. Critical-section is a ~~piece~~ part of code where we are working on shared variable.

- So, structure of a code is —

Pi
- Entry Section // locking
- Critical Section
- Exit Section // Unlocking
- Remainder Section

In order to control the critical section, we need some constraints, these constraints are defined in entry section and are released (or freed) in exit section.

If the code has many no. of parts where shared variables are being used then, we can define that many no. of critical sections.

Date : 5th Oct, 2018.

Solution of a critical section is also in the form —

- Entry section
- Critical section
- Exit section
- Remainder section

There are 3 conditions that must be satisfied by soln of a critical section —

1) Mutual Exclusion
2) Progress
3) Bounded wait.

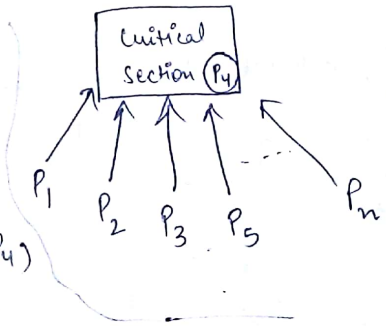
- 

Mutual exclusion :— If more than one processes attempt

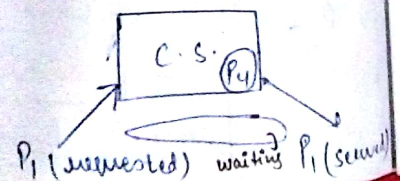
to perform write operation on the shared variable then mutual exclusion ensures that at a time, only one process is allowed to perform the write.

- Progress.

If one process is already in the critical section (P_4) and many other processes are making request to enter the critical section then after the P_4 leaves the C.S. then decision should be made as to which process should be next put to C.S. and this decision should be taken in finite time otherwise all other processes will have to wait for infinite time.



- Bounded wait



The waiting duration of P_i is bounded, i.e., the number of process executions between the request of P_i (say) and the time when P_i (say) got the chance to be executed, should be finite.

P_i : while turn $\neq i$; // do nothing

// critical section

turn = j

Entry section

turn [0] initialization of turn with 0

This is 2-process critical section soln with single shared variable.

$\langle P_0, P_1, P_0, P_1, P_0, \ldots \rangle$

Hence, mutual exclusion working.

There is no progress and bounded wait because P_i waiting for infinite time

---

So, updated solution :—

(P.T.O.)

P_i :

```
while flag[i] ; // do nothing
flag[i] = True ;
```

// critical section

```
flag[i] = false ;
```

Two processes : P_i and P_j s.t. $j = 1 - i$

Here, a flag array taken and initialized as —

false	false

Flag: [0] [1]

In this, mutual exclusion is also violated along with the other two conditions.

- Peterson's Solution.

P_i: do {

```
Flag[i] = TRUE ;
turn = j ;
while (Flag[j] && turn == j);
```

// critical section

```
Flag[i] = FALSE ;
```

do nothing

// Remainder section

} while (TRUE).

This is also a two-process solution.

we have : Flag

[0]	[1]
F	F

and a shared variable turn 0 .

Flag

F	F
T	T

Turn

0
1 0

Execution steps

// critical section P_0

P_1 waiting in while loop.

when P_0 leaves critical section then Flag[0] = F

Here, mutual exclusion working.

Progress is also being carried out & hence turn is also bounded wait.

- Homework solution.

```
boolean TestandSet (boolean * target)
{ boolean rv = * target ;
```

```
* target = TRUE ;
return rv ;
}
```

* test and set instruction is used to write 1 (set) to a memory location and return its old value as a single non-interruptive operation

```
do {
    waiting[i] = TRUE ;
    key = TRUE ;
    while ( waiting[i] && key ) do
        key = TestandSet ( & lock ) ;

    // critical section

    j = (i+1) % n ;
    while ( (j != i) && (!waiting[j]) )
        j = (j+1) % n ;
    if (j == i) then lock = FALSE ;
    else waiting[j] = FALSE ;

    // RS

} while (TRUE)
```

* If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, then no other process can begin another test-and-set until the first one is finished

We have :—

Global : waiting [0] [1] [2] [3] ...

F	F	F	F	...

lock F

Local : key F

In this solution, mutual exclusion is ensured, progress is to working and bounded wait also exists.

* case with multiprocessor system.

enable <interrupts>
use
disable <interrupts>

P_1 P_2 P_3 P_4 P_5 P_6

information needs to be passed to all processors individually.

Date : 09/10/18.

Semaphore.

It is a synchronization tool / some integer variable.
Two operations are defined on that variable —
(i) wait (s) (ii) signal (s)
↓
Semaphore variable.

(i) wait (s)
{ while s ≤ 0 ; // do nothing
s-- ;
}

(ii) signal (s) { s++ ; }

Types of semaphore —
(i) Counting Semaphore (s can take any value)
(ii) Binary Semaphore (when s = 0 or 1).
↓
used when we need to perform locking and unlocking (mutual exclusion)

Counting semaphore is used in the case when a resource has a no. of instances.
eg. say resource R has instances $R_1, R_2, R_3, R_4, \ldots$ s initialized with the no. of instances & then operation performed.

Some more applications of Semaphores.

$P_1 \rightarrow I_1$
$P_2 \rightarrow I_2$
$I_1 \rightarrow I_2$ (I_1 instruction executed before I_2)
$P_1 \rightarrow P_2$ (P_1 process executed before P_2).

Write a code for this case using semaphores

$s = 0$

① $P_1 : I_1$
signal (s)

~~② $P_2 : I_2$~~

② wait (s)
$P_2 : I_2$

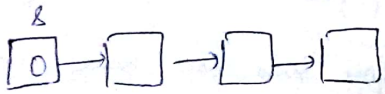
Extension of this problem for two more processes.

③ wait (s)
$P_3 : I_3$

④ wait (s)
$P_4 : I_4$

This type of waiting is called Busy waiting.

This is called busy waiting because in this waiting, CPU cycle is consumed. Initially, all the four processes are in ready state inspite of this; three processes are waiting in their while loop which checks the processes if it is free or not. This processor checking involves CPU cycles. To avoid this, the processes are placed in the waiting queue of the semaphore variable, & change their state to blocked state.



So, we need to redefine semaphore —

- Redefinition of semaphore.

