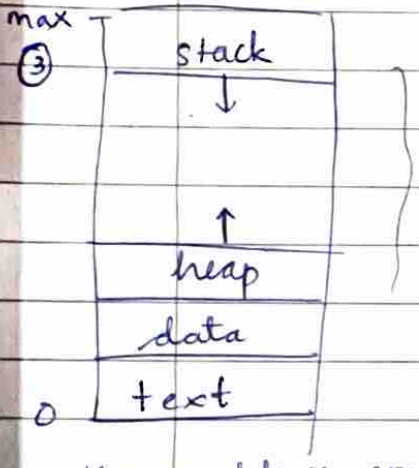
UNIT - 2 Process Management.

Date : _____

① Memory → Paging (demand paging)
→ Segmentation - during compile time, divided into ② segments, loaded in M-M segment wise.

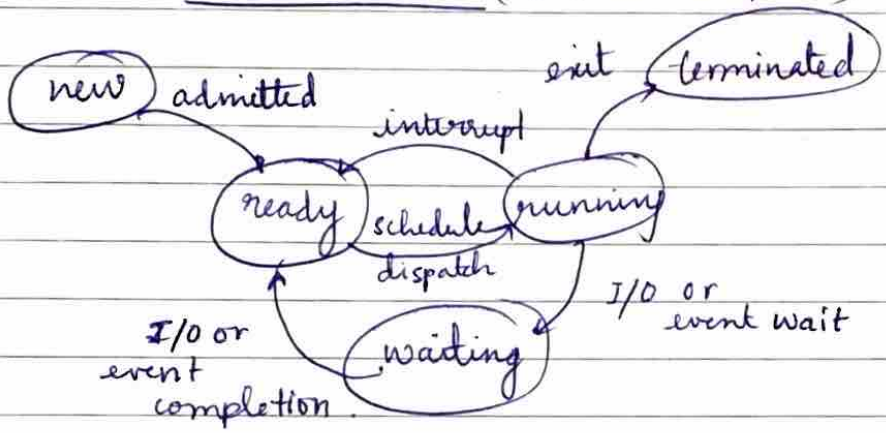
process is stored in M·M as segments:
- ↳ text seg. - instructions / program code / executable (not declare/ initialization)
- ↳ stack seg. - function call return address, local var, parameters
- ↳ data seg. - global variables < initialized / uninitialized
- ↳ heap seg. - dynamic memory allocation

③



process address space.

④ Process State (5 state diagram)



⑤
- new → process is newly loaded into M·M.
- ready → process that got all the resources required.
- running → process is being executed
- terminated → process completes execution.
- waiting → process is interrupted while running or event wait or requires non available resource.
  - ↓ moved to SWAP area (virtual/memory) secondary

⑥
- * dynamic binding → when memory is available / free, load and start execution
- * static binding → goes to fixed address space

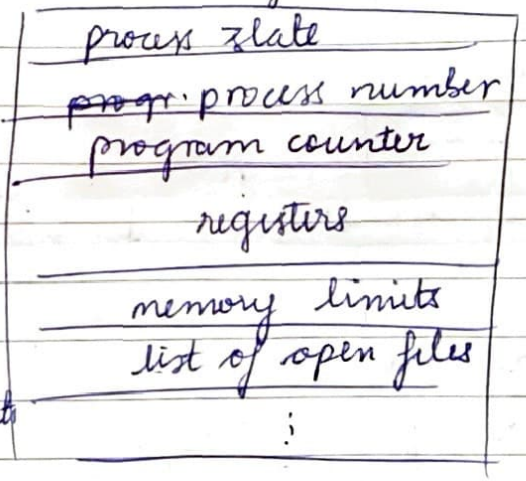
note: cache affinity ? ~~no prob~~ problem, solution, merits

* medium term scheduler used to select partially executed processes.

* Running → waiting: interrupts, unavailable resource, higher priority process, wait events

Process Control Block (PCB) → Windows

* complete information of all processes in the system.
* each process has one PCB.
* also Task Control Block
* stored in secondary memory.



state → any 5 state.
number → process ID
PC → address of next instruction
memory limits → memory requirements
eg: 1000 to 2000
open files → if stdio.h, packages.
scheduling info, accounting info (clock time), I/O status.

