Operating System

It is a system software that allows one to perform task on hardware conveniently & efficiently. It is a control program, controls the operation of user program and also concerned with the operation and control i/o devices.

- It also act as a resource allocator or resource manager
- OS provide services to users of the system.

Operating System Functions

* Process Management
* Memory Management
* File System Manipulation
* I/O devices Management
* Protection and Security.

Process Management

Process is defined as program in execution. Program by itself is not a process. A program is a passive entity such as contents of a file stored on a disc whereas process is a active entity with a program counter specifying the next instruction to execute ~~a set~~ and a set of associated resources.

A process may contain program code and also includes current activity as represented by a value of the program counter and contents of the processors register. And in addition which contains the temporary data &

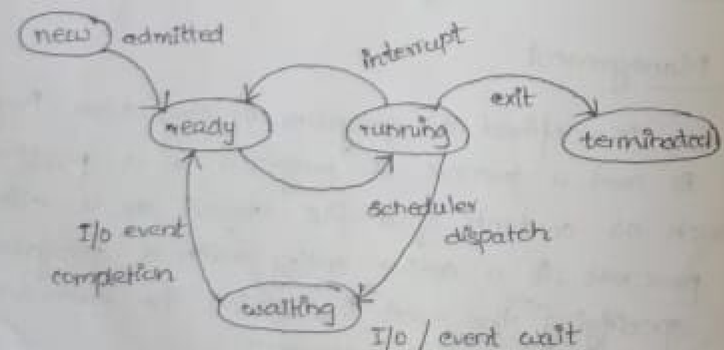
a data section which contains global variable.

Different states

Process state :

The state of a process is defined in part by the current activity of that process.
Each process may be in one of the following states :

1. New - the process being created.
2. Running / execution state : instructions are being executed.
3. Waiting state : The process is waiting for some event to occur.
4. Ready state : The process is waiting to be assigned to the processor.
5. Terminated : The process has finished execution.



Process State Diagram.

CPU Scheduling

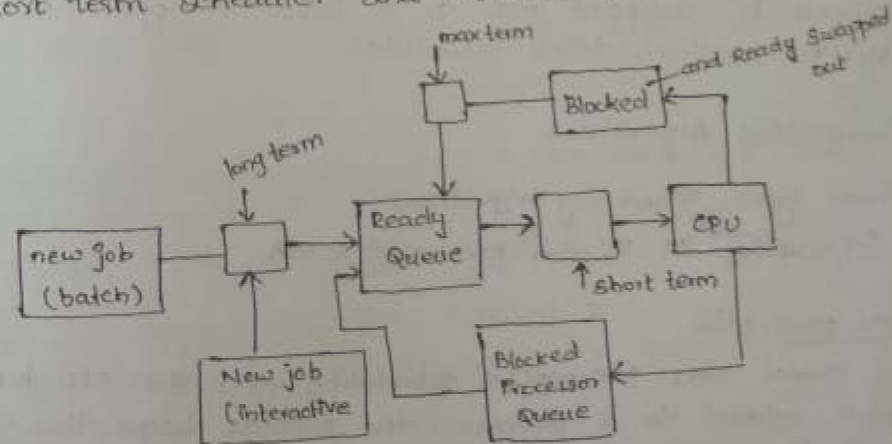
Types of schedulers

When there are more than one process ready to execute with the processor a selection decision need to be made to pick a process for execution from among the ready processes. This activity is called process scheduling.

The different types of scheduling queue's are :-

i) Job Queue
ii) Ready Queue
iii) Device Queue

* Different types of schedulers : long term scheduler, short term scheduler and medium term scheduler.



Long term schedulers

The long term scheduler or the job scheduler select processes from its pool and load them into memory for execution.

Short term scheduler

The short term scheduler or CPU scheduler select from among the processes that are ready to execute & allocate the CPU to one of them.

Medium term scheduler

Medium term scheduler remove the processes from memory & thus reduces the degree of multi programming at some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is known as swapping. The process is swapped out & is later swapped in by the medium term scheduler

3rd module

Scheduling Algorithms

Two types scheduling algorithms :

i) Pre emptive SA ii) Non pre-emptive SA.

