

## Process Synchronization

- ① Mutual Exclusion :- If one process is using CS then other process can't use that CS.
- ② Progress :- If <sup>no</sup> any process is in CS & at least one process wants to enter in CS then it should be allowed.
- ③ Bounded Waiting :- waiting of process should be bounded, process can't wait for  $\infty$  time.

### Solution for Synchronization

#### ① Using lock (S/W sol<sup>n</sup>)

```

Boolean lock = false;
while(true){
    while(lock);
    lock = true;
    CS
    lock = false;
    RS
}

```

P<sub>0</sub>

```

while(true){
    while(lock);
    lock = true;
    CS
    lock = false;
    RS
}

```

P<sub>1</sub>

Mutual Exclusion	X
Progress	✓
Bounded wait	✓

#### ② Using turn [strict alternation] (S/W sol<sup>n</sup>)

```

int turn = 0;
while(true){
    while(turn != 0);
    CS
    turn = 1;
    RS
}

```

P<sub>0</sub>

```

while(true){
    while(turn != 1);
    CS
    turn = 0;
    RS
}

```

P<sub>1</sub>

Mutual Exclusion	✓
Progress	X
Bounded wait	✓

### ③ Peterson Solution (s/w soln) [two process soln]

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

```
while(true){
    flag[0] = true;
    turn = 1;
    while (!flag[1] && turn == 1);
    CS
    flag[0] = false;
    RS;
}
```

must be same

P<sub>0</sub>

```
while(true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    CS
    flag[1] = false;
    RS;
}
```

P<sub>1</sub>

Termination	✓
Progress	✓
Bounded	✓

### ④ Test And Set ( ) [Hardware Instruction]

```
Boolean lock = false;
Boolean TestAndSet(Boolean *tobj){
    Boolean rv = *tobj;
    *tobj = true;
    return rv;
}
```

Atomic

```
while(true){
    while (!TestAndSet(&lock));
    CS
    lock = false;
}
```

P<sub>0</sub>

### ⑤ Swap ( )

```
Boolean key, lock = false;
void Swap(Boolean *a, Boolean *b){
    Boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

```
while(true){
    Key = true;
    while (Key == true){
        Swap(&lock, &key);
        Key = false;
    }
    lock = false;
    RS
}
```



## # Bounded Buffer Problem

Mutex = 1 → to lock on buffer (Binary Semaphore) [mutual Exclusion]  
 Full = 0 → counting semph., to # of occupied slots.  
 Empty = n → " " , to # of empty slots.

Producer() {

wait(empty) // to check if empty space available

// produce item

wait(mutex) // buffer is shared, only one access at time.

// add item on

// buffer

signal(mutex)

signal(full)

}

Note:

Full → भरी हुई है!

Empty → खाली जगह है!

consumer() {

wait(full)

wait(mutex)

// remove an item from buffer

signal(mutex)

// consume the item.

signal(empty)

Empty used for not produce if buffer is full

full used for not consume if buffer is empty

mutex used for protect Buffer to avoid mutual exclusion

3

Table: Properties of all the solution

Solution	Mutual Exclusion	Progress	Bounded wait
Lock Variable	X	✓	X
Static Allocation (Dekker's Algo)	✓	X	✓
Peterson's Algo	✓	✓	✓
TSL Instruction Set	✓	✓	X



## # Reader - writer problem -

(4)

→ for protection of readcount variable

mutex = 1 → Bin. Sem. for Mutual Exclusion

wrt = 2 → Bin Sem. to restrict readers or writers on writing

readcount = 0 → Integer variable, no. of active readers.

```
writer() {
```

```
    wait(wrt)
```

```
    // perform writing
```

```
    signal(wrt)
```

```
}
```

```
Reader() {
```

```
    wait(mutex)
```

```
    readcount ++
```

```
    if (readcount == 1)
```

```
        wait(wrt)
```

```
    signal(mutex)
```

```
    // perform reading
```

```
    wait(mutex)
```

```
    readcount -- ;
```

```
    if (readcount == 0)
```

```
        signal(wrt)
```

```
    signal(mutex)
```

```
}
```

## # Dining philosopher problem

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