Q) When p1 is executing, what will be OS be doing? as in which state? (system has one CPU).
one ALU

CPU switching

* When process switches from running to idle, the PCB needs to be stored in memory. (save state)
* When switch back to running, (reload state)

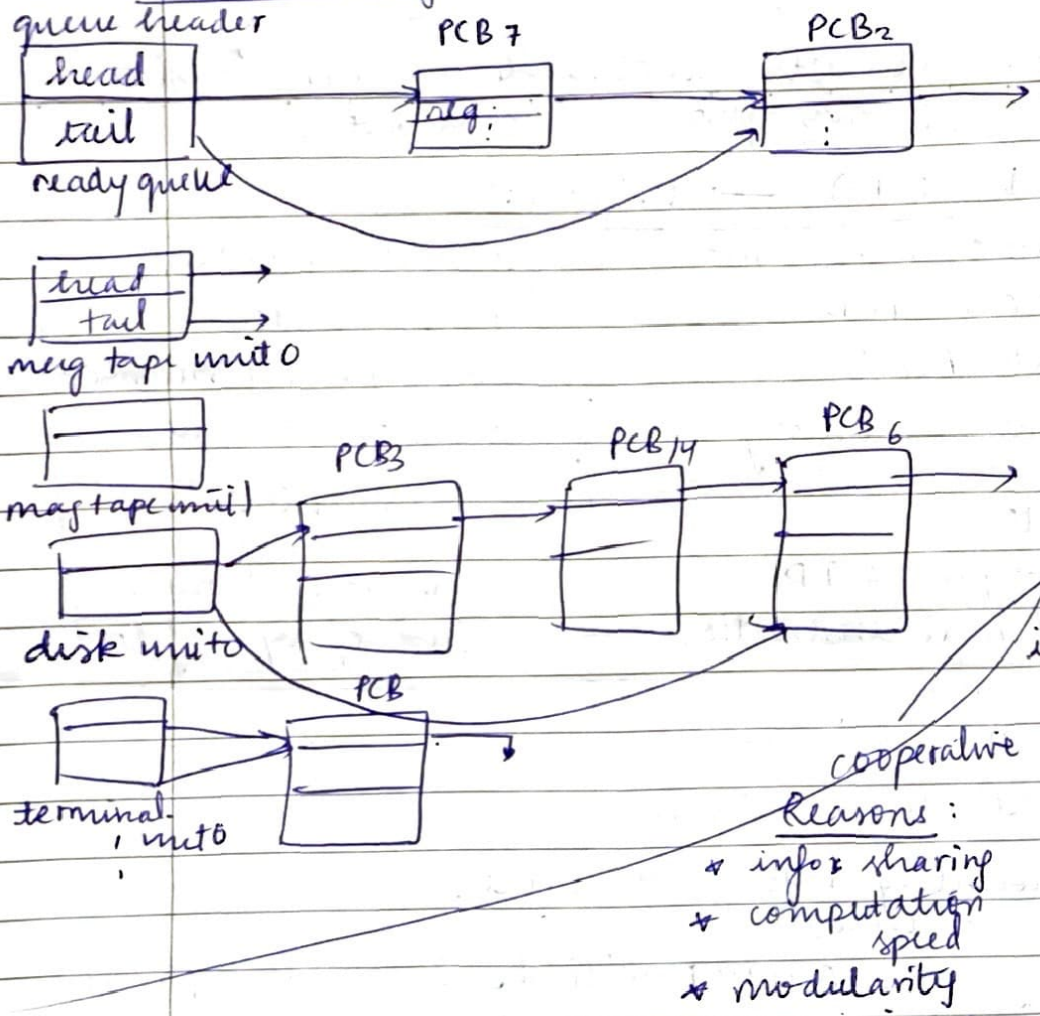
Process Scheduling — Scheduler Types

short term
(CPU scheduling)
→ frequently (ms)
→ allocate CPU

long term
(job scheduling)
→ infrequently (s, min)
→ from ready queue
→ controls multiprogramming

medium term

⊕⊗ Process Scheduling



note:
module is used instead of traditional linear code, because M:M is small

Interprocess comm^{n}

process

independent dependent

cooperative

Reasons:
- info sharing
- computation speed
- modularity
- convinience

Two models of IPC:

- Shared memory
- Message passing

Producer consumer buffer.