(i) wait (semaphore * s)

```
{  s → value --
   if ( s → value < 0)
     { add this process to s → list;
       block ( );
     }
}
```

(ii) signal (semaphore * s)

```
{  s → value ++;
   if ( s → value <= 0)
     { remove a process P from
       s → list ;
       wakeup ( P) ;
     }
}
```

(iii)

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

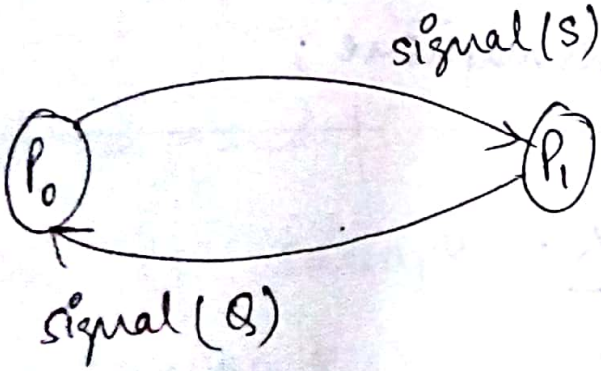
DEADLOCK.

P_0	P_1
wait (S)	wait (Q)
wait (Q)	wait (S)
:	:
signal (S)	signal (Q)
signal (Q)	signal (S)

If semaphores are not used in proper manner then, some problems like deadlock may occur.

eg. suppose S, Q → 1, then

P_0	P_1
wait (S) // S--	wait (Q) // Q--
wait (Q) // Q=0 so waits in while loop	wait (S) // S=0 so waits in while loop
signal (S)	signal (Q)
signal (Q)	signal (S)



P_0 is waiting for signal(S) and P_1 is waiting for signal(Q) but both are waiting indefinitely. so, neither of them executes.

Semaphore Solution of Producer - Consumer Problem using Semaphores.

eg. full = 0, empty = n, mutex = 1 (mutual exclusion)

```
Producer : do {
    // produce an item in nextp
    . . .
    wait(empty);
    wait(mutex);
    . . .
    // add nextp to buffer
    . . .
    signal(mutex);
    signal(full);
} while(TRUE);

Consumer : do {
    wait(full);
    wait(mutex);
    . . .
    // Remove an item from buffer to nextc
    . . .
    signal(mutex);
    signal(empty);
    . . .
    // consume the item in nextc
} while(TRUE);
```

- 'empty' tells the state of buffer.
'full' tells the no. of items in buffer.
'mutex' is shared variable.

Reader - Writer Problem

```
Reader : do {
    wait(mutex);                    -> mutex, wrt -> readcount = 0
    readCount++;
    if(readcount == 1)              locking
        wait(wrt)
    signal(mutex)
    . . .
    // reading is performed
    . . .
    wait(mutex)
    readCount--;
```

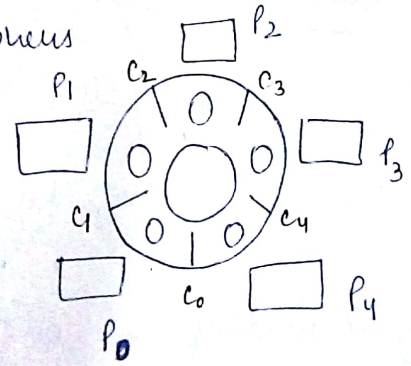
- readers process shares the semaphores mutex & wrt & the integer readcount. Semaphore wrt is also shared with the writer processes

locking
```
if (readCount == 0)
    signal (wrt)
signal (mutex)
} while (T);
```

Writer :
```
do { wait (wrt);   // this is actually the previous wait marked.
   . . .
   // writing is performed
   . . .
   signal (wrt);   // this is also the previous signal marked.
} while (T);
```

Dining Philosopher Problem.

There are 5 philosophers and 5 chopsticks (or spoons) represented by lines. In the middle, a bowl of rice is kept. A philosophes eats by using two chopsticks at a time. How to handle the case when two adjacent philosophers are hungry at the same time.



Solution : —

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

But this is not a proper solution because deadlock situation may arise in the case when all philosophers attempt to choose left chopstick first and then the right chopstick.

MONITOR.

①	②
signal (mutex) ... // critical section wait (mutex)	wait (mutex) ... // critical section ~~signal~~ wait (mutex).

Semaphores are not always useful as they undergo "timing error" when not used properly.

In the case ①, many processes are allowed to enter the CS (as mutex = 1 (initialisation) changes to mutex = 2).

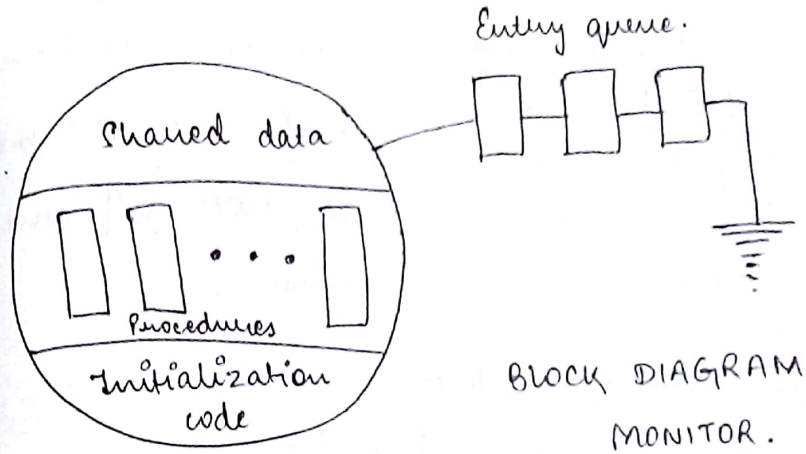
In case ②, deadlock occurs. They are solved using monitors.

Monitor is an abstract data type (ADT).

Private data is accessed through Public methods.

Syntax for definition of monitor:—

```
monitor <monitor name>
{   // shared data
    Procedure 1 ( ) {  }
    Procedure 2 ( ) {  }
        . . .
    Procedure 3 ( ) {  }
    InitializationCode ( ) {  }
}
```



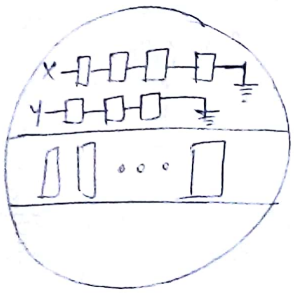
BLOCK DIAGRAM OF MONITOR.

Property of Monitor:—

If more than one processes are interested to do operation on the shared data using procedures, then at a time, monitor allows only one process to execute the monitor code & perform the operation. All other processes are put in the queue

the processes waiting in the queue may undergo starvation. To avoid this, we use conditional variable.

Inspite of putting the processes in the queue outside the monitor, we can put them inside the monitor as per some condition.



x and y are two different conditions and are the basis of two different queues.

There are also two methods — wait() and signal().

