DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

Course: Operating System

Code: 15CS64

IAT2- Solution

1. Consider the following set of processes given in table

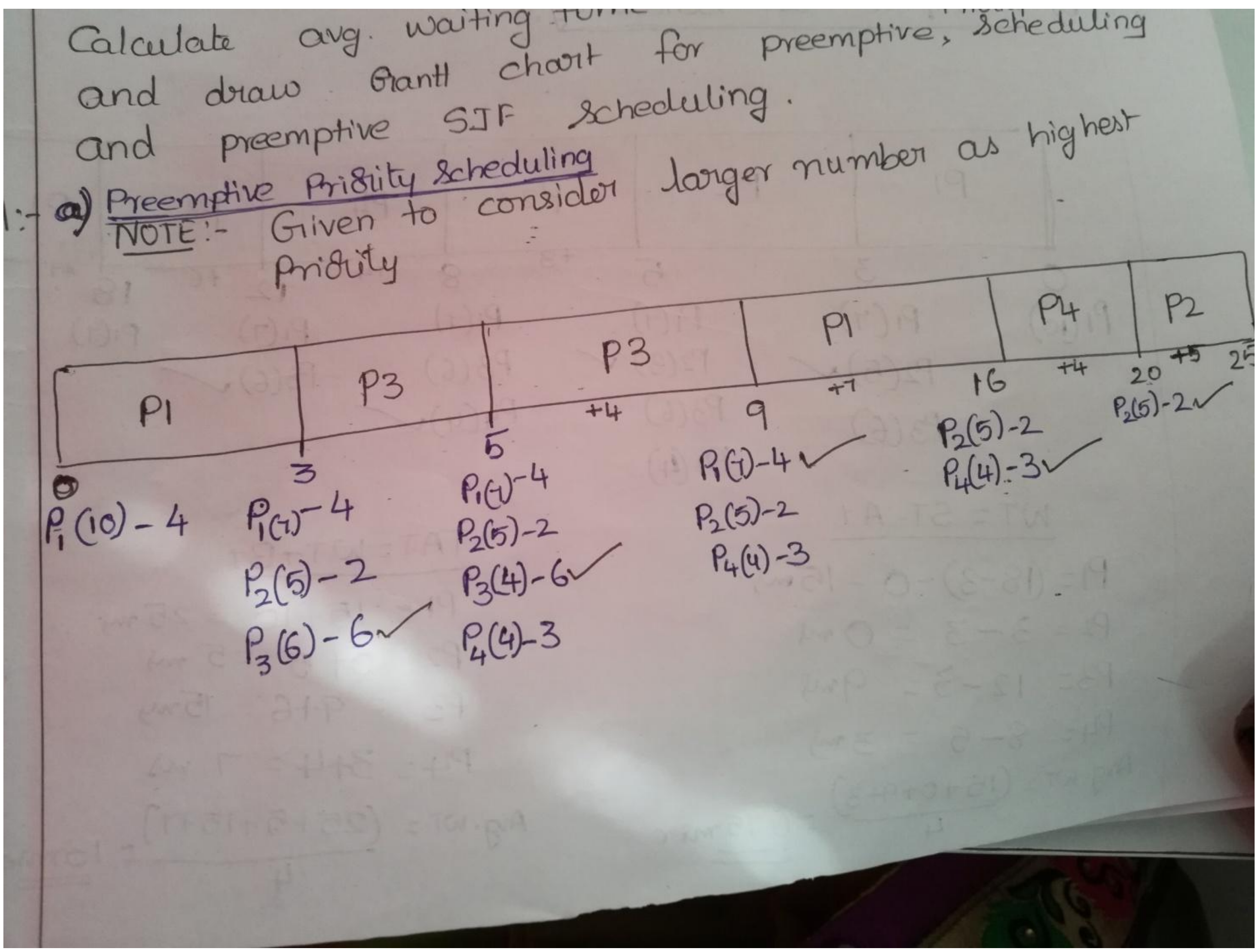
Processes	Arrival Time (m sec)	Burst Time (m sec)	Priority
P1	0	10	4
P2	3	5	2
P3	3	6	6
P4	5	4	3