Bounded → fixed length.
- Initially buffer empty, producer starts.

unbounded
- no acknowledgement
- infinite capacity

buffer → shared b/w producer and consumer

Message passing

If process P and Q want to comm^{n},
- establish a comm^{n} link
- exchange message in send/receive

In PPT, only till buffering

child
parent

30/10/24

Date :

CPU Scheduling

* OS needs to utilize CPU to max.
* CPU is capable of arithmetic and logical
* Identify how many CPU bound I/O bound instructions in ~~red~~ ready queue.

eg: CPU 80% (arithmetic and logical.) I/O 20%

* CPU burst time is time req for CPU to complete a process.
* I/O burst time is time req for I/O devices

* completion time = CPU burst time + I/O burst time.
* Short term scheduler is used to the algorithms we are learning.
* CPU scheduling takes place when
  1. running → waiting
  2. waiting to ready
  3. running → ready
  4. terminates

non preemptive — can't continue so scheduler has to be called. — ① & ④

preemptive — ② & ③

Scheduling criteria → uses to max extend
* CPU utilization
* Throughput → no of instructions complete per time
* Turnaround time
* Waiting time → process waiting in ready queue.
* Response time

Turnaround time → time b/w job submission & completion

Response → first response it got when scheduled

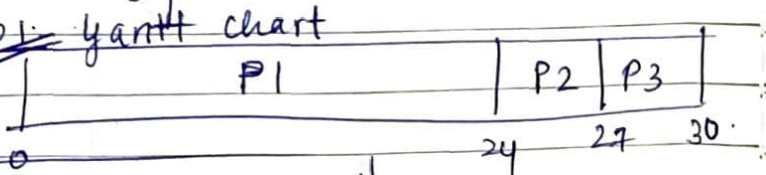
1) First-come, First Served (FCFS) Scheduling (non preemptive)

waits until a process is complete

eg: order: P1, P2, P3.

Process	Burst time (ms)
P1	24
P2	3
P3	3

Step 1: Gantt chart



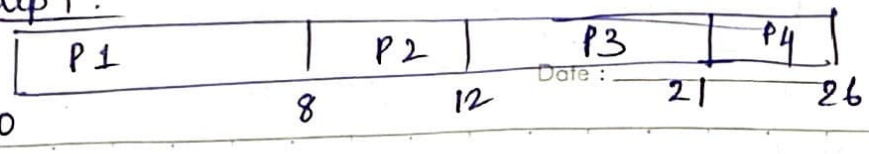
Arrival time → enter the ready queue (0ms by default)

Step 2:	Waiting time	Turnaround time
P1	0	24
P2	24	27
P3	27	30
avg	17	27

eg:

P1	8
P2	4
P3	9
P4	5

Step 1:



Step 2: Waiting time

Step 3: Turnaround time (TAT)

	Waiting time	TAT
P1	0	8
P2	8	12
P3	12	21
P4	21	26
avg	10.25	16.75

note:

TAT = waiting time + burst time

Waiting time = start time − arrival time

eg:

P1	8	0
P2	4	1
P3	9	2
P4	5	3

Waiting time

P1 = 0 − 0 = 0
P2 = 8 − 1 = 7
P3 = 12 − 2 = 10
P4 = 21 − 3 = 18
avg = 8.75

TAT

P1 = 0 + 8 = 8
P2 = 7 + 4 = 11
P3 = 10 + 9 = 19
P4 = 18 + 5 = 23
15.25

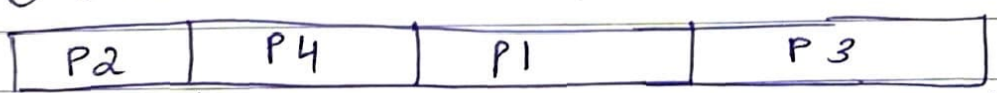
* easy to code
* not effective process.