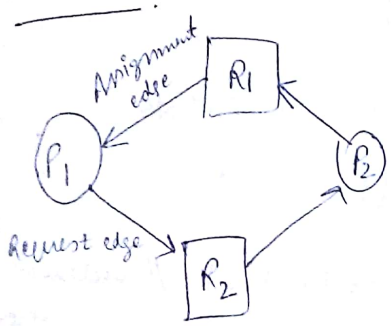
- wait() method here is same as that in semaphores. But signal() method varies because in case of semaphores, the semaphore variable was always incremented (s++). But in case of monitors, semaphore variable is incremented only when some process is waiting otherwise there is no change in the variable.

solution of Dining Philosophers Problem using Monitor.

```
monitor dp
{ enum { THINKING, HUNGRY, EATING } state[5];
  condition self[5]; // conditional variable.
  void pickup(int i)
  { state[i] = HUNGRY;
    test(i);
    if(state[i] != EATING)
      self[i].wait();
  }
  void putdown(int i) // releases the chopsticks.
  { state[i] = THINKING;
    test((i+4)%5);
    test((i+1)%5);
  }
  void test(int i) // used to check whether both chopsticks available
  { if((state[(i+4)%5] != EATING)
      && (state[i] == HUNGRY) &&
    (state[(i+1)%5] != EATING))
    { state[i] = EATING; self[i].signal();
    }
```

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

DEADLOCK :



In deadlock, resource utilisation is very poor.

Following conditions ensure that our system is in deadlock state —

(1) Mutual Exclusion

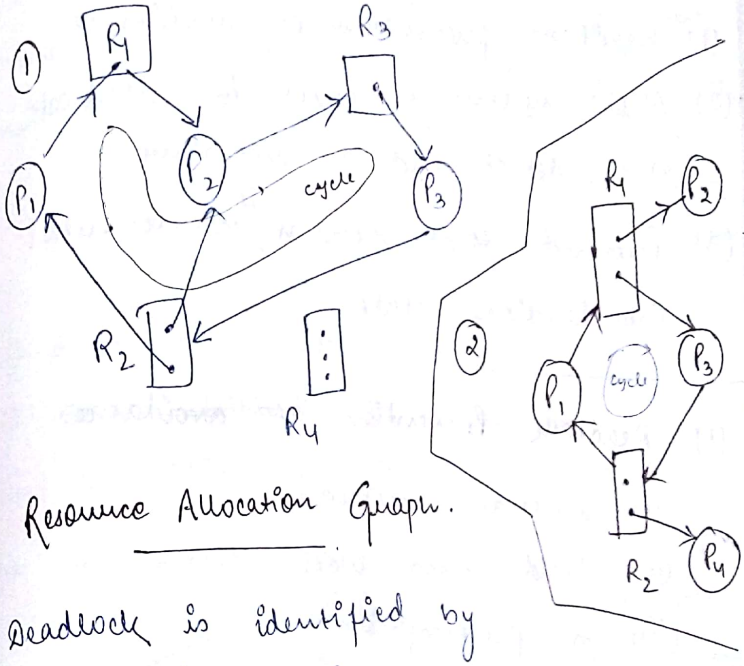
(2) Hold-and-wait (Holding one resource and waiting for another resource).

(3) No preemption

(4) Circular wait. (mean in $\{P_1, P_2, P_3 \ldots P_n\}$ P_1 is waiting for resource held by P_2, P_2 is waiting for resource held by P_3 ... P_n is waiting for P_1 resource)

If all these four conditions satisfied simultaneously, then there is deadlock.

Deadlock identification using Graphs.



Resource Allocation Graph.

Deadlock is identified by checking if there is cycle or not.

No cycle	→ no deadlock
cycle	→ deadlock may be (not sure)

Here, in given eg., both have cycles but only 1st one has deadlock, 2nd one does not have deadlock.

Solutions to tackle the deadlock.

(DEADLOCK HANDLING)

(1) Deadlock prevention or avoidence.

(2) Allow system to enter into deadlock state, detect and do recovery.

(3) Pretend that system is never ~~enter~~ into deadlock state.

---

(1) Deadlock Prevention ~~or avoidance~~.

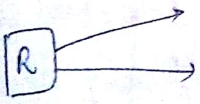
(i) mutual exclusion

(ii) hold - and - wait

(iii) no preemption

(iv) circular wait.

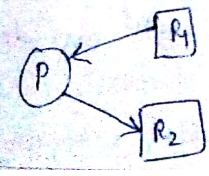
- mutual exclusion (ME)



Shareable resources (Possibility of violation of ME)

Non - shareable resources (no possibility)

- hold - and - wait



This condition can be violated if we give the process all the required resources from the very beginning because then it will not go into waiting state as it has all resources that it requires.

This condition can also be violated when before making any resource request, the other process releases all the resources. In this case, there will not be any holding.

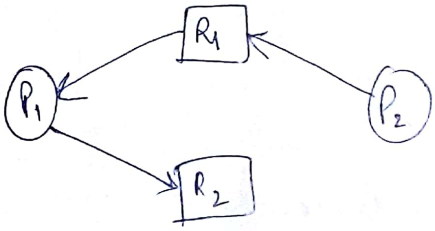
eg. DVD drive → file on disk → Print content on printer.

i) reserving resources unnecessarily

ii) Over utilisation of resources

iii) starvation problem.

- no preemption.



If P_1 releases R_1 voluntarily then P_2 can utilise R_1.

- circular wait

to avoid circular wait, a protocol called "Total ordering of Resources" is follow

Total Ordering of Resources.

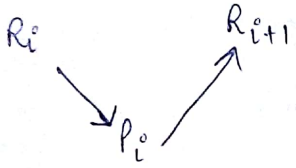
$F(R) \longrightarrow N$

$F(\text{DVD Drive}) = 1$

$F(\text{Disk}) = 4$

$F(\text{Printer}) = 10$

corresponding to each resource, an integer value is alloted. Resources are requested on the basis of increasing order of the alloted numbers.

eg. 

$F(R_i) < F(R_{i+1})$

P_i is holding R_i and requesting for R_{i+1}.

Deadlock Avoidance.

Here we use requirement of all the processes and design an algorithm assuming that the system never goes into deadlock state.

corresponding to all processes, we consider the maximum demand.

eg. 1)

Process	Maximum Demand	Current Need
P_0	10	5
P_1	4	2
P_2	9	2
		9

→ Safe state and Safe sequence.

If initially we assume that system not in deadlock state and after executing the algorithm, if the state remains the same then we calls it safe state.

If after execution of the ~~algorithm~~ process P_1, the system not in deadlock state then we say that process is a part of safe sequence $\langle P_1, P_2, \dots, P_n \rangle$.