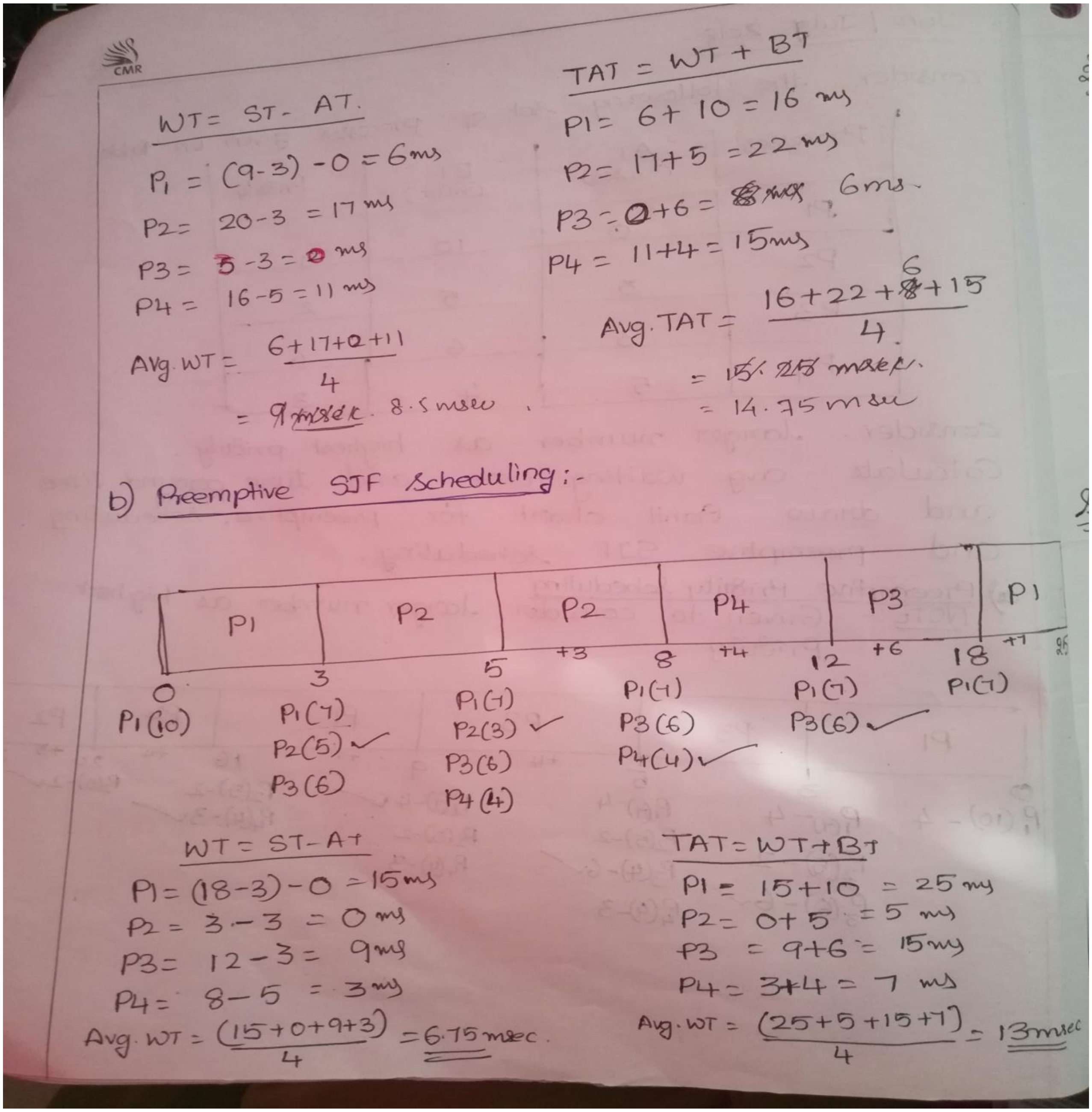
Considering **larger number as highest Priority**, for **Preemptive Priority scheduling & preemptive SJF scheduling**

(i) calculate average Waiting time and average Turnaround time

(ii) Draw a Gantt Chart respectively

Solution:





WT= ST- AT.