[2] Shortest Job First Scheduling (non preemptive)

eg:
* gives minimum waiting time.
* choose job with least burst time

P1	8
P2	4
P3	9
P4	5

① order: P2, P4, P1, P3



Step 2:

	Waiting time	TAT
P1	9	17
P2	0	4
P3	17	26
P4	4	9
avg	7.5	14

eg: P1 24
P2 3
P3 3

P2	P3	P1

0 3 6 30

	wt	TAT
P1	6	30
P2	0	3
P3	3	6
avg	3	13

4/11/24

3 Shortest Job First (preemptive)
↳ Shortest Remaining Time First Algorithm.

eg: ①

Ready queue	CPU Burst time	Arrival time
P1	8	0
P2	4	1
P3	9	2
P4	5	3

* When new process, ~~scheduling~~ scheduler takes a break, and decides whether to continue or choose new process (preempt ~~new~~ old process)

P1	P2	P4	P1	P3

0 1 5 10 17 26

	Waiting time	TAT
P1	10 − 1 − 0 = 9	9 + 8 = 17
P2	1 − 1 = 0	0 + 4 = 4
P3	17 − 2 = 15	15 + 9 = 24
P4	5 − 3 = 2	2 + 5 = 7
avg	6.5	13

②:

	Arrival time	CPU burst time
P1	0.0	6 3
P2	3.0	5 3
P3	3.0	3 1
P4	5.0	5 5

Gantt chart

will be same for preemptive and non-preemptive.

~~(non preemptive)~~

P1	P3	P2	P4

0 6 9 14 19

~~(preemptive)~~

P1	P3	P1	P2	P4

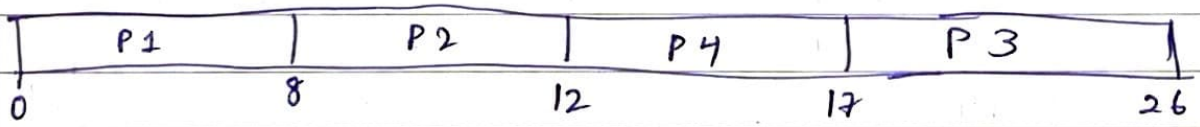
0 3 6 9 14 19

	Waiting Time	TAT
P1	0	0+6 = 6
P2	9-3 = 6	6+5 = 11
P3	6-3 = 3	3+3 = 6
P4	14-5 = 9	9+5 = 14
avg.	4.5	9.25

Date :

note: Non preemptive SJF & when arrival time is diff

P1	8	0
P2	4	1
P3	9	2
P4	5	3



	Waiting	TAT
P1	0 - 0 = 0	0+8 = 8
P2	8 - 1 = 7	7+4 = 11
P3	17 - 2 = 15	15+9 = 24
P4	12 - 3 = 9	9+5 = 14
avg:	7.75	14.25

4 Priority Scheduling algorithm

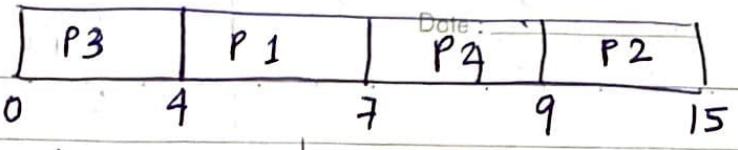
* CPU is allocated to highest priority process from ready queue. Each process has priority no. and it is unique.
* sometimes if 2 or more process have same priority, then FCFS algo. is applied for solving the tie.
* In case studies, we assume that lowest no. of have highest priority.
* Priority can be preemptive or non preemptive.

Multiprocessor scheduling

eg: **Non preemptive**

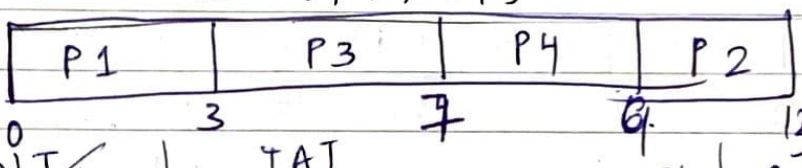
Arrival time of all = 0.

