OS before midsem  
Must read about MUTEX, there is much information in it.

Synchronous message passing and asynchronous message passing  
Synchronous signals and asynchronous signals in threading issues

Task queue in thread pools  
buffer queue in spooling

Multiprogrammed OS : **maximum CPU utilisation**

multiple processes are loaded into the main memory.

This ensures maximum CPU utilisation i.e whenever the current running process requires an i/o operation , CPU can switch to next process in ready queue.

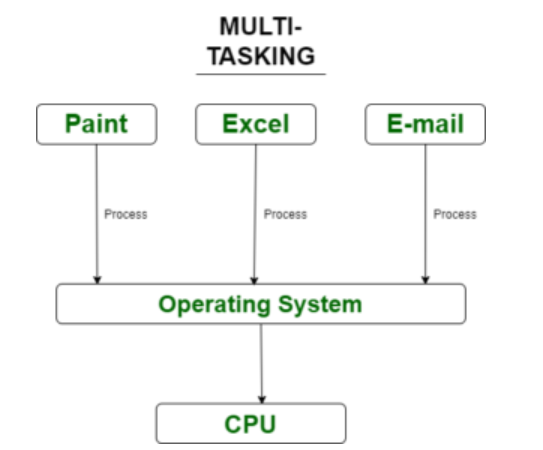
ex1:  
lets consider a process as a simple C code....  
During cin and cout i/o operations have to be done  
ex2:  
When process wants to read/access something from the secondary memory

Multitasking OS: **Responsiveness** (logical extension of multiprogramming)

Multiple processes appears to run simultaneously.

Here CPU rapidly switches the processes so that user feel that processes are running simultaneously

Here we define a time slice , i.e after each time slice CPU switches to another process

   
  
**Note:**A task in a multitasking system is not whole application program but it can refers to a “thread of execution” when one process is divided into sub-tasks.

User interface:

1.Command line interface :

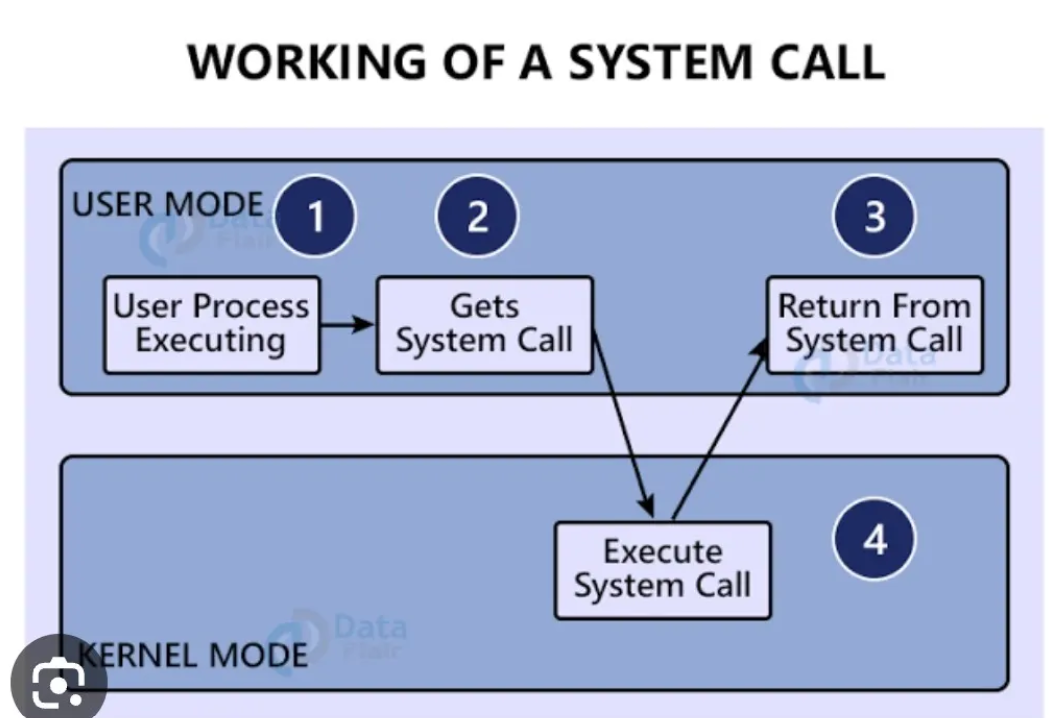
Interface between user and OS in which User have to enter the commands in command interpreter