P1 = (9-3) -0 = 6ms

P2= 20-3 = 17 ms

P3= 5 -3 = [illegible] ms

P4 = 16-5 = 11 ms

Avg. WT = (6+17+0+11)/4

= ~~9 msec~~. 8.5 msec

TAT = WT + BT

P1= 6+ 10 = 16 ms

P2= 17+5 = 22 ms

P3 = 0+6 = ~~8 ms~~ 6ms.

P4 = 11+4 = 15ms

Avg. TAT = (16+22+6+15)/4

= ~~15.25 msec~~

= 14.75 msec

b) Preemptive SJF Scheduling:-

P1	P2	P2	P4	P3	P1
0	3	5 (+3)	8 (+4)	12 (+6)	18 (+7) 25

0: P1(10)

3: P1(7), P2(5) ✓, P3(6)

5: P1(7), P2(3) ✓, P3(6), P4(4)

8: P1(7), P3(6), P4(4) ✓

12: P1(7), P3(6) ✓

18: P1(7)

WT= ST-AT

P1= (18-3)-0 = 15ms

P2 = 3-3 = 0 ms

P3= 12-3= 9ms

P4= 8-5 = 3ms

Avg. WT = (15+0+9+3)/4 = 6.75 msec.

TAT=WT+BT

P1 = 15+10 = 25 ms

P2= 0+5 = 5 ms

P3 = 9+6 = 15ms

P4= 3+4 = 7 ms

Avg. WT = (25+5+15+7)/4 = 13msec

2. What is a **critical section problem**? What **requirements** should a solution to critical section problem satisfy? State **Peterson's solution** and indicate how it satisfies the above requirements.

Solution:

- **Critical-section** is a segment-of-code in which a process may be
  → changing common variables
  → updating a table or
  → writing a file.
- Each process has a critical-section in which the shared-data is accessed.
- General structure of a typical process has following (Figure 2.12):
  1) Entry-section
  - Requests permission to enter the critical-section.

  2) Critical-section
  - Mutually exclusive in time i.e. no other process can execute in its critical-section.

  3) Exit-section
  - Follows the critical-section.

4) Remainder-section

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

- Problem statement:

  –Ensure that when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section.
- A solution to the problem must satisfy the following 3 requirements:

  1) Mutual Exclusion:
  - No more than one process can be in critical-section at a given time.

  2) Progress:
  - When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay..

  3) Bounded Waiting (No starvation):
  - There is an upper bound on the number of times a process enters the critical section, while another is waiting.
- Two approaches used to handle critical-sections:

  1) Preemptive Kernels
  - Allows a process to be preempted while it is running in kernel-mode.
  - More suitable for real-time proframming

  2) Non-preemptive Kernels
  - Does not allow a process running in kernel-mode to be preempted as it is free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

<u>Peterson's Solution</u>

- This is a classic **software-based solution** to the critical-section problem.
- This is limited to 2 processes.
- The 2 processes alternate execution between

  → critical-sections and

  → remainder-sections.
- The 2 processes (say i & j)share two globally defined variables:

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

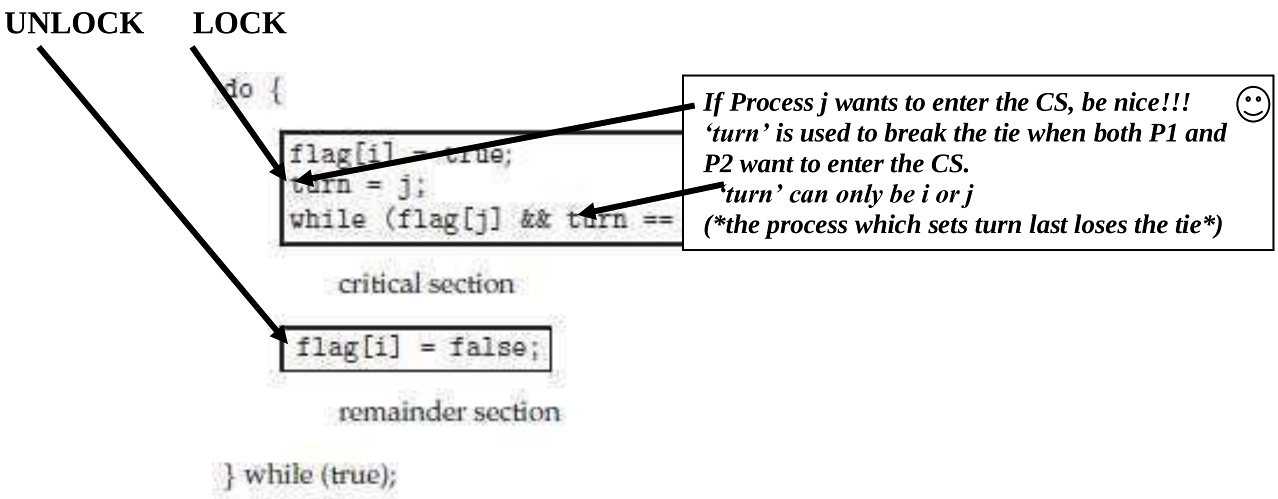
'turn' – indicates whose turn it is to enter its critical-section.

(i.e., if turn==i, then process Pi is allowed to execute in its critical-section).

'flag' – indicates if a process is ready to enter its critical-section.

(i.e. if flag[i]=true, then Pi is ready to enter its critical-section).

- The following code shows the structure of ***process P_i*** in Peterson's solution:



- To enter the critical-section,
  - → firstly, process P_i sets flag[i] to be true and
  - → then sets turn to the value j.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- The final value of turn determines which of the 2 processes is allowed to enter its critical-section first.
- To prove that this solution is correct, we show that:
  1) Mutual-exclusion is preserved:
     - Observation1: Pi enters the CS only if flag[j]== false or turn ==i.
     - Observation2: If both processes can be executing in their CSs at the same time, then flag[i]==flag[j]==true.

     These two observations imply that P_i and P_j could not have successfully executed their *while* statements at about the same time, since the value of turn can be either i or j but cannot be both. Hence, the process which sets 'turn' first will execute and Mutual Exclusion is preserved.
  2) The progress requirement & The bounded-waiting requirement is met:
     - The process which executes while statement first (say Pi), doesn't change the value of turn. So other process (Say Pj) will enter the CS (Progress) after at most one entry (Bounded Waiting)

3. Define **Semaphores**. Explain **dinning philosopher's problem** using Semaphores.

Solution:

- A semaphore is a synchronization-tool.
- It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.
- A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations:
  1) wait() and
  2) signal().

- wait() is termed P ("to test or decrement") signal() is termed V ("to increment").