If we are able to find a safe sequence then our system never goes into deadlock state by executing process in the sequence

---

Assume in the given eg. 1) that $R = 12$ instances.

Safe sequence :—

$\langle P_1, P_0, P_2 \rangle$

Here, a process gets available resource as per requirement, then uses it and releases all its resources.

After execution of all the processes, 12 instances of resource is remaining.

Available Resource.

3
− 2
#1
+ 4
5
− 5
0
+ 10
10
− 7
3
+ 9
12

Generalised algorithm (n Process, n Resources).

(V. imp.)

(BANKER'S ALGORITHM).

Steps –

1. Let WORK and FINISH be vectors of length m and n, initialized with WORK = AVAILABLE and ~~FINISH[i]~~ Finish[i] = false for i = 0, 1, . . . , n-1

2. Find an index i such that finish[i] == false

   Needi ≤ WORK

   if no such i exists, go to step 4

3. WORK = WORK + Allocation;

   Finish[i] = TRUE;

   Go to step 2.

4. If finish[i] == TRUE for all i, then system is in safe state.

Need[i][j] = MAX[i][j] - ALLOCATION[i][j]

a ≤ y if a[i] ≤ y[i] for all i.

eg.

Process	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			

5 processes, 3 resources and each resource has certain instances as shown in table. We need to find the safe sequence.

Process	Need (MAX[i][j] - Allocation[i][j]) 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

Work = Available

A	B	C
3	3	2
+ 2	0	0
5	3	2
+ 2	1	1
7	4	3
+ 0	0	2
7	4	5
+ 0	1	0
7	5	5

Now, P_1 needs less than available so include it in sequence.

$\langle P_1, P_3, P_4, P_0, P_2 \rangle$

7	5	5
+ 3	0	2
10	5	7

Date : 24th Oct, 2018.

1) Request of P_1 → Request (1, 0, 2) (Question)

(i) Request$_i$ ≤ Need$_i$
(ii) Request$_i$ ≤ Available
→ If these two conditions are satisfied then, system in safe state.

(iii) Available = Available − Request$_i$
Allocation$_i$ = Allocation$_i$ + Request$_i$
Need$_i$ = Need$_i$ − Request$_i$

Available = 3 3 2 − 1 0 2 = 2 3 0

Allocation: 2 0 0 + 1 0 2 = 3 0 2

Need : 1 2 2 − 1 0 2 = 0 2 0

Process	Allocation A	B	C	MAX A	B	C	Need A	B	C
P_0	0	1	0	7	5	3	7	4	3
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

Available : 2 3 0

2 3 0

3 0 2

5 3 2

2 1 1

7 4 3

0 0 2

7 4 5

0 1 0

7 5 5

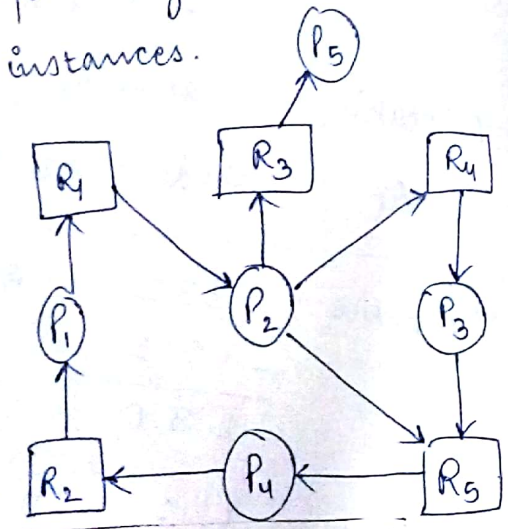
3 0 2

10 5 7

$\langle P_1, P_3, P_4, P_0, P_2 \rangle$

Deadlock Detection.

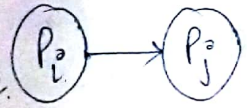
wait for Graph → applicable for single resource instances.



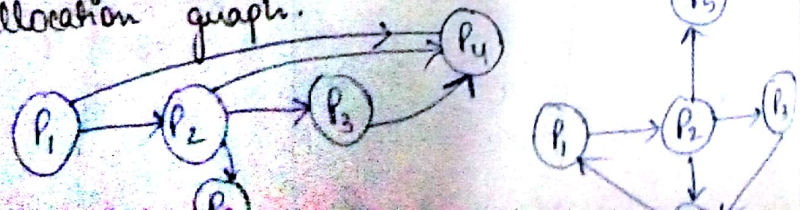
Resource allocation graph.

wait for Graph :→

symbol for equivalence



- wait for graph for above resource allocation graph.



If cycle exists in the wait-for graph, then the system is in deadlock state.

Deadlock detection in case of more than one instances.

(i) Let work and finish be the vector of length m and n respectively.

Initialized as work = Available for i = 0, 1, ..., n

if Allocation$_i$ ≠ 0 then FINISH[i] = FALSE

otherwise FINISH[i] = TRUE

~~(a) FINISH[i] = FALSE~~

(ii) Find an index i s.t.

(a) FINISH[i] = FALSE

(b) Request$_i$ ≤ WORK

if no such i exists then go to step (iv)

(iii) WORK = WORK + Allocation$_i$

FINISH[i] = TRUE

go to step (ii)

(iv) if FINISH[i] = = FALSE for some i then the system is in deadlock.

Example. $< P_0, P_2, P_1, P_3, P_4 >$

Process	Allocation A B C	Request A B C	Available A B C
P_0 ✓	0 1 0	0 0 0	0 0 0 → 0 1 0
P_1 ✓	2 0 0	2 0 2	0 1 0 → 3 0 3
P_2 ✓	3 0 3	0 0 0	3 0 3 → 3 0 3
P_3 ✓	2 1 1	1 0 0	3 0 3 → 2 0 0
P_4 ✓	0 0 2	0 0 2	5 0 3 → 2 1 1

7 1 3
0 0 2
7 1 5

- Banker's algo works on the hypothesis that in form of MAX field while here actual measures and conditions are checked after system is entered in deadlock. Here, in this algo, we have request field. To check whether request is granted or not, safe sequence is checked.

Deadlock Recovery.

(i) Process abort (breaks cyclic break condition)

(ii) Resource preemption

There are two ways of process aborting
— (i) about all the processes at the same time. By this, we reach to initial
(ii) about processes one by one.

Both the ways have some disadvantages.

For (i), if we about all the processes, then computation overhead will be very much because some processes might be in their last stage of execution. After aborting, they'll have to be restarted from the beginning.

For (ii), there may be the possibility that even after aborting processes one by one, the system may still be in deadlock state. Identifying the process whose abo
tion will remove deadlock is difficult as (it is still a research problem.)