2.Graphical user interface:

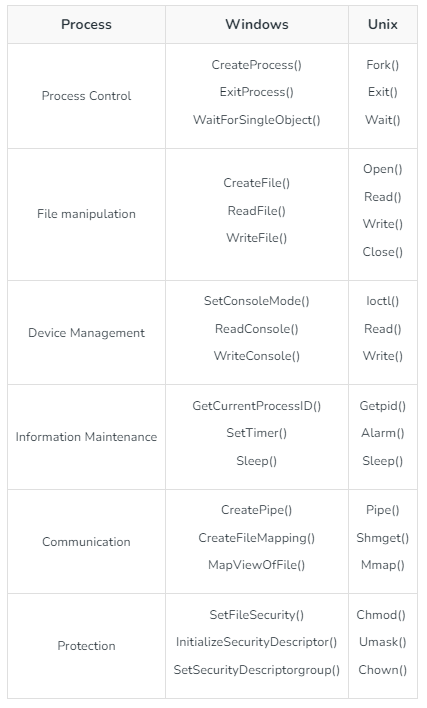
Interface between user and OS in a graphical way.

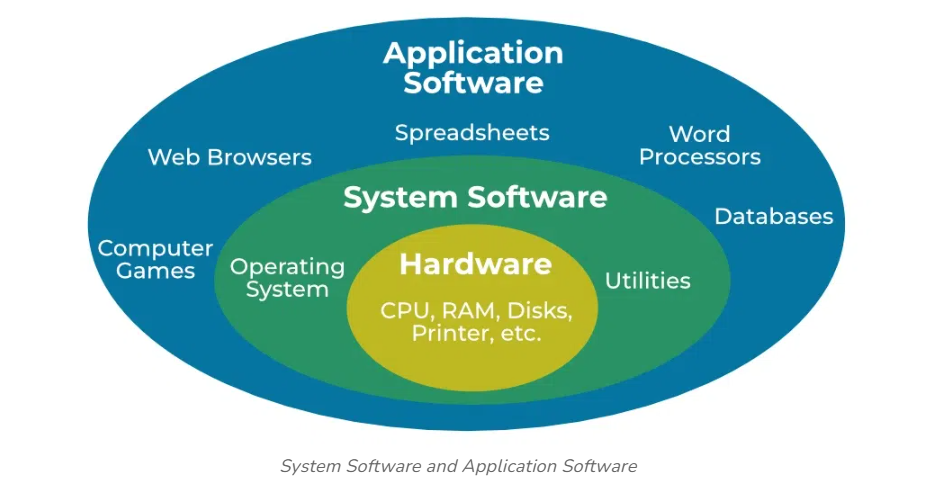
System call:

It is a programmatic way in which a program requests a service from the kernel of Operating system.(kernel - core of OS)



When a system call is made, the program is temporarily switched from user mode to [kernel](https://www.geeksforgeeks.org/kernel-in-operating-system/) mode.

  
Ex: wait, signal, fork, exec



Just see and leave

System programs:   
System programs are like backbone of the operating systems which ensures that our computer is running smoothly.

Examples are

a)**operating system kernels:** Ex:Windows, linux

b)**Device drivers:**   
Device drivers work as a simple translator for OS and devices . Basically it act as an intermediatory between the OS and devices and provide facility to both OS and devices to understand each other’s language so that they can work together efficiently   
 Ex: printer drivers, graphic card drivers

c)**File management tools:** Ex: File explorer in windows

d)**memory management software:**Makes allocation and deallocation of memory resources among different processes.