*

Non preemptive.

Under non ~~p~~reemptive scheduling once the CPU has been alloted to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to a waiting state.

OS : Windows 3.1 ⇒ Eg.

Premptive scheduling.

In premptive scheduling, the process replace with another process when an higher priority process in the Ready Queue.

Scheduling Criterias

i) CPU utilization : utilizing CPU idle time.

ii) Throughput : the no. of processes completed per unit time is known as throughput.

iii) Turnaround time : the interval from the time of submission of a process to the time of completion is the turn around time & turnaround time is the sum of periods spend waiting to get into memory, waiting in the ready queue, execution on the CPU and doing I/O request.

Turnaround time (TAT) P_i = $CT(P_i) - AT(P_i)$

Completion time - Arrival time

iv) Waiting time : waiting time is the sum of periods spend waiting in the ready queue.

WT - TAT - BT (Burst time) total time taken to complete process

v) Response time : the time interval b/w the time from ~~the~~ submission of a request until the first response is produced

CPU scheduling algorithms

i) First - Come, First Served (FCFS)

In ~~with~~ this scheme, the process that request the CPU first is ~~alloted~~ allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue its PCB (Process Control Block) is linked onto the tail of the queue. When the CPU is free it is allocated to the process at the head of the queue.

Q. Determine the average waiting time under the FCFS policy.

Process	Burst time
P_1	24
P_2	3
P_3	3.

All the processes are arrived at time 0.

* Gantt chart

P_1	P_2	P_3
0 – 24	24 – 27	27 – 30

Process	Burst time	TAT CT-AT	WT TAT-BT
P_1	24	24	0
P_2	3	27	24
P_3	3	30.	27

The average waiting time = 0+24+27/3 $\left(\frac{\Sigma WT_i}{n}\right)$

= 51/3 = 17 ms

The average waiting time under FCFS policy is often quite long.

The FCFS is non-preemptive i.e., once the CPU has been alloted to a process that process keeps the CPU until it releases the CPU. either by terminating or by requesting I/O.

Disadvantage

Convey effect: There is a convey effect.

All the other processes wait for the one big process to get off the CPU. This affect result in lower CPU & device utilization than might be possible if the shorter process were allowed to go the first.

Shortest Job First (SJF) / optimum Algorithm

SJF
- SJF non preemptive
- Shortest remaining job first / next

This algorithm associated with each process, the length of the latters next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

Q. Find the average waiting time using SJF scheduling

Process	AT	BT
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Non - preemptive

Gantt chart

P1	P2	P4	P3

0   8   12   17   26

Process	AT	BT	(CT-AT) TAT	(TAT-BT) WT
P_1	0	8	8-0 = 8	0
P_2	1	4	12-1 = 11	7
P_3	2	9	26-2 = 24	15
P_4	3	5	17-3 = 14	9

Average waiting time = $\frac{0+7+15+9}{4}$

= 31/4 = 7.75

Preemptive

Gantt chart

P1	P2	P4	P1	P3

0   1   5   10   17   26

Process	AT	BT	TAT	TAT-BT WT
P_1	0	8	17-0 = 17	9
P_2	1	4	5-1 = 4	0
P_3	2	9	26-2 = 24	15
P_4	3	5	10-3 = 7	2

AWT = 9+15+2+0/4

= $\frac{13}{26}/\frac{}{4_2}$ = 6.5 ms

The next CPU Burst is generally predicted as an exponential average of the measured length of the previous CPU Burst. Let t_n be the length of nth CPU burst & T_{n+1} be our predicted value for the next CPU burst. Then α, $0 \le \alpha \le 1$.

Defined

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n$$

This formula defines an exponential average.

Advantage: decrease waiting time, ∴ increase response.

Disadvantage: have to use formula to predict burst time.

Priority Scheduling. Both pre & non-preemptive

~~It is a special case of shortest job first algorithm~~

Shortest Job first algorithm is a special case of priority scheduling algorithm

A priority is associated with each process and the CPU is allocated to the process with highest priority.