Definition of wait():

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

Definition of signal():

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

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

The Dining-Philosophers Problem

- Problem statement:
  - There are 5 philosophers with 5 chopsticks (semaphores).
  - A philosopher is either eating (with two chopsticks) or thinking.
  - The philosophers share a circular table (Figure 2.21).
  - The table has
    → a bowl of rice in the center and
    → 5 single chopsticks.
  - From time to time, a philosopher gets hungry and tries to pick up the 2 chopsticks that are closest to her.
  - A philosopher may pick up only one chopstick at a time.
  - Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
  - When 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.
- Problem objective:
  To allocate several resources among several processes in a deadlock-free & starvation-free manner.
- Solution:
  - Represent each chopstick with a semaphore (Figure 2.22).
  - A philosopher tries to grab a chopstick by executing a wait() on the semaphore.
  - The philosopher releases her chopsticks by executing the signal() on the semaphores.
  - This solution guarantees that no two neighbors are eating simultaneously.
  - ***Shared-data***
    semaphore chopstick [5];

1. Initialization

chopstick [5] = {1,1,1,1,1}.

Figure 2.21 Situation of dining philosophers

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

Figure 2.22 The structure of philosopher

- Disadvantage:
  1) Deadlock may occur if all 5 philosophers become hungry simultaneously and grab their left chopstick. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Three possible remedies to the deadlock problem:

1) Allow **at most 4** philosophers to be sitting simultaneously at the table.
2) Allow a philosopher to pick up her chopsticks **only if both chopsticks are available**.
3) Use an **asymmetric solution**; i.e. an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

4. Define **Deadlock**. List and explain the **necessary conditions** for a deadlock to occur and **methods for handling** them in detail.

Solution:

<u>Deadlocks</u>

- Deadlock is a situation where a set of processes are blocked because each process is
  → holding a resource and
  → waiting for another resource held by some other process.
- Real life example:
  When 2 trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other.
- Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s).

<u>Necessary Conditions</u>

- There are four conditions that are necessary to achieve deadlock:

i) Mutual Exclusion

- At least one resource must be held in a non-sharable mode.
- i.e., If one process holds a non-sharable resource and if any other process requests this resource, then the requesting-process must wait for the resource to be released.

1) Hold and Wait

- A process must be simultaneously
  → holding at least one resource and
  → waiting to acquire additional resources held by the other process.

2) No Preemption

- Resources cannot be preempted.
- A resource can be released voluntarily by the process holding it.

3) Circular Wait

- A set of processes { P0, P1, P2, . . ., PN } must exist
  P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2 ….and PN is waiting for a resource held by P0.

<u>Methods for Handling Deadlocks:</u>

- There are three ways of handling deadlocks:
  1) **Deadlock prevention or avoidance** – Use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlock state.
  2) **Deadlock detection and recovery** – allow the system to enter a deadlock state, detect it, and recover
  3) **Ignore the problem all together** – Ignore the problem altogether and pretend that deadlocks never occur in the system.

- The third solution is the one used by most operating systems, including UNIX and Windows.
- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future.
- Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system

will gradually slow down.

5. Consider the following snapshot of a system

Process	Allocation			Max.			Available		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	0	2	0	0	4	1	0	2
P1	1	0	0	2	0	1			
P2	1	3	5	1	3	7			
P3	6	3	2	8	4	2			
P4	1	4	3	1	5	7			

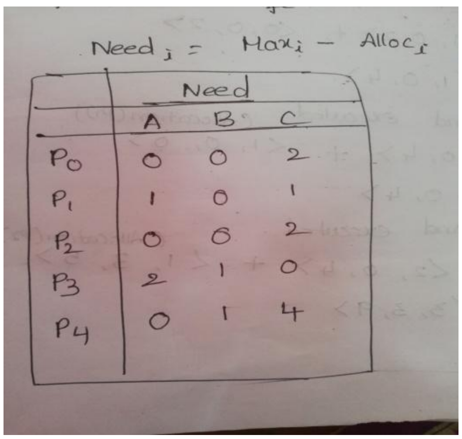
Using **Banker's Algorithm**, answer the following questions

(i) Find out Need Matrix

(ii) Is the system in a safe state?