Application programs:

These are programs used by the end users to perform specific tasks.

Ex:

Word processors  
Web browsers   
Media players   
Games

**PROCESS MANAGEMENT:**

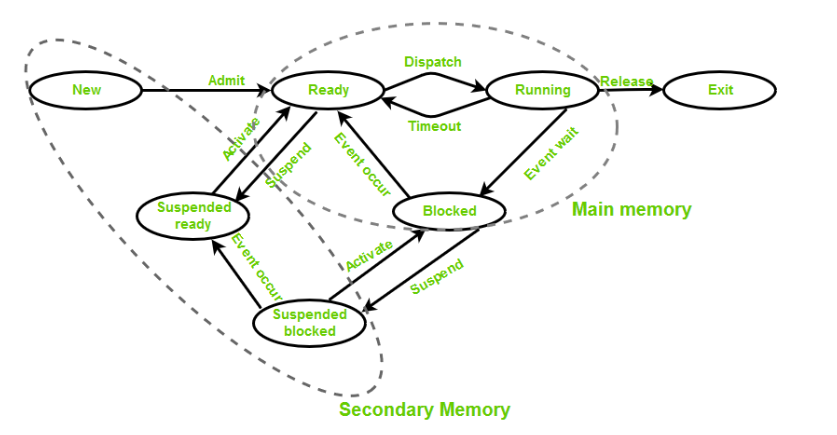
Process states:

NEW, READY, RUNNING, WAITING, TERMINATED = main states

I/O request example: during execution of a process if process needs to access a file in the secondary memory then process goes to waiting state for sometime  
  
**NEW**:   
the process is about to be created but not yet created. It is the program that is present in secondary memory that will be picked up by the OS to create the process.  
  
**READY**:  
necessary resources for the process are allocated   
process is ready to get executed and waiting inside the ready queue.  
When the CPU scheduler selects it, it get executed and goes to running state.  
  
**RUNNING**:  
CPU executes instructions of the process one by one.  
When an interrupt signal is received, process should stop executing and go back to ready queue.

**TERMINATED**:  
when process completes its execution then it will be terminated.  
resources are deallocated

**WAITING**: (spool)  
During the i/o operation, the running process has to go for waiting state.  
The process continues to wait in the main memory and does not require CPU.   
Once the I/O operation is completed the process goes to the ready state.

process go from RUNNING to WAITING during i/o operation – occurs naturally  
Process go from RUNNING to READY during interrupt signal – done forcefully  
  


**Suspend Ready:**Process that was initially in the ready state but was swapped out of main memory(refer to [Virtual Memory](https://www.geeksforgeeks.org/virtual-memory-in-operating-system/) topic) and placed onto external storage by the scheduler is said to be in suspend ready state. The process will transition back to a ready state whenever the process is again brought onto the main memory.

**Suspend Wait or Suspend Blocked:**Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.  
  
Here don’t concentrate on what will happen after completion of waiting state, suspended waiting state

Context switch: (also explain about PCBs)  
Mainly contain 2 steps.  
a) state save.  
b) state restore.  
When an interrupt occurs CPU has to save the context of current process so that it can be continued from where it was stopped after the completion of interrupted process.

Process control block(PCB):   
As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process’s execution status

It contains all necessary information about a process

a)process id

b)process state

c)program counter:

address of the next instruction in the process that is going to be executed.   
Because when an interruption occurs, after the interruption CPU can resume the process

d)Stack Pointer: The stack pointer points to the top of the stack in the system's memory. The stack is a region of memory used for temporary storage like local variables associated with the corresponding process.  
During context switching, the stack pointer in the PCB saves the address of the top of the process's stack. This allows the system to restore its local variables when it resumes execution.

e)Registers: When context switch occurs the current value of process specific registers would be stored in the PCB and the process would be swapped out. When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the register’s saved values in the PCB.

f)memory limits: information about page tables, segment tables

g)open files list:  
Maintains a list of files that the process has opened during its execution. This list ensures that there will not be any data corruption when multiple processes try to access a file

h)priority:  
a priority is assigned to the process for priority based scheduling algorithms.