Process	BT	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Arrived at time 0.

Non - preemptive

Gantt chart

~~Process~~

P4	P3	P1	P5	P2

0 1 3 13 18 19

Process	BT	TAT	WT
P1	10	13-0 = 13	13-10 = 3
P2	1	19-0 = 19	19-1 = 18
P3	2	3-0 = 3	3-2 = 1
P4	1	1-0 = 1	1-1 = 0
P5	5	18-0 = 18	18-5 = 13

Average waiting time = $\frac{3+18+1+13+0}{5}$ = 35/5

= 7

* Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with priority of the currently running process.

A preemptive priority scheduling will preemp to the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

Disadvantage :

Indefinite locking / blocking or starvation

A process that is ready to run but lacking the CPU can be considered as blocked or waiting for the CPU.

A solution to the problem of starvation is aging. Aging is the technique of gradually increasing the priority of processes that wait in the system for a long time.

Round Robin scheduling (RR sch.) Preemptive

RR scheduling algo is designed for time sharing systems. It is similar to the FCFS algorithm. But preemption is added to switch b/w processes. A small unit of time called time quantam / slice is defined and the ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of upto 1 time quantum.

Process	Burst Time
P1	24
P2	3
P3	3

Time slice - 4 ~~mts~~ ms. AT = 0

Gantt chart

P1 P3 P2 P1

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

P1 24 20

P2 3

P3 3

Process	BT	TAT	WT
P1	24	30-0 = 30	30-24 = 6
P2	3	7-0 = 7	7-3 = 4
P3	3	10-0 = 10	10-3 = 7

Average waiting time = 6+4+7/3 = 17/3 = 5.66

Non - preemptive

Gantt chart

~~Process~~

P4	P3	P1	P5	P2

0 1 3 13 18 19

Process	BT	TAT	WT
P_1	10	13-0 = 13	13-10 = 3
P_2	1	19-0 = 19	19-1 = 18
P_3	2	3-0 = 3	3-2 = 1
P_4	1	1-0 = 1	1-1 = 0
P_5	5	18-0 = 18	18-5 = 13

18
3
1
13
35
5
= 7

Average waiting time = $\frac{3+18+1+13+0}{5}$ = 35/5

= 7

* Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with priority of the currently running process.

A preemptive priority scheduling will preemp to the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

Disadvantage :

Indefinite locking / blocking or starvation

A process that is ready to run but lacking the CPU can be considered as blocked or waiting for the CPU.

A solution to the problem of starvation is aging. Aging is the technique of gradually increasing the priority of processes that wait in the system for a long time.

Round Robin scheduling (RR sch.) Preemptive

RR scheduling algo is designed for time sharing systems. It is similar to the FCFS algorithm. But preemption is added to switch b/w processes. A small unit of time called time quantum/ slice is defined and the ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of upto 1 time quantum.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Time slice - 4 ~~m/s~~ ms. AT = 0.

Gantt chart

P_1 24 20
P_2 7
P_3 3

$P_1 P_2 P_3 P_1$

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

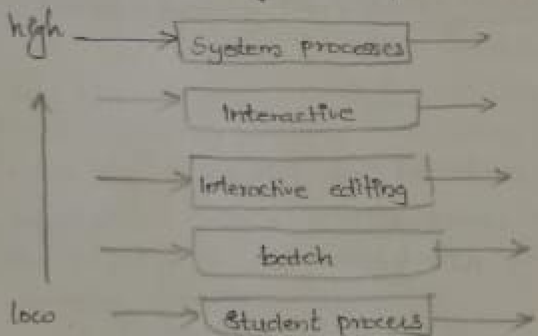
Process.	BT	TAT	WT
P_1	24	30-0 = 30	30 - 24 = 6
P_2	3	7-0 = 7	7 - 3 = 4
P_3	3	10-0 = 10	10 - 3 = 7

Average waiting time = 6+4+7/3 = 17/3 = 5.66

* If n processes in the ready queue & the time quantam is q then each process gets $1/n$ of the CPU time atmost q time unit, then each process must wait no longer than $(n-1)q$ time units until its next time quantum