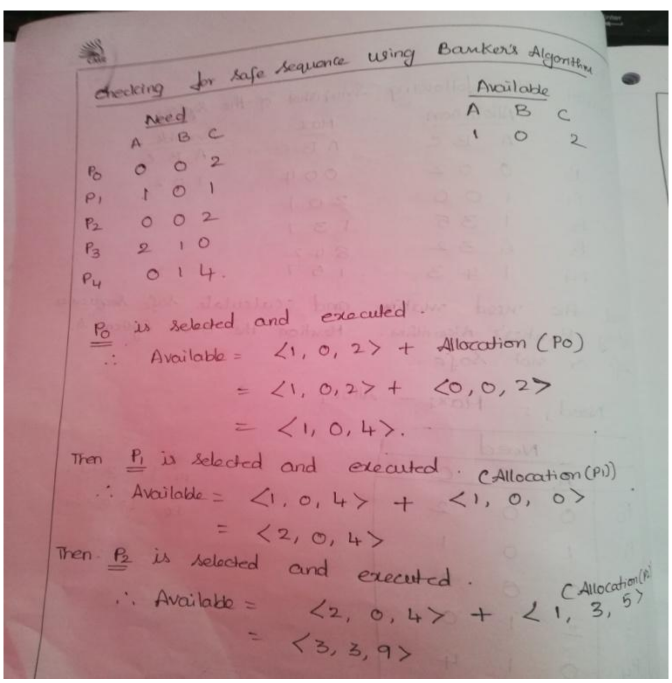
(iii) When a request from P1 arrives for (1, 0, 1), can the request be granted immediately?

Solution:



$Need_i = Max_i - Alloc_i$

	Need A	Need B	Need C
P0	0	0	2
P1	1	0	1
P2	0	0	2
P3	2	1	0
P4	0	1	4



Checking for safe sequence using Banker's Algorithm

	Need A	Need B	Need C
P0	0	0	2
P1	1	0	1
P2	0	0	2
P3	2	1	0
P4	0	1	4

Available: A B C = 1 0 2

P0 is selected and executed.

∴ Available = <1, 0, 2> + Allocation (P0)

= <1, 0, 2> + <0, 0, 2>

= <1, 0, 4>.

Then P1 is selected and executed. (Allocation (P1))

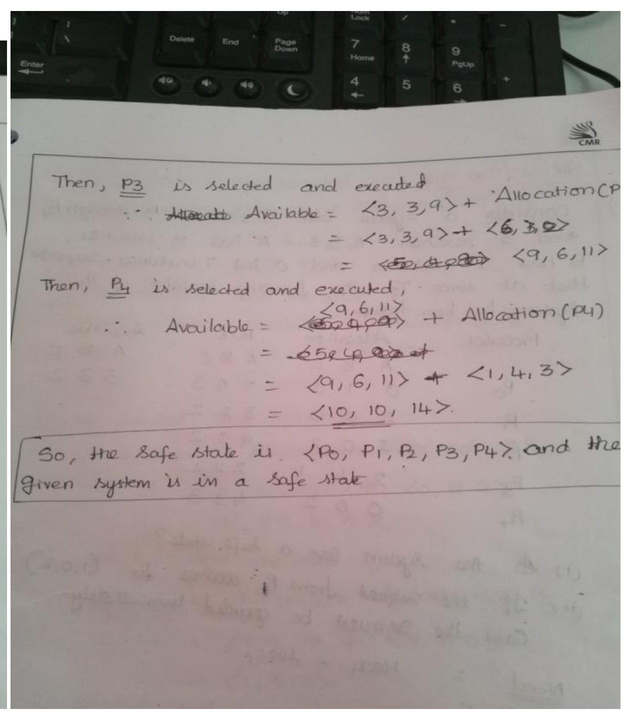
∴ Available = <1, 0, 4> + <1, 0, 0>

= <2, 0, 4>

Then P2 is selected and executed. (Allocation (P2))

∴ Available = <2, 0, 4> + <1, 3, 5>

= <3, 3, 9>



Then, P3 is selected and executed

∴ Available = <3, 3, 9> + Allocation (P3)

= <3, 3, 9> + <6, 3, 2>

= <9, 6, 11>

Then, P4 is selected and executed,

∴ Available = <9, 6, 11> + Allocation (P4)

= <9, 6, 11> + <1, 4, 3>

= <10, 10, 14>.