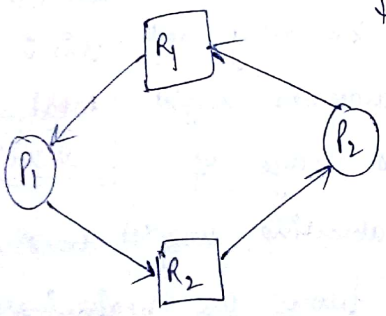
stage where all the resources were free for use. This is calle total aborting.

• Resource Preemption.

In this, we break / abort / release the resource. Here also the challenge is to find the victim resource whose release will remove deadlock.

→ Rote Backing.

If we preempt a particular resource R_1, then P_1 (which needs resource R_1) will again claim for R_1 after some time.



This time is called roll backing.

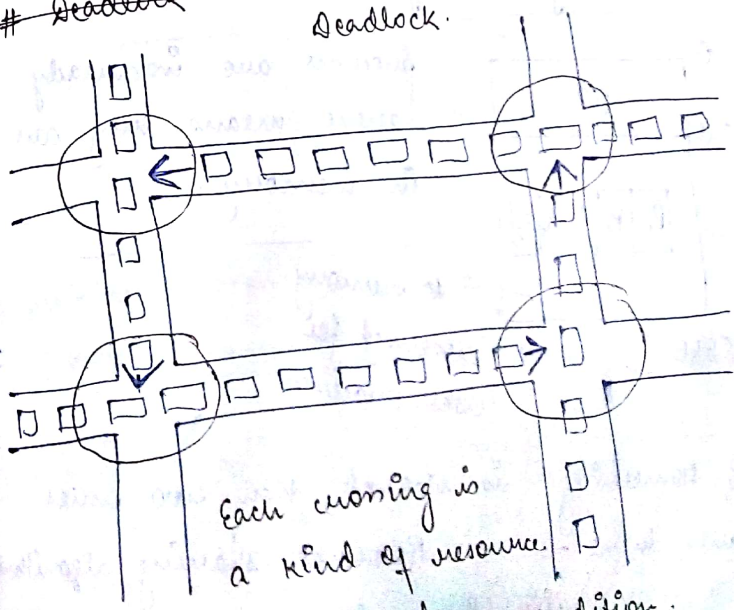
One solution of roll backing is that we roll back processes at the very beginning of their execution so that they can restart.

Another solution is to roll back ~~at~~ after the last safe state. Both of these are difficult to achieve. This is similar to the ATM situation where in case of error in transaction either we can restart the transaction by cancelling the previous one or we can have a window displaying the last safe step (that was properly executed) eg. identification of Pin no. & account).

• Starvation Problem.

Preemption selection algorithm may select a particular resource again and again so that processes waiting for that resource stave forever.

~~# Deadlock~~ # Road crossing example of Deadlock.



Now, check the same condition.

Mutual exclusion ✓ satisfied

Hold and wait ✓ "

No preemption ✓ "

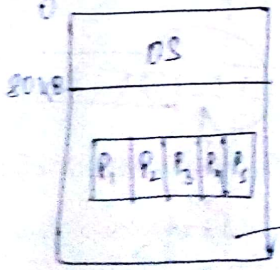
Circular wait ✓ "

Hence, the problem is in deadlock state.

- How to solve this deadlock problem? Give solution one for each of the four conditions.

Date : 31/10/18.

Memory management.



processes are in ready queue means they are in memory.

→ memory reserved for user processes

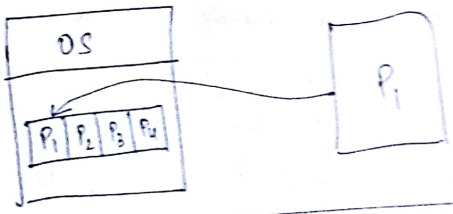
If something is shared, then two issues arise — (i) Resource sharing algorithm (ii) Protection mechanism.

Protection mechanism is for protecting one process's area from other processes' interference.

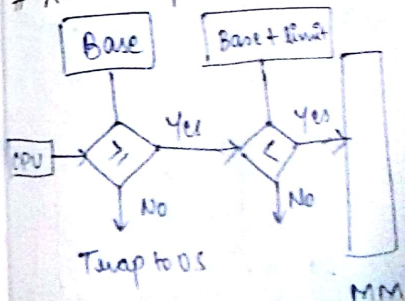
These two are the main concerns of the memory management.

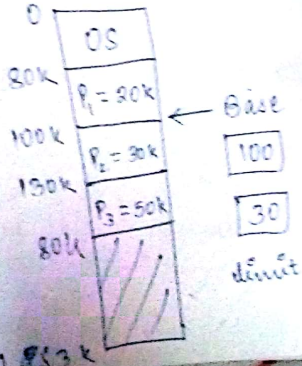
Resource sharing algo. → Paging, Segmentation, Paging with segmentation.

These three require some hardware support that does the mapping of processes into the memory.



Achieving protection





'Base' and 'limit' are the two registers for providing protection.

CPU (processor) generates some address (say 122) then it is checked and after meeting the conditions, it is mapped to the main memory.

Address Binding.