Multi-level Queue Scheduling

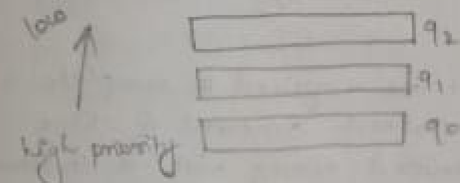
In this scheduling, the processess are clasified into different groups. For eg: a common division is made b/w foreground (interactive processes) and background (batch processes) these 2 type of processes have different response time & have diff scheduling algorithm. In addition foreground processes may have the priority over the background processes.



A multi-level queue scheduling partitions the ready queue into several separate queues based on the property of the process such as memory size, process priority or process type. Each queue has it own scheduling algorithm. The foreground queue might be scheduled ~~typeby~~ RR algorithm while the background queue is scheduled by an FCFS algorithm.

Multi-level feedback queue Scheduling.

Multi level feedback queue scheduling allows a process to move b/w queues. The idea is to separate processes with different CPU burst characteristics. If a process uses too much CPU time it will move to a lower priority queue. Similarly a process that wait too long in the lower priority queue may be moved to a higher priority queue. This form of aging prevents the starvation.



A process entering the queue is put into q_0. A process in the q_0 is given to a time quantum of 8 ms. If it does not finish within this time, it is moved to the tail of q_1. If q_0 is empty, the process at the head of q_1 is given a quantum of 16 ms. If it does not complete it is preempted & put into q_2 and the process in q_2 are run on FCFS basis only when q_0 and q_1 are empty.

~~Multiple~~ Multi-processor scheduling.

If multiple CPUs are available, the scheduling problems is correspondingly more complex.

Two types: Homogeneous system & Heterogeneous system

* Homogeneous system

When the processors are identical in terms of their functionality any available processor can then be used to run any processes in the queue.

Heterogeneous system.

Here the systems are of different kind. Only programs compiled for a given processors instruction set could be run on that processor.

Real time scheduling

Real time computing is divided into 2 types: Hard real time systems & soft real time systems

Hard real time system

In hard real time system are required to complete a critical task within a guaranteed amount of time. Generally a process is submitted along with a statement of the amount of time in which it need to complete. The scheduler then either admits the process guaranting that the process will complete on time or reject the request as impossible (resource reservation).

Soft real time system

Soft real time system is less restrictive. It requires that critical processes recieve priority over less fortunate ones.

Process Synchronization and Coordination

A cooperating process is one that can affect or be affected by other processes executing in the system. The cooperating process may either directly share a logical address space or be allowed to share data only through files.

Cooperating sequential process, all running asynchronously & possibly sharing data.

Producer - Consumer Problem

Producer

```
while (1) {
while (counter == buffersize)
    ;  // do nothing
buffer [in] = nextproduced;
in = (in + 1) % buffersize;
 counter ++;
}
```

Consumer

```
while (1)
{
 while (counter == 0)
   ;
nextconsumed = buffer [out];
out = (out + 1) % buffersize;
 counter --;
}
```

Machine level Implementation

P	C
register1 = counter	register2 = counter
register1 = register1 + 1	register2 = register2 - 1
counter = register1	counter = register2

Concurrent execution of both process lead to a inconsistent state that means the concurrent execution of counter++ & counter-- is equivalent to a sequential execution. Once such interleaving is as follows:

T_0 : P execute register1 = counter {$r_1 = 5$}

T_1 : P execute $r_1 = r_1 + 1$ {$r_1 = 6$}

T_2 : C execute r_2 = counter {$r_2 = 5$}

T_3 : C execute $r_2 = r_2 - 1$ {$r_2 = 4$}

T_4 : P execute counter = register1 {counter = 6}

T_5 : C execute counter = r_2 {counter = 4}

Notice that we have arrived at the incorrect state counter == 4 recording that there are 4 full buffers when in fact there are 5 full buffers. If we reverse the order of statements T_4 and T_5 we would arrive at the incorrect state counter == 6. We could arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this where several processes access & manipulate the same data concurrently & the outcome of execution depends on the particular order in which the access takes place which is called race condition. To guard against the race condition we need to ensure that only one process at a time can be manipulate the variable counter. To make such a guarantee we require some form of synchronization of processes.