So, the safe state is <P0, P1, P2, P3, P4> and the given system is in a safe state.

6. Explain **Deadlock Detection** mechanisms for single & multiple instances with neat diagrams
Solution:

Deadlock Detection

- If a system does not use deadlock-prevention or deadlock-avoidance algorithm then a deadlock may occur.
- In this environment, the system must provide
  1) An algorithm to examine the system-state to determine whether a deadlock has occurred.
  2) An algorithm to recover from the deadlock.

Single Instance of Each Resource Type

- If all the resources have only a single instance, then deadlock detection-algorithm can be defined using a wait-for-graph.
- The wait-for-graph is applicable to only a single instance of a resource type.
- A wait-for-graph (WAG) is a variation of the resource-allocation-graph.
- The wait-for-graph can be obtained from the resource-allocation-graph by
  → removing the resource nodes and
  → collapsing the appropriate edges.
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists if and only if the corresponding graph contains two edges
  1) $P_i \rightarrow R_q$ and
  2) $R_q \rightarrow P_j$.
- For example:
  Consider resource-allocation-graph shown in Figure 3.6 Corresponding wait-for-graph is shown in Figure 3.7.

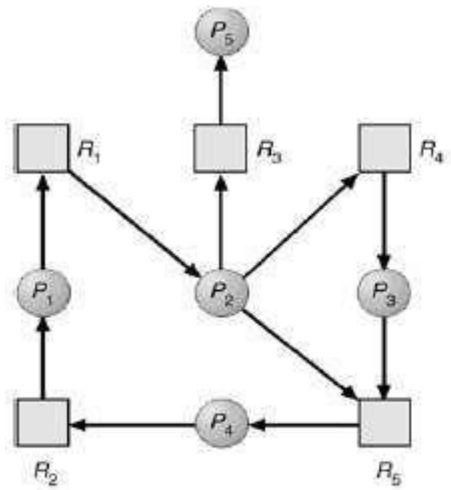


Figure 3.6 Resource-allocation-graph

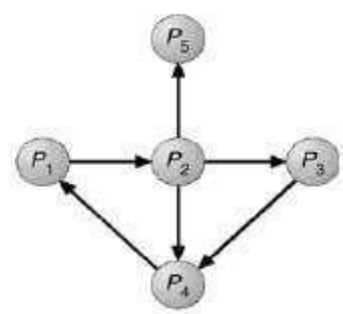


Figure 3.7 Corresponding wait-for-graph.

- A deadlock exists in the system if and only if the wait-for-graph contains a cycle.
- To detect deadlocks, the system needs to
  → maintain the wait-for-graph and
  → periodically execute an algorithm that searches for a cycle in the graph.

Several Instances of a Resource Type

- The wait-for-graph is applicable to only a single instance of a resource type.
- Problem: However, the wait-for-graph is not applicable to a multiple instance of a resource type.
- Solution: The following detection-algorithm can be used for a multiple instance of a resource type.
- Assumptions:

Let ‘n’ be the number of processes in the system Let ‘m’ be the number of resources types.

- Following data structures are used to implement this algorithm.

3) ***Available [m]***
   i. This vector indicates the no. of available resources of each type.
   ii. If Available[j]=k, then k instances of resource type Rj is available.

4) ***Allocation [n][m]***
   - This matrix indicates no. of resources currently allocated to each process.
   - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.

5) ***Request [n][m]***
   i. This matrix indicates the current request of each process.
   ii. If Request [i, j] = k, then process Pi is requesting k more instances of resource type Rj.