RQ	CPU B.T	Priority
P1	3	2
P2	6	4
P3	4	1
P4	2	3



	W.T	TAT
P1	4	4+3=7
P2	9	9+6=15
P3	0	0+4=4
P4	7	7+2=9
avg	5	8.75

If arrival time is 0, 1, 2, 3



	WT	TAT
P1	0-0=0	0+3=3
P2	9-1=8	8+6=14
P3	3-2=1	4+4=5
P4	7-3=4	4+2=6
avg	3.25	7

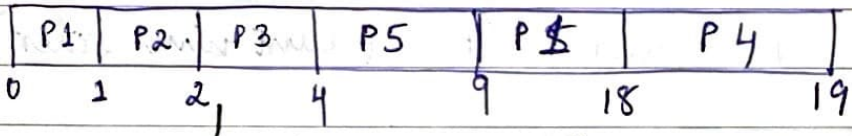
	WT	TAT
P1	0	0+3=3
P2	6-1=5	5+6=11
P3	3-2=1	1+4=5
P4	6-3=1	4+2=3
avg	1.75	

eg: **Preemptive**

note: For priority, focus on Priority column only

	B.T	Priority	A.T
P1	10	3	0
P2	1	1	1
P3	2	3	2
P4	1	4	3
P5	5	2	4

Gantt chart.



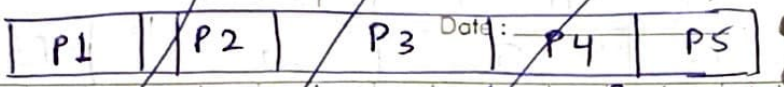
since P1 is in swap and will take extra clock cycle to become ready, P3 is chosen

this is only when they have same priority

	W.T	T.A.T
P1	9-0-1=8	8+10=18
P2	1-1=0	0+1=1
P3	2-2=0	0+2=2
P4	18-3=15	15+1=16
P5	4-4=0	0+5=5
avg	4.6	8.4

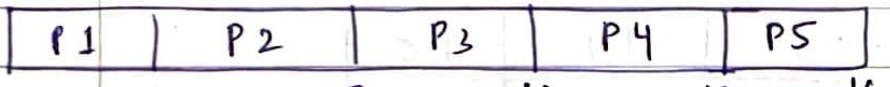
process	B.T	Priority	A.T
P1	7	3	0
P2	2	2	3
P3	3	1	4
P4	1	1	4
P5	3	3	5

① Non preemptive FCFS



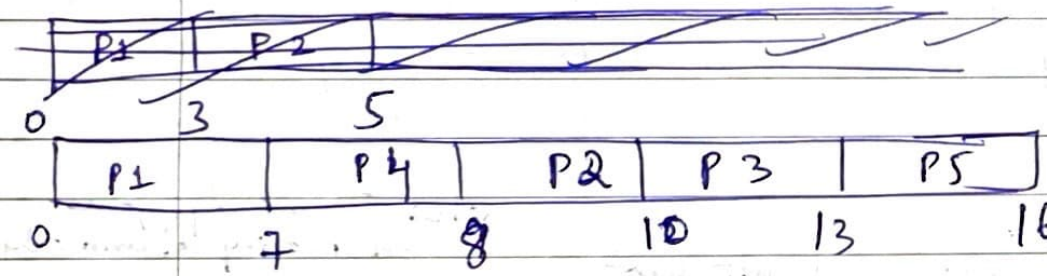
	WT	TAT
P1	0	
P2	3	
P3	5	
P4		
P5		

① Non preemptive FCFS (arrival order)