```
// Bin Sem.
```

```
/*
```

```
    // all will run
```

```
    wait(chopstick[i])
```

```
    wait(chopstick[(i+1)%5])
```

```
    // eat
```

```
    signal(chopstick[i])
```

```
    signal(chopstick[(i+1)%5])
```

```
    // 1 philosopher will run
```

```
    wait(chopstick[(i+1)%5])
```

```
    wait(chopstick[i])
```

```
    // eat
```

```
    signal(chopstick[(i+1)%5])
```

```
    signal(chopstick[i])
```

# Job with same burst length, round arrive at same time then RR scheduler not able to provide better average turn around time than FIFO



## # Round scheduling

Note:- In R.R, If Arrival time is zero of all process P, Q and R then they will schedule at this order.  $\boxed{P|Q|P|P|Q|R} \dots$

Note:- If Burst time is given + no preemitive then for min. avg waiting time, use  $\boxed{SFS}$  else use  $SFRS$

$$\boxed{TAT = CT - AT} \quad \boxed{WT = TAT - BT}$$

$$\boxed{RT = FR - AT}$$

$$\boxed{\text{Response Ratio} = \frac{W.T + B.T}{B.T}}$$

# used in Highest response ratio next

$$\boxed{\text{Throughput} = \frac{\# \text{ of process}}{\text{Max}(CT) - \text{Min}(AT)}}$$

## # term

TAT - Turn around Time

BT - Burst time

CT - Complete time

RT - Response Time

AT - Arrival time

FR - First response time

WT - Waiting time

Note:- If we have to use LRTF, and arrival time is zero and sum of Burst time is B the Turn around of process from <sup>higher</sup> ~~lower~~ ID to ~~lower~~ <sup>higher</sup> ID will be  $B, B-1, B-2, \dots$

Note:- Response time - The time difference b/w first response and arrival time.

# note:- SJF < SRT  
 less min      more min

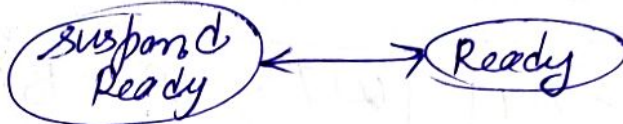
take minimum average, W.T, TAT, TAT-...  
 out of all algo.

### Schedules

• Long term scheduler



• Medium " "



• (Swapping)



• Short " "



### Scheduling table

- FIFS - NO - every process get chance due to arrival time
- NP, SJF - YES - If shorter complete and suddenly shorter come in ready.
- SRJF - YES - Same as SJF
- PF - NO - Every process will run as per arrival time
- NP, LSF - YES - If longer complete and suddenly long come.
- LRTF - NO - Memory is limited, It will run all till they got short time.
- NP, Priority - YES - If more priority come again and again
- PF, Priority - YES - Same as above.
- HRRN - NO - Response ratio will calculate after seeing waiting time
- MLQ - YES -
- MLFQ - YES -



- # RR is better than FCFS in terms of response time
  - # Reducing the quantum length of a RR will tend to improve the responsiveness of scheduled job.
  - # For a given set of jobs, all non-pre-emptive scheduling policies will require the same amount of context switch time overhead.
- 

### Process

- Program under execution
- contains stack, heap and data section/global section

### Program →

- ① Passive & static
- ② It resides in secondary memory
- ③ File containing

# OS - Introduction & Background

[src: made-easy notes]

- # multi-programming OS:- If one job is leaving the CPU to perform I/O operation, then other jobs which is ready for execution will be scheduled onto the CPU.
- # Multi-tasking:- multiple job executed time-sharing mode.
- # Synchronous I/O:- Process perform I/O operation in blocked state.
- # Asynchronous I/O:- process is not placed in the blocked state.

Fork -

```
main(){
```

```
int pid;
```

```
pid = fork();
```

```
}
```

- Fork return, -ve value to parent if child process creation process is unsuccessful
- " " , 0 value to newly created child process
- " " , +ve value (process ID of child) to parent process.
- Parent and child process have same virtual address, but physical address is different.

Note:-

If program has, 'n' fork calls then, there will be  $2^n - 1$  child process created



# Dead lock

7

## # Condition for mutual-Exclusion deadlock

- ① Mutual Exclusion
- ② Hold & wait
- ③ No-preemption
- ④ circular wait.

## # Recovery from Deadlock

- ① Make sure that deadlock never occur.
  - Prevent the system from deadlock or avoid deadlock
- ② Allow deadlock, detect and recover
- ③ Pretent that there is no any deadlock.

## # Some topic notes

- one switching two kernel threads in context switching register, PC and SP must be changed, but memory context remain same
- user to kernel mode can occur by interrupt or system call.
- Multilevel feedback Queue
  - Short IO > short CPU > long IO > long CPU
  - ← Priority
- On Receiver Interrupt, kernel will give sudden interrupt service

# Memory management

In Virtual address

page number	page offset
-------------	-------------

In Physical address

frame number	frame offset
--------------	--------------

# Page Table format



Let  $A$  = Page number  
 $f$  = frame number  
 $O$  = offset  
 $N = 2^p$   
 $M = 2^f$

Page Entry size  $n$

$$n = \# \text{ bits for } f + \text{ status bits}$$

$$\text{Size of table} = N \times n \Rightarrow 2^p \times n \text{ bits}$$

# TLB - Table look aside buffer

Tag	Index	offset
-----	-------	--------

$$\text{Tag} = V.A \text{ bit} + \log(\text{Associative})$$

$$\text{Index} = \log(\frac{\text{line number}}{\text{Associative}})$$

$$\text{Tag} = V.A + \log(Asso) - \text{Index}$$

$$\text{TLB Entry} = \text{Tag bit} + P.A + \text{status bit}$$