This algorithm requires an order mxn^2 operations to detect whether the system is in a deadlocked state

Step 1:
```
        Let Work and Finish be vectors of length m and n respectively.
                a) Initialize Work = Available
                b)      For i=0,1,2…......n
                        if Allocation(i) != 0
                        then
                                Finish[i] = false;
                        else
                                Finish[i] = true;
```
Step 2:
```
        Find an index(i) such that both
                a) Finish[i] = false
                b) Request(i) <= Work.
                If no such i exist, goto step 4.
```
Step 3:
```
                Set:
                    Work = Work + Allocation(i)
                    Finish[i] = true
        Go to step 2.
```
Step 4:
```
        If Finish[i] = false for some i where 0 ≤ i < n, then the system is in a deadlock state.
        Moreover, if Finish[i] == false, then process Pi is deadlocked.
```

7. With a neat diagram, explain how hardware supports **memory allocation and protection** of processes

Solution:

Contiguous Memory Allocation

- Memory is usually divided into 2 partitions:
  → One for the resident OS.
  → One for the user-processes.
- Each process is contained in a single contiguous section of memory.

Memory Mapping & Protection

- Memory-protection means
  → protecting OS from user-process and
  → protecting user-processes from one another.
- Memory-protection is done using
  → **Relocation-register**: contains the value of the smallest physical-address.

→ **Limit-register**: contains the range of logical-addresses.

• Each logical-address must be less than the limit-register.

• The MMU maps the logical-address dynamically by adding the value in the relocation-register. This mapped-address is sent to memory (Figure 3.13).

• When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit-registers with the correct values.

• Because every address generated by the CPU is checked against these registers, we can protect the OS from the running-process.

• The relocation-register scheme provides an effective way to allow the OS size to change dynamically.

• **Transient OS code**: Code that comes & goes as needed to save memory-space and overhead for unnecessary swapping.

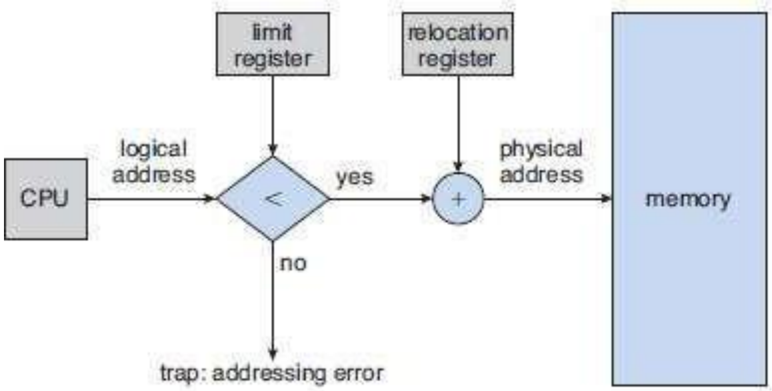


Figure 3.13 Hardware support for relocation and limit-registers

Memory Allocation

• Two types of memory partitioning are: 1) Fixed-sized partitioning and 2) Variable-sized partitioning

1) Fixed-sized Partitioning

- The memory is divided into fixed-sized partitions.
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is
  → selected from the input queue and
  → loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

2) Variable-sized Partitioning

- The OS keeps a table indicating
  → which parts of memory are available and
  → which parts are occupied.
- A **hole** is a block of available memory.
- Normally, memory contains a set of holes of various sizes.
- Initially, all memory is
  → available for user-processes and
  → considered one large hole.
- When a process arrives, the process is allocated memory from a large hole.
- If we find the hole, we
  → allocate only as much memory as is needed and
  → keep the remaining memory available to satisfy future requests.

• Three strategies used to select a free hole from the set of available holes.

1) First Fit

- Allocate the first hole that is big enough.
- Searching can start either
  → at the beginning of the set of holes or
  → at the location where the previous first-fit search ended.

2) Best Fit

- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

3) Worst Fit

- Allocate the largest hole.
- Again, we must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.

- First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.