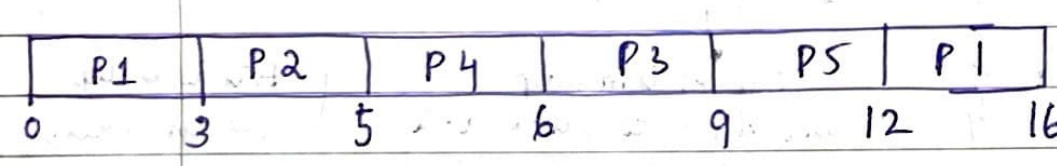
	WT	TAT
P1	0-0=0	0+7=7
P2	7-3=4	4+2=6
P3	9-4=5	5+3=8
P4	12-4=8	8+1=9
P5	13-5=8	8+3=11
avg	5	8.2

② Non preemptive SJF (burst time order)



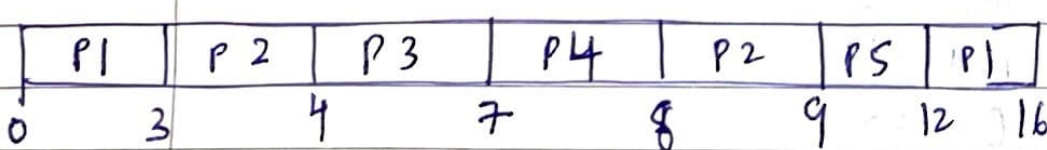
	WT	TAT
P1	0-0=0	0+7=7
P2	8-3=5	5+2=7
P3	10-4=6	3+6=9
P4	7-4=3	3+1=4
P5	13-5=8	8+3=11
avg	4.4	7.6

③ Preemptive SJF (burst time order)



	WT	TAT
P1	12-0-3=9	9+7=16
P2	3-3=0	0+2=2
P3	6-4=2	2+3=5
P4	5-4=1	1+1=2
P5	9-5=4	4+3=7
avg	3.2	6.4

④ Preemptive Priority (priority order)



	WT	TAT
P1	12-3-0=9	9+7=16
P2	8-3-1=4	4+2=6
P3	4-4=0	0+3=3
P4	7-4=3	3+1=4
P5	9-5=4	4+3=7
avg	4	7.2

(5) Non preemptive priority (priority order)

P1	P3	P4	P2	P5
0	7	10	11	13	16

	W.T	TAT
P1	0-0=0	0+7=7
P2	11-3=8	8+2=10
P3	7-4=3	3+3=6
P4	10-4=6	6+1=7
P5	13-5=8	8+3=11
avg	5	8.2

(6) Round Robin Scheduling Algorithm.

* shares CPU with all process equal amount of time
* process selection is like FCFS
* Preemptive by nature.

eg:

Process	CPU burst time
P1	6
P2	3
P3	2
P4	5

Time slice = 2

P1	P2	P3	P4	P1	P2	P4	P1	P4
0	2	4	6	8	10	11	13	15	16

	WT	TAT
P1	13-2-2=9	9+6=15
P2	10-2=0=8	8+3=11
P3	4-0=4	4+2=6
P4	15-2-2-0=11	11+5=16
avg	8	12

eg:

P1	7	0
P2	2	3
P3	3	4
P4	1	4
P5	3	5

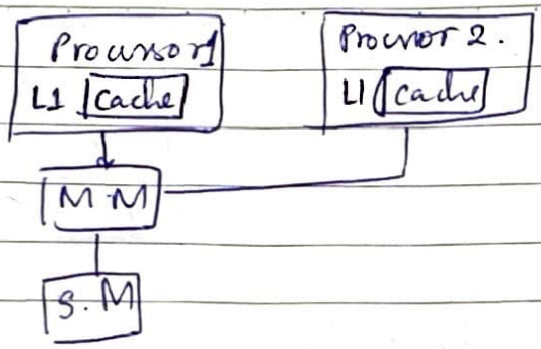
Time slice : 1ms

P1	P1	P1	P2	P3	P4	P5	P1	P2	P3	P5	P1	P3	P5	P1	P1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

	WT	TAT
P1	15-6-0-9	9+7=16
P2	8-1-3=4	4+2=6
P3	12-2-4=6	6+3=9
P4	5-4=1	1+1=2
P5	13-2-5=6	3+6=9
avg	5.2	8.4

Process synchronization

- two or more dependent processes are working on same data.



