1) Student Name : Venkatesh G D

2) USN : 2GI19CS175

3) BE/MBA : B.E

4) Semester : 4

5) Course Name : OS

6) Course Code : 18CS42

7) Name of Colg : KLS GIT

8) Date & Time : 19/07/21 , 10:30AM

9) Mob. No : 9972287030

10) Signature : Venkatesh

6) Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that main memory can be made available for other processes. It is used to improve main memory utilization. In secondary memory, the place where the swapped-out process is stored is called swap space.

A process can be swapped temporarily out of memory to a backing store & then brought back into memory for continued execution.

Roll out, roll in - Swapping variant used for priority based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded & executed. Major part of swap time is transfer time; total transfer time is directly proportional to amount of memory swapped.
System maintains a ready queue of ready to run processes which have memory images on disk.

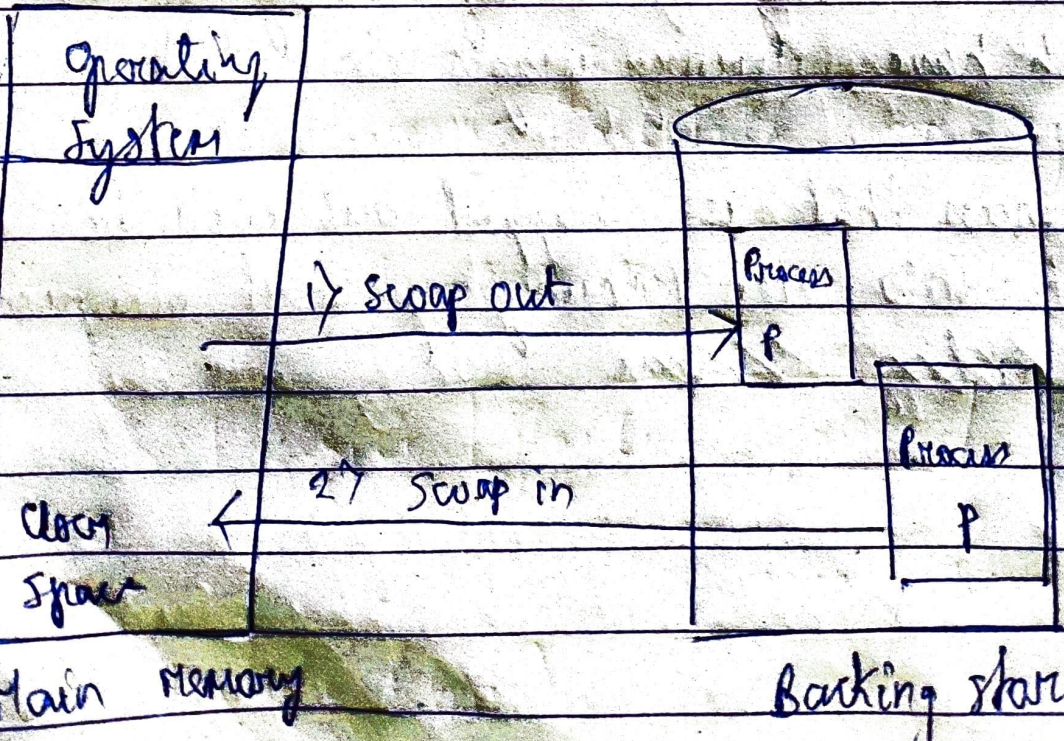
Normally, a process that is swapped out will be swapped back into same memory space it occupied previously. If binding is done

6.) If execution-time binding is being used, however then a process can be swapped into a different memory space, because the physical address are computed during execution time.

Swapping is constrained by other factors as well. If one want to swap a process, we must be sure that it is completely idle. Never swap a process with pending I/O.

The purpose of swapping in OS is to access the data present on the hard disk & bring it to RAM so that the application programs can use it. The thing to remember is that swapping is used only when data is not present in RAM.

Although the process of swapping affects the performance of system, it helps to run larger & more than one process. This is the reason why swapping is also referred to as memory compaction.



17 * Critical section problem: The critical section is a code segment where the shared variables can be accessed.

* Solution to critical-section problem must satisfy these requirements:

i) Mutual exclusion: If process P is executing in its critical section, then no other process can be executing in their critical section.

ii) Progress - If no process is executing in its critical section & there exists some process that wish to enter their critical section, then the section of process that will enter the critical section next cannot be postponed indefinitely.

iii) Bounded waiting - A bound must exist on the number of times that other processes are allowed to enter the critical sections after process has made a request to enter its critical section & before that request is granted.