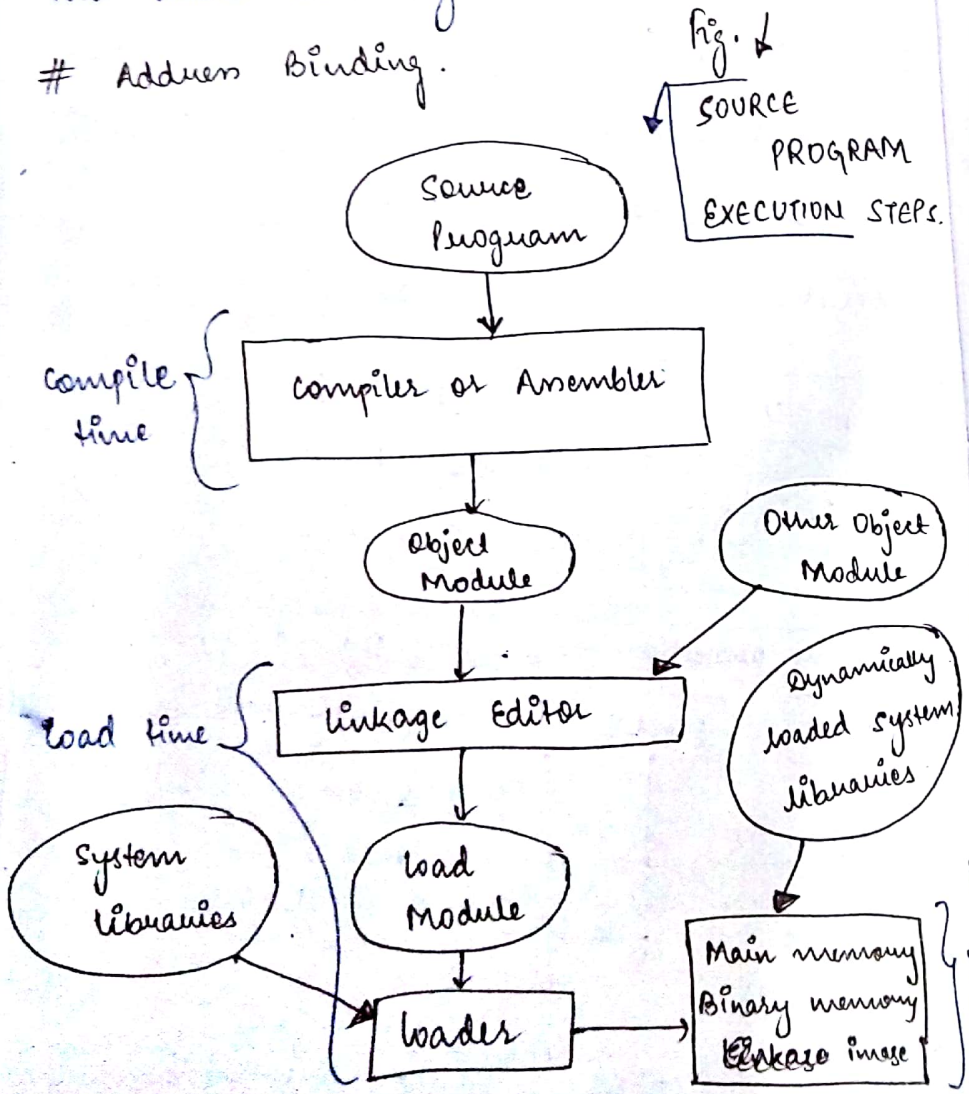
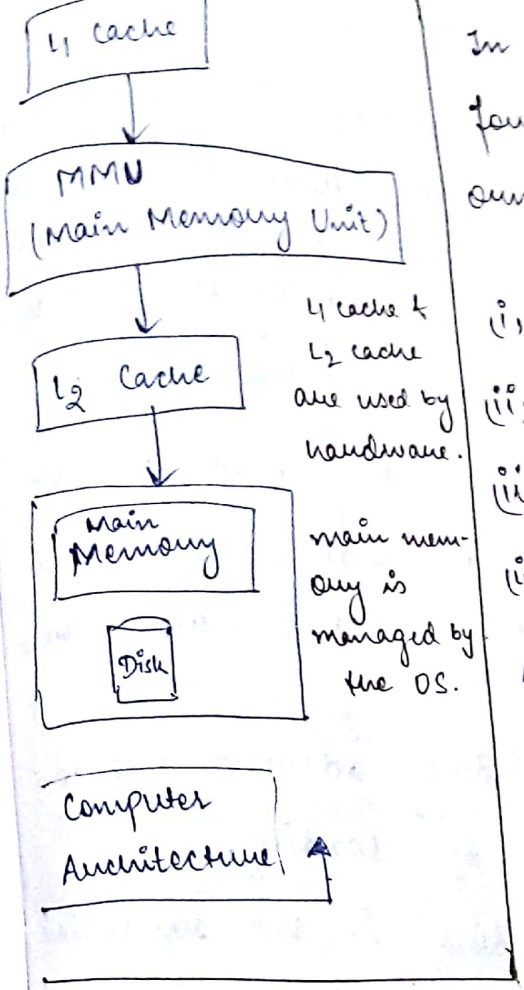


fig. 1

SOURCE PROGRAM EXECUTION STEPS.



In general, there are four processes performed by a compiler :—

(i) preprocessing
(ii) compilation
(iii) assembly
(iv) linkage.

After the compilation, we get a single object file.

Inside the main-memory, our program (the loaded one is present in the form of bytes. Process image is the ~~set~~ one actually present in memory representing the proc

Binding types —

1) Compile-time binding 2) Load-time b
3) Execution-time binding.

Address types —

1) Symbolic addresses (symbols / variables used)
2) Relocatable addresses (some other address can be relocated using the base address)
3) Absolute addresses. (actual address reflected in the main memory)

- If compile-time binding used, the location of program in main memory is known in advance. No changes are allowed after compilation.
- If load-time binding used, addresses are computed at the time of loading.
- Execution-time binding is the run-time binding. Addresses are ~~compile~~ computed at run-time as per requirement. It is better than the other two.

Drawbacks.

1) compile-time ~~needs~~ binding needs to fix the address of a process in memory even though it may not be required later on.

2) To shift from one place to another, mapping should be known.

Dynamic loading and Dynamic linking

If something is desired at the execution time, then only that process is loaded.

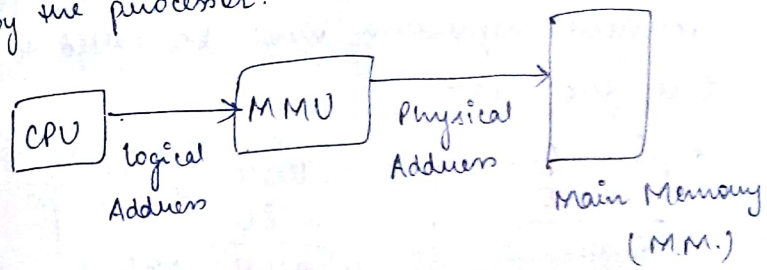
Suppose if a system call is there in if block then that will be linked at run time.

linking at the time of execution. eg. when we have conditional code like if(- -) else ...

Logical Address and Physical Address.

Address generated by the processor.

Actual address reflected inside the M.M.



logical Address Space and Physical Address Space:

LAS → Set of all logical addresses belonging to a particular process.

Fetch → Decode → Execute → Write Back.

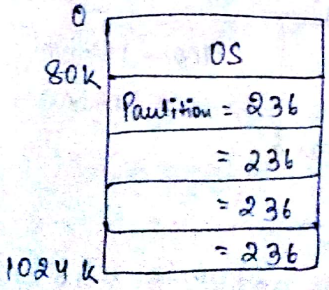
Mapping of logical Address to Physical Address.

I. CONTIGUOUS MEMORY ALLOCATION.

i) divide memory in fixed no. of partition
ii) divide memory in variable no. of partitions.