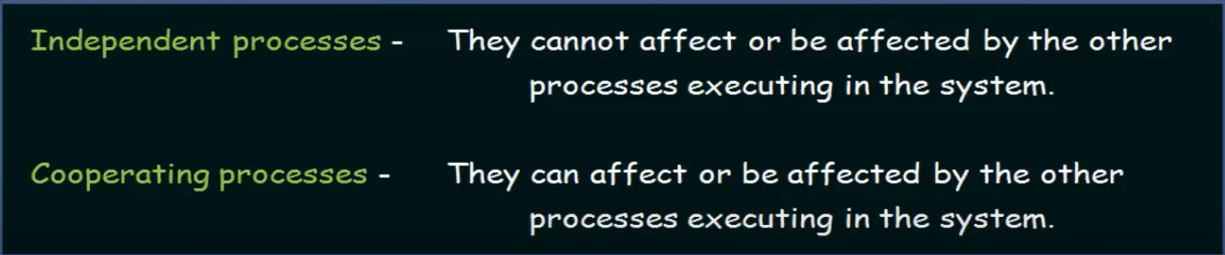
i)information about signals and signal handlers:  
Signals are the notifications sent by the operating system  
Signal handlers are functions that are executed when a specific signal is sent to a process.  
When a signal is delivered to the process the normal execution of the process is interrupted and then the corresponding signal handler is invoked

Once the signal handler completes its execution, the process resumes execution from where it was interrupted  
The signal handler execution involves operations like   
a)logging information i.e noting down about the event and the signal  
b)cleaning up resources allocated to the process(When signal is SIGTERM)  
c)modifying the data structures(process id, memory limits, open files list, priority, signal handlers) of the process.  
Examples:  
1. Event : request to terminate the process  
 signal : SIGTERM  
2. Event : user pressing ctrl+C to interrupt the process   
 signal : SIGINT (interrupt)  
terminate – naturally done when process completed its execution  
interrupt – forcefully stopping in the middle of execution

**Location of The Process Control Block**

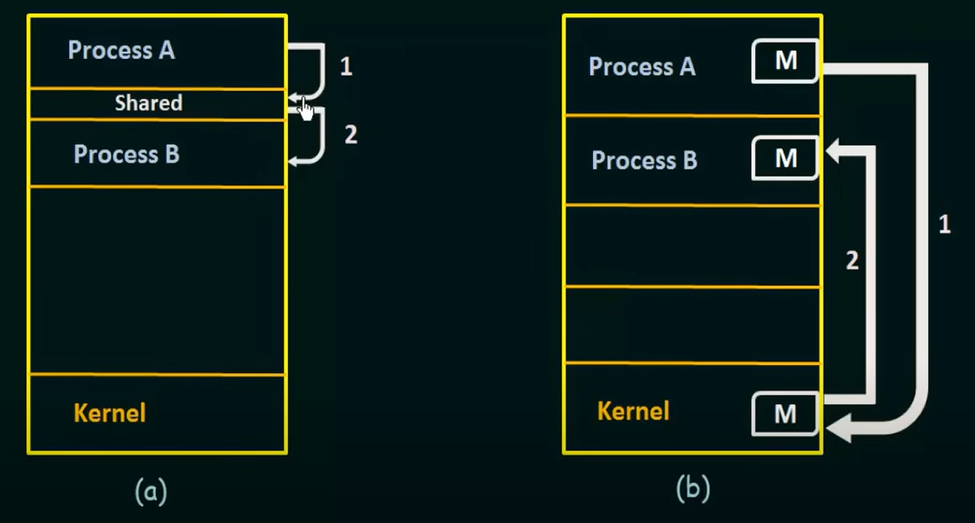
The Process Control Block (PCB) is stored in a special part of memory that normal users can’t access. This is because it holds important information about the process. Some operating systems place the PCB at the **start of the kernel stack** for the process, as this is a safe and secure spot.