3) Peterson's solution

→ It is a good algorithmic description of solving problem, it is a two process sol^n

→ Assume that the load & store machine language are atomic i.e cannot be interrupt

→ The two processes share two variables :

* int turn;
* boolean flag[2]

→ The variable turn indicates whose turn it is to enter the critical section.

→ The flag ~~memory~~ array is used to indicate if a process is ready to enter the critical section.

flag[i] = true → implies process P_i is ready.

→ Algorithm:

Do {

```
flag[i] = true;
turn = j;
while ( flag[j] && turn = = j);
```

Critical section

flag[i] = false;

Remainder section.

} while (true);

3.) The 3 critical section requirements are met:
* Mutual exclusion is preserved
* Progress requirement is satisfied.
* Bounded-waiting requirement is met.

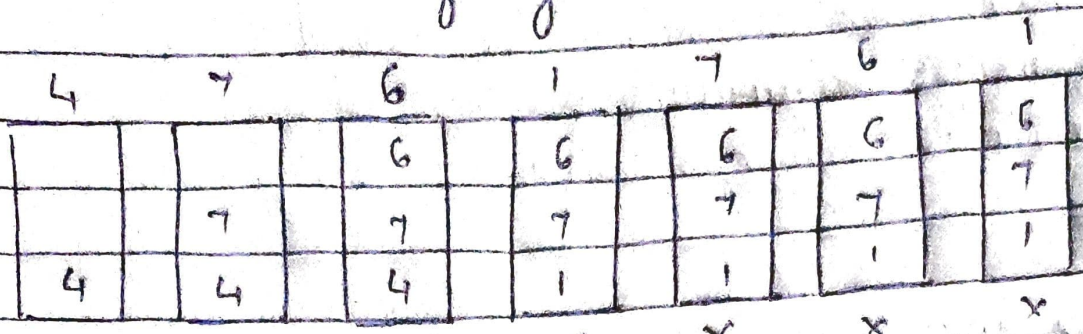
2.)* Semaphores are integer variables that are used to solve critical section problem by using two atomic operations, wait & signal that are used for process synchronization

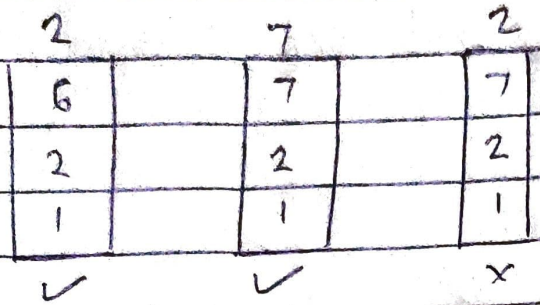
* Semaphores Implementation:

→ Must guarantee that no two processes can execute wait() & signal() on same semaphore at same time.
↔ Thus, the implementation becomes the critical section problem where wait & signal code are placed in critical section.
- Could now have busy waiting in critical section implementation.
  > But implementation code is short
  > Little busy waiting if critical section rarely occupied
→ Note that applications may spend lots of time in critical sections & therefore this not a good solution

FIFO

7.) Total number of references = 10





7 2 1 6 7 4

QUEUE

From here,

Total number of page faults occurred = 6

LRU:

f3			6	6	6	6	6	6	7	7
f2		7	7	7	7	7	7	2	2	2
f1	4	4	4	1	1	1	1	3	1	1
	M	M	M	M	Hit	Hit	Hit	M	M	Hit

4, 7, 6, 1, 7, 6, 1, 2, 7, 2

The number of hits are : 4

The number of faults are : 6

1.) Banker's Algorithm

Process	Allocation	Max	Available	Need
P_0	1 2 1	1 0 3	5 5 5	0 2 2
P_1	2 0 1	0 1 2	6 7 6	2 1 1
P_2	2 2 1	1 2 0	8 7 7	1 0 1
			10 9 8	

We know that,
Need = Max - Allocation
If need ≤ available
unavailable = available + allocation
$P_0 \Rightarrow 022 \leq 555$ ✓ P_0 will be executed
unavailable = 555 + 121
= 676

$P_1 \Rightarrow 211 \leq 676$ ✓ P_1 will be executed
unavailable = 676 + 201
= 877

$P_2 \Rightarrow 101 \leq 877$ ✓ P_2 will be executed
unavailable = 877 + 221
= 1098

∴ The system is in safe state & safe sequence is
$\langle P_0, P_1, P_2 \rangle$.