Critical Section

Consider a system consisting of n processes $p_0, p_1, \ldots p_{n-1}$. Each process has a segment of code called critical section in which the process may be changing a common variable. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. This is called mutual exclusion.

A solution to critical section problem must satisfy the following 3 requirements:

i) mutual exclusion :- If process p_i is executing in its critical section, then no other process can be executing in their critical sections.

ii) Progress :- If no process is executing in its critical section and some process wish to enter their critical regions, then only those processes that are not executing in their remaining section can participate in the decision on which will enter its critical section next. And this selection cannot be postponded indefinitely.

iii) Bounded waiting :- There exist a bound on the no. of times that the other processes are allowed to enter their critical sections after a process has made a request to enter its critical section & before that request is granted.

Two Process Solutions

Algorithm 1

In the first approach, let the processes share a common integer variable that is turn, initialised to 0 or 1. If turn ==1 then the process Po is allowed to execute its critical section. That means, if turn == 0, Po is ready to enter its critical section, P1 cannot do so ~~even~~ eventhough Po may be in its remainder section. This solution does not satisfy the progress requirement.

Algorithm - 2

The problem with algorithm-1 is that it does not retain sufficient information about the state of each process. A remedy to this problem, we can replace the variable turn with the following array.

boolean flag[2];

In this algorithm pi first set flag[i] to be true, signalling that it is ready to enter its critical section. Then pi checks to verify that process pj not also ready to enter its critical section. If pj were ready, then pi could wait until pj had indicated that it no longer needed to be in critical section. At this point pi could enter the critical section. On exiting the critical section, pi would set flag[i] to be false allowing the other processes to enter its critical section.

```
do {

    flag[i] = true;
    while (Flag[j]);

    critical section;

    flag[i] == false;

    remainder section;

} while (1);
```

In this solution, mutual exclusion requirement is satisfied, but progress requirement is not satisfied.

Algorithm

By combining the key ideas of both algorithm 1 & algorithm 2 we obtain a correct solution where all the 3 requirements are met, the process share two variables boolean flag[2], int turn.

```
do
{

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

    critical section;

    flag[i] = false;

    remaining section;

} while (1);
```

Synchronization in hardware.

Race conditions are prevented by requiring that critical regions be protected by locks i.e., the a process must acquire a lock before entering a critical section it releases the lock when it is exit the critical section.

```
do
{
    Acquire lock

    Critical Section;

    Release lock

} while (TRUE)
```

Unfortunately, this solution is not feasible in a multi-processor environment disabling interrupts on a multi processor can be a time consuming as the message is passed to all the processors. This message passing delays entry into each critical section & the system efficiency decreases.

The modern computers therefore uses a special hardware instruction TestAndSet() instruction and Swap() instruction.

TestAndSet()

The imp characteristics of TestAndSet() instruction is that it is executed automatically, if 2 TestAndSet() instructions are executed simultaneously they will be executed sequentially in some arbitrary order if the machine support TestAndSet() instr., then we can implement mutual exclusion by declaring a variable lock.

The swap() instr. in contrast to the TestAndSet() instruction operates on the content of two words.

Mutual Exclusion Implementation using TestAndSet() variable

```
do
{
    while ( TestAndSet (&lock))
        ;
    // critical section.
        lock = false ;
    // remainder section ;
} while (TRUE)
```

TestAndSet() Definition

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

Definition of Swap()

```
void Swap ( boolean *a, boolean *b)
{
    boolean temp = *a ;
    *a = *b ;
    *b = temp ;
}
```

swap Implementation.

```
do
{
  key = TRUE;
  while (key == TRUE)
   swap (&lock, &key);
    // critical section
    lock = false;
   // remaining section
}
```

Imp

Semaphore.