Interprocess communication:

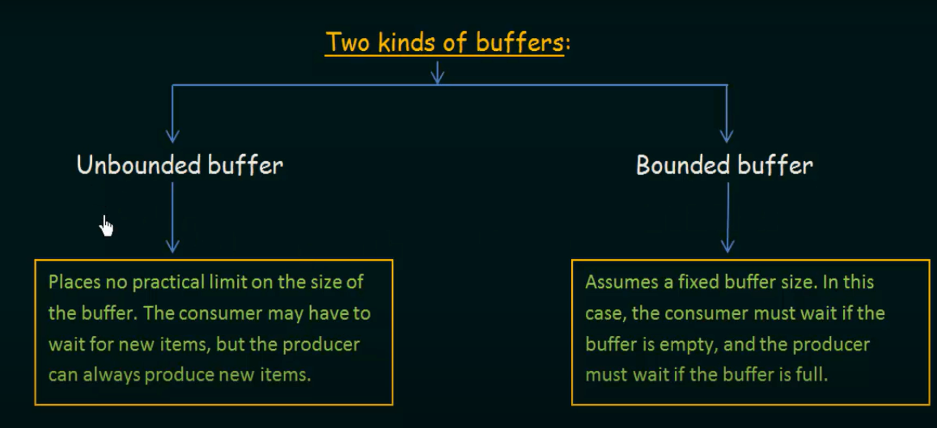


So communication is involved in Cooperating processes.

Why we have to allow processes to cooperate/communicate with each other?  
**Information sharing :**   
 2 process may share information among each other. For example producer consumer buffer  
**Computation speedup:**   
Diving a task into subtasks and all these subtasks are made to run concurrently. This can speedup the computation. And different subtasks are assigned to different processes and they need to communicate with each other

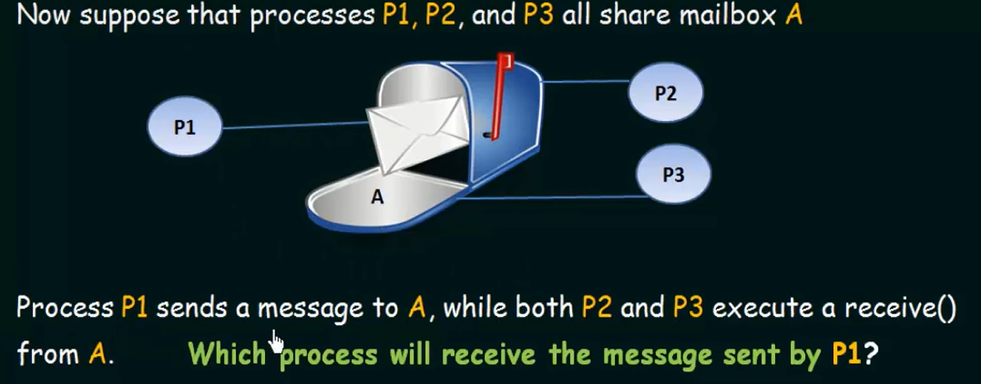


(a) -memory shared (b) - message passing (OS/kernel involvement is there)

1 Memory shared::   
processes shares common buffer   
Generally it is fast because it avoids data copying  


Ex: producer consumer problem (compiler produces assembly code which is produced by the assembler)  
Synchronization should be done properly.

2 message passing::   
Communication can be done with the involvement of kernel even though processes not have shared memory.  
So it is particularly useful in a distributed environment where processes resides on different computers connected by a network.