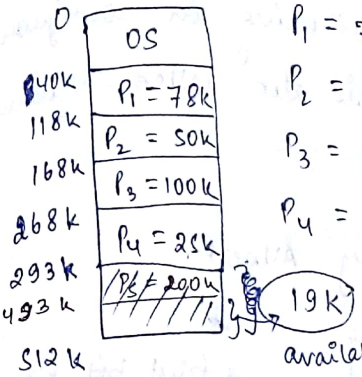
→ fixed no. of partition.

i) All components of a process are allocated memory in contiguous manner. If contigous memory not present then the process has to wait, or some replacement algorithm must be called to free the space.



$$
\begin{array}{r}
1024 \\
-\ 80 \\
\hline
4\overline{)944}\,(236 \\
8 \\
\hline
14 \\
12 \\
\hline
24 \\
24 \\
\hline
x.
\end{array}
$$

ii) Variable no. of partitions.

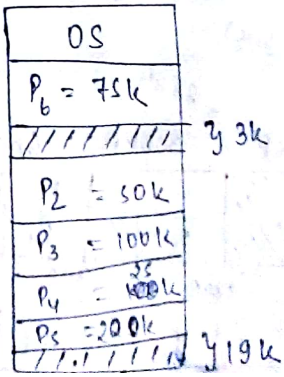


$P_1 = 78K$,
$P_2 = 50K$
$P_3 = 100K$
$P_4 = 25K$
$P_5 = 200K$
$P_6 = 75K$

Now, 19K is free space available and if P_6 makes request then it is not allowed to map as it requires 75K. But, suppose if P_1 (= 78K) completes its execution then P_6 will be allocated memory because then the P_1's 78K will get free.

unused space inside a partition.

Fragmentation → internal fragmentation
→ external "



This 3K space can't be utilised because processes normally required larger memory.

Date : 02/11/18

- Variable length partitioning in contiguous memory allocation is also called dynamic memory allocation.

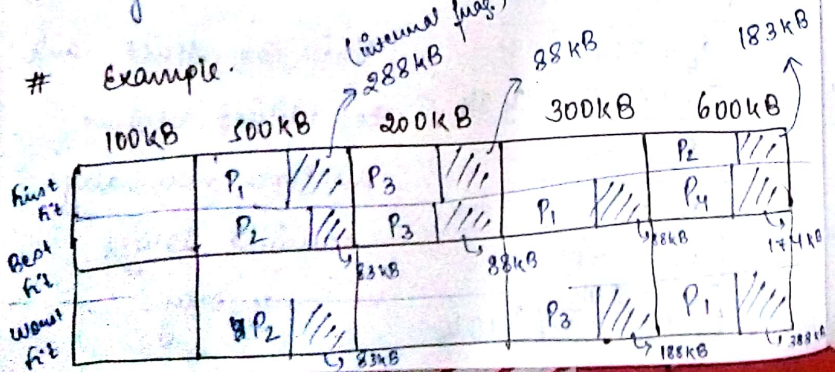
Contiguous Memory Allocation

→ fixed partitioning

→ variable partitioning → first fit
→ Best fit
→ worst fit

- first fit (allocate the 1st hole, i.e., big enough to accomodate a process)
- Best fit (allocate the smallest hole that is big enough to accomodate a process)
- Worst fit (allocate the largest hole that is big enough to accomodate a process)

Example.



we have : $P_1 = 212\,kB$, $P_2 = 417\,kB$,
$P_3 = 112\,kB$, $P_4 = 426\,kB$.

1) first fit : internal frag. = 288+88+183
external frag. = 0

P_4 is not being granted in the first fit and worst fit because no memory remaining to accomodate it.

Hence, best fit algorithm is the best for contiguous memory allocation.

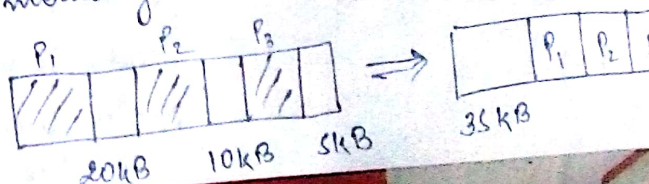
Solution of internal and external fragmentations.

1) Solution of External fragmentation

— COMPACTION TECHNIQUE.

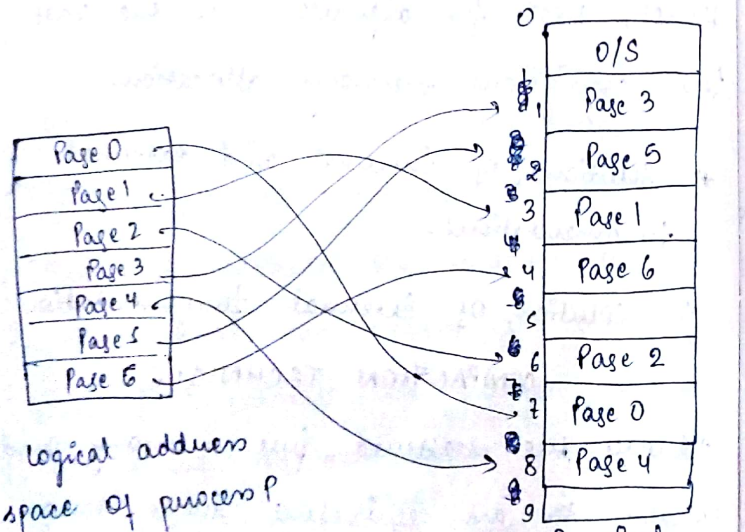
Compact the scattered free memory and make an equivalent single large block of free memory in another space of memory.

eg.



Compaction technique doesn't work in case of compile-time binding because in case of compile-time, addresses are allocated at compile-time and can't be moved further. so, compaction will not be possible in this case.

- Another solution —— PAGING.