Semaphore is a synchronization tool.
A semaphore S is an integer variable that apart from initialization is accessed only through two standard atomic operations wait() and signal()
The wait() (decrement) operation also called P (to test) and the signal() called V (increment)

```
wait (S)                 signal(S)
{ while S<=0             {
    ;                        S++;
  S--;                   }.
}
```

Binary and counting semaphore

Semaphore is mainly of 2-types :
⇒ counting semaphore
⇒ binary semaphore (mutex lock)

The value of counting semaphore can range over an unrestricted domain but the value of a binary semaphore can range only b/w 0 and 1

The n processess share a semaphore (mutex) which is initialised to 1. The counting semaphore can be used to control access to a given resource consisting of a finite no. of instances.

Each process that wishes to use a resource perform a wait operation on the semaphore thereby decrementing the count. When a process releases a resource it performs a signal operation thereby incrementing the count. When the count for the semaphore goes to 0 all resources are being used after that processes that wish to use a resource will block until the count becomes greater than 0.

Eg Two concurrent executing processes P_1 and P_2. P_1 with a stmnt S_1 & P_2 with a stmnt S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this synchronization problem using semaphore as follows :

```
⇒ Synch = 0
   P1                 P2
   S1;                wait (synch)
   signal (synch)     S2;
```

Implementation

```
do
{
   wait (mutex);
   // critical section
     signal (mutex);
       // remainder section
} while (TRUE)
```

Classical Synchronization Problem.

Producer - consumer problem using bounded buffer

Producer

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

Consumer

```
do {
  wait(full);
  wait(mutex);
  ...
  // remove an item c
  from buffer.
  signal(mutex);
  signal(empty);
  // consume next item
} while (TRUE)
```

We assume that the pool consist of n buffers each cabable of holding one item. The mutex semaphore provides mutual exclusion for the accesses to the buffer pool and is initialised to value 1. The empty & full semaphore count the no. of empty & full buffers. The semaphore empty is initialized to a value n & the semaphore full is initialized to a value 0.

• Readers Writers Problem.

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want to only to read the database whereas other may want to update the database, we distinguish these type of processes by refering readers & writers. Obviously, if 2 readers access the shared data simultaneously now adverse affect occurs. However, if a writer & some other processes access the database simultaneously a problem may ensure. To ensure that these difficulties do not arise we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is refered as readers - writers problem.

Solution using semaphore.

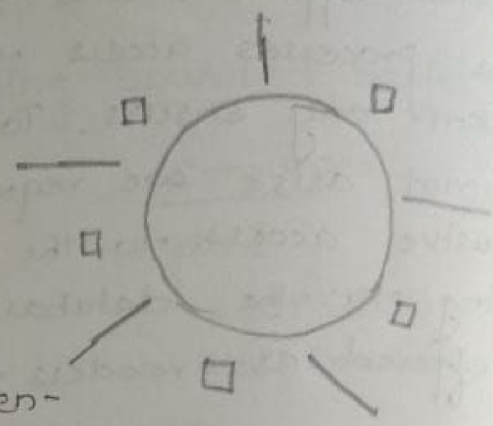
Reader

```
do {
wait(mutex);
readcount++;
if(readcount==1)
  wait(wrt);
  signal(mutex);
// reading occurs
  wait(mutex);
  readcount--;
  if(readcount==0)
    signal(wrt);
    signal(mutex);
} while (TRUE)
```

Writer

```
do {
  wait(wrt);
  ...
// writing is done
  ...
  signal(wrt);
} while(true)
```

Dining - philosophers problem.



Dining - philosopher problem is a classical synchronization problem. It is an example of large classes of concurrency problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free & a starvation-free manner. In this problem there are 5 philosophers whose spent their life thinking and eating. The philosophers share a circular table surrounded by 5 chairs each belonging to one philosopher & the table is laid with 5 single chopstick. When a philosopher thinks he does not interact with their neighbours from the time to time a philosopher gets hungry & tries to pick up the 2 chopstics that are closes to him. A philosopher may pick up only 1 chopstick at a time. When a hungry philosopher has both chopstick at the same time he eats without releasing his chopsticks. When he finish eating he puts down them & start thinking again.