note: counter ++ / counter --

↳ reg1 = counter / reg2 = counter
reg1 = reg1 + 1 / reg2 = reg2 - 1
counter = reg1 / counter = reg2

Producer

```
while (true){
    while (counter == SIZE)
        /* do nothing */
    buffer[i] = next_prod.
    i = (i+1);
    in = (in+1) % SIZE
    counter ++
```

Consumer

```
while(true){
    while (counter == 0)
        // do nothing
    next_con = buffer[out]
    out = (out+1) % SIZE
    counter --;
```

* Race condition : while producer is executing consumer starts execution causing inconsistency in data.
  - solution : Synchronization

* Critical section: Part of M.M reserved for process coordination
  If any var has its value continuously changed, those are stored in critical section.
  - solution:
    1. Mutual exclusion
    2. Progress
    3. Bounded waiting

```
do{
    entry section      <- ask permission
    critical section
    exit section
    remainder section
}while(true)
```

* Peterson's solution:

shared variables
- int turn;
- boolean flag[2];

* only two process applicable

P1:

```
do{ flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    critical
    flag[i] = false;
    remainder
} while(true);
```

P2:

```
do{ flag[j] = true;
    turn = i;
    while (flag[i] && turn == i);
    critical
    flag[j] = false
    remainder
} while(true)
```

Synchronization Hardware

* locking – protecting critical section by locks.

Date : 13 | 11 | 24

test_and_set instruction

```
boolean test_and_set (boolean *target){
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

} executed as one instruction – h/w solution
CPU instruction

① Executed atomically
② Returns original value of passed parameter
③ Set the new value of passed parameter as TRUE

(*)(*)

```
do { while (test_and_set (&lock));
        // do nothing
        /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

lock is shared variable

(case i) P1 enters critical ~~cati~~ section.
* Execute test_and_set by passing lock.
* Initially lock is FALSE
* make lock = TRUE and return false. so P1 enters critical section.
* once done, lock again becomes false.

Q) If P1 is in critical section and P2 wants enter critical section, will P2 be allowed or denied?
No because, * when P2 wants to enter, it executes test_and_set with lock value TRUE.
* makes lock = TRUE and returns true.
* Hence ~~P2~~ nothing happens and P2 doesn't enter critical section.

(*)(*).

Q) If P2 is in remainder section, and it wishes to enter critical section, is it allowed or denied?

compare-and-swap instruction

```
int CAS (int *value, int expected, int new_val)
    int temp = *value;
    if (*value == expected)
        *value = new_val;
    return temp;
}
```

① Execute atomically
② Return 0 or val of value
③ if val = expected, set val to passed value of new_val.

```
do{ while (CAS (&lock, 0, 1) != 0);
        // do nothing
        /* critical section */
    lock = 0; /* remainder */
}while (true);
```

case i) If P1 wants to enter CS.
* P1 executes CAS with lock initially '0', expected is 0, new_val is 1.
* make lock as 1 since lock == expected. and returns 0.
* while loop will be 0 != 0 so P1 enters CS. → false.
* once done lock becomes 0.

case ii) P2 wants to enter CS.
* P2 executes CAS with lock = 1, expected = 0, new_val = 1.
* Since lock ≠ expected, lock value doesn't change.
* so CAS returns 1. In loop condition 1 != 0 becomes true, so it does nothing.
* So P2 cannot enter.

Bounded waiting mutual exclusion with test_and_set.

```
do{ waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set (&lock);
    waiting[i] = false;
        /* critical section */
    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
```

lock = F
no process in CS
lock = T
process in CS

```
if (j==i)
    lock = false;
else
    waiting[j] = false;
/* remainder */
} while(true);
```

Initially, lock = F, key = F.

waiting

F	F	F	F
0	1	2	3

P1 P2 P3 P4.

(case i) P1 wants to enter CS.

* | T | F | F | F |
  | 0 | 1 | 2 | 3 |
  key = T.
* while condition is satisfied, execute TAS.
  key = F, lock = T (CS is occupied)
* while not satisfied, exit.
* | F | F | F | F |, P1 in the CS.

(case ii) P4 wants to enter CS.