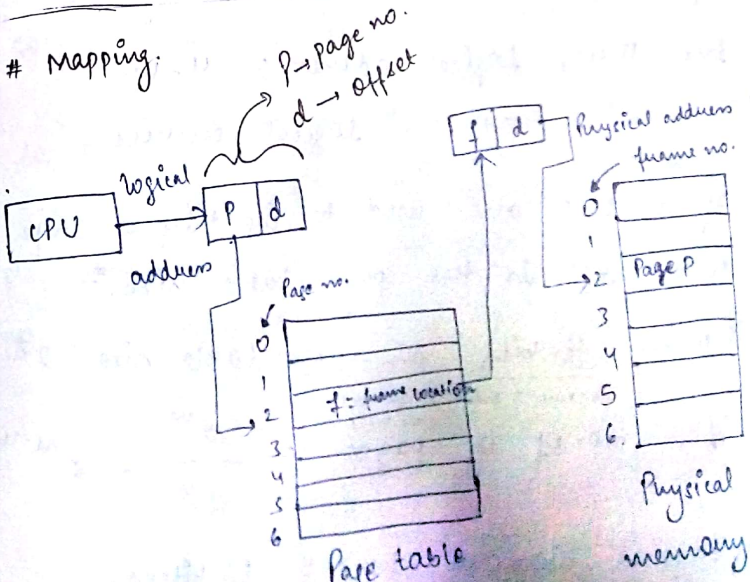
logical address space of process P.
(In this, size of each segment is equal to the size of a page)

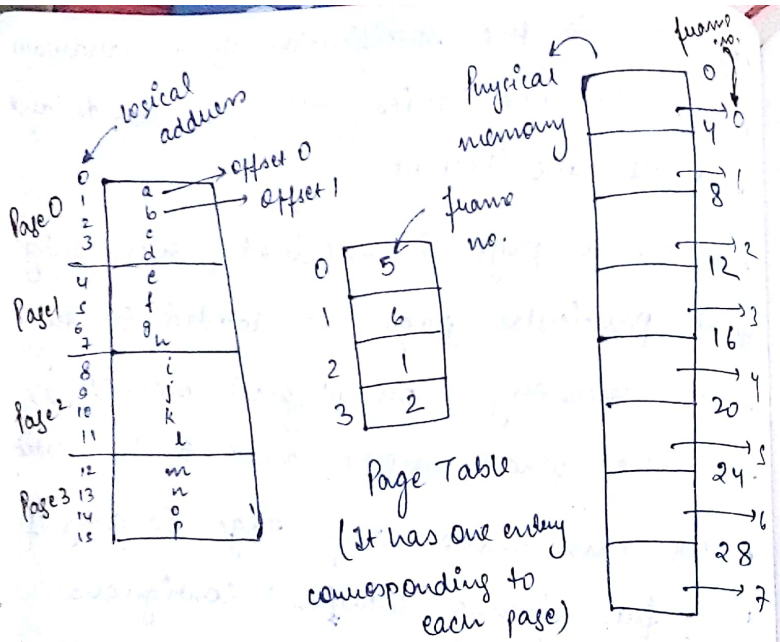
Physical memory.
(It is partitioned into frames)
(size of frame = size of page)

Page size is the specification of a hardware i.e., page size varies with the hardware and its architecture.

- when a page is required, then only that particular page is loaded to the main memory (or physical memory), not the whole process need to be loaded.
- we can move any page to any of the free frame location. Contiguous allocation is not necessary.

Mapping.





If logical address is of m-bit then 2^m logical entries are there.

Here, logical address = 4-bit

$\Rightarrow$ 2^4 logical entries.

If n-bits are used to identify a page then 2^n is the page table size.

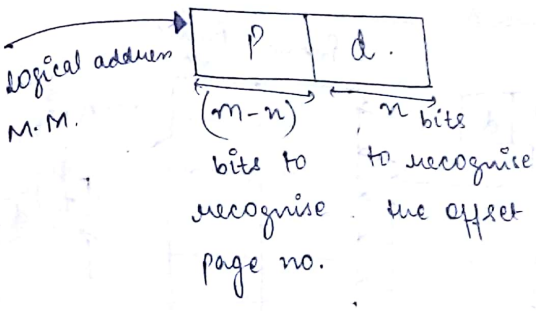
Here, 2-bits, so, page table size = 2^2.

(because 4 entries in one page are there)

So, no. of pages $= \frac{2^m}{2^n} = 2^{m-n}$

Physical memory here = 32 bytes.

$\frac{16}{4}$ = 4 pages in this case.



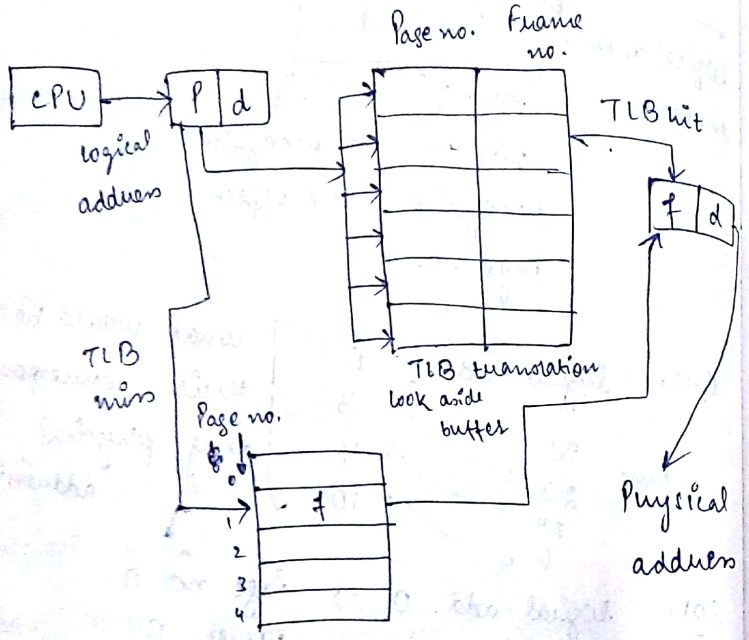
Que. logical add. = 0, 3, 4, 10 — what would be their corresponding physical address?

Ans:
20
23
24
6

Sol. logical add. 0 $\Rightarrow$ Page no. 0
Offset 0

So, checks 0th no. in page table where frame no. is 5. So, goes to 5th frame in the physical memory and allocates the corresponding offset.

Paging solves external fragmentation but doesn't solve internal fragmentation.



If page no. is present in look aside buffer then it is called TLB ~~miss~~ hit otherwise called TLB miss.

Hit Ratio (%)

Main memory reference = 100 nsec
TLB reference = 20 nsec
Hit Ratio = 80 %
Effective access time = ?

Sol. 80% of the time, the page is available 20 ns for searching & 100 ns to place it in main memory.

$$\underbrace{EAT}_{\text{Effective Access Time}} = \underbrace{0.80 \times 120}_{\text{for hit}} + \underbrace{0.20 \times 220}_{\text{for miss}}$$

$$= 152 \times 8 + 252 \times 2$$

$$= 956 + 454 = 1410 \text{ nsec}$$

If hit ratio is 98 %, then find EAT?

Sol. $EAT = 0.98 \times 120 + 0.02 \times 220$

$= 98 \times 1.2 + 2 \times 2.2$

$= 117.6 + 4.4$

$= 122.0$ ns

If hit ratio is 100 %, then EAT?

Sol. EAT = 120 ns.

So, increasing the hit ratio decreases the EAT.

Here, 20 ns is the overhead time for search using paging because paging is