* | F | F | F | T | key = T. lock = T.
* while condition is satisfied, execute TAS
  key = T, lock = T.
* goes in infinite loop.
* if P1 completes execution,
  * j = (0+1)%4 = 1., i = 0.
  * ~~j != i and~~ keep incrementing j until i = j or waiting[j] = T.
  * since waiting[3] = T, waiting[3] = F (in ~~wait~~ else)
* now, in P4, the while condition is not satisfied (since waiting[3] = F).
* so | F | F | F | F | and P4 enters CS.

(case iii) P2 ~~enters~~ wants to enter CS.

* | F | T | F | F | key = T lock = T.
* while satisfied, execute TAS key = T lock = T.
* infinite loop.
* Repeat above case(ii)

~~* Ans~~

Note: If P4 is in CS, P3 wants to enter CS.
* once P4 completes execution, j = (3+1)%4 = 0, i = 3
* check while loop, satisfied, j increments
  j = (1+1)%4 = 1, i = 3
  * j = 2, i = 3.
    ↳ waiting[2] = T.

Q) If P1 completes CS, and it wants to enter CS again, will P1 be allowed or denied?

Semaphore - sftw tool for synchronization
* integer value accessed/modified by wait() and signal().

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

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

CS (mutual exclusion)

```
do{ wait(mutex); // block
      // CS
    signal(mutex); // continue
      // Remainder
} while(true);
```

no Busy waiting

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

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

Bounded Buffer Problem

n buffers, each can hold one item
Semaphore mutex initialized to 1
Semaphore full initialized to 0.
Semaphore empty initialized to n.

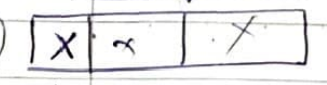
Producer:

```
do{ /* produce an item in
       next_produced */
    wait(empty);
    wait(mutex);
    /* add next produced to buffer */
    signal(mutex);
    signal(full);
} while(true);
```

I ① Initially n = 3 : empty = 3, mutex = 1, full = 0

② wait (empty) → wait (3). empty = 2.

③ wait (mutex) → wait (1). ~~wait~~ 0. mutex = 0.

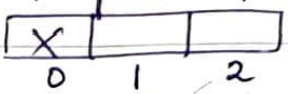
④ 

enter CS.

⑤ signal (1) mutex = 1

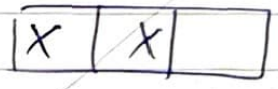
⑥ signal (0) full = ~~0~~ 1

∴ mutex = 1, full = 1, empty = 2.



II ① wait (2) empty = 1

② wait (1) mutex = 0

③ 

④ signal (0) mutex = 1

⑤ signal (1) full = 2

case i) When producer in CS, and consumer tries to CS.

mutex = 1/0.
empty = 3/2.
full = 0.

+ wait (full) → wait (0)
full = -1.
Add procss to list
block()

~~Now producer completes CS. to wait (mutex)~~

```
consumer:
do { wait(full);
     wait (mutex);
     /* remove item */
     signal (mutex);
     signal (empty);
     /* consume item */
} while (true);
```

case ii) If the buffers are full & the producer tries to enter CS, is it allowed?

case iii) If buffers are empty & consumer tries to enter CS, is it allowed?

case iv) If consumer is in CS, the producer is called, is it allowed or denied? ~~(producer~~ empty = 0, full = 3.

case ii) In this case, ~~empty~~ mutex = 1, full = 3, empty = 0.

* producers tries to execute code.

① wait (empty) → wait (0) ∴ empty = -1.

Adds this process to ~~list~~ list and blocks the process.

So producer will not be allowed

iii) In this case, mutex = 1, full = 0, empty = 3.
Consumer executes the code. Date :
- wait (full) → wait(0) → full = -1.
adds process to list and blocks the process.

∴ consumer is denied.

iv) mutex = 1, full = 3, empty = 0.
consumer.
- wait (full) → full = 2.
- wait(mutex) → mutex = 0.
- CS.

produces :
- wait (empty) → empty = -1. (blocks the producer).

* Reader-writer problem

* Dining philosopher.