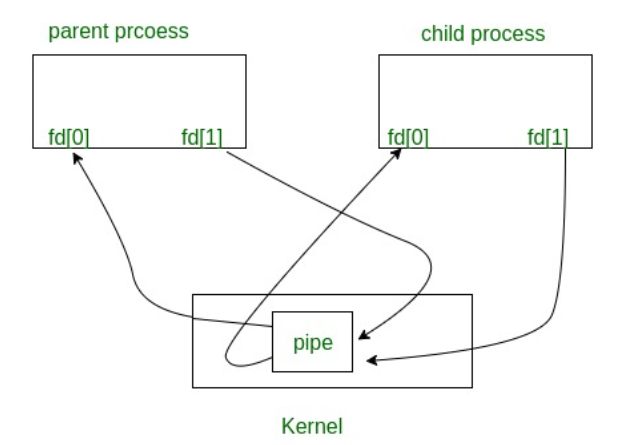
Here messages may be of fixed size or variable size.  
For fixed size – its easy for OS designer to make the design but difficult for a programmer to implement it and for variable size its vice versa(chuthiya point)  
  
A communication link must exist between 2 processes in order to communicate.  
Below are the ways to implement the links:  
  
**a)Direct and indirect communication:**Direct communication:  
Each process that wants to communicate must explicitly name the receiver or sender.   
 send(P, message) = send a message to process P  
 receive(Q, message) = receive a message from process Q  
  
Indirect communication:  
The messages are sent to and receive from mailboxes or ports  
Two processes can communicate only if they have a shared mailbox  
 send(A, message) = send a message to mailbox A  
 receive(A, message) = receive a message from mailbox A  
A mailbox may be owned either by the process or by the operating system  
  
We may define an algorithm to decide the priority to receive among process  
or allow at most one process at a time to execute receive( ) operation  
  
  
**b) Synchronous and Asynchronous message passing:**  
  
Synchronous message passing:  
Sender blocks until the receiver receives the message  
both sender and receiver are synchronized

Asynchronous message passing:  
sender continues its execution after sending message without waiting for receiver to receive.  
Requires additional mechanisms for error handling(message lost or duplicated) and synchronization as sender and receiver operates at different speeds.

**c) Buffering:**

When messages sent from sending process they temporarily wait inside a queue, if a particular message is already transmitting.  
So this queue can be specified as a buffer.

1.Zero capacity buffer:  
Queue has a maximum length zero. So the sender must block until receiver receives the message  
2.Bounded capacity buffer:  
Queue has some finite length. If the queue is not full, then sender can continue sending the messages. If queue is full sender must block until queue has some space  
3.Unbounded capacity buffer:  
sender never blocks  
  
  
3 Pipes:  
 Pipes are commonly used for communication between a parent process and its child processes.  
 1.The parent process creates a pipe using the **pipe()** system call  
 2.Then for the parent process 2 new file descriptors named pipefd[0] and pipefd[1] are generated   
 pipefd[0] – reading data from the pipe   
 pipefd[1] – writing data to the pipe  
 3.The parent process forks a child process using the **fork()** system call  
 4.The child is an exact copy of the parent including its file descriptors.   
 5.So child process also have pipefd[0] and pipefd[1] file descriptors which means both parent and child  
 is pointing to the same pipe.   
 6.For better communication, the parent and child processes close the file descriptors that they do not   
 need for communication. For example, the parent process may close the pipefd[0] descriptor and   
 the child process may close the pipefd[1] if we want communication from parent to child.  
 7.Now parent writes data to the pipe and child reads data from the pipe  
 8.After communication is complete both parent and child process should close the remaining file  
 descriptors associated with the pipe.



file descriptors:  
1)in Unix like OS; open files, sockets are identified with a unique numbers known as file descriptors  
2)Each process has its own set of file descriptors and the same file opened by different processes may have different file descriptor values but refers to the same file.  
3)When a process wants to read or write any file, corresponding file descriptors are sent as arguments to read( ) and write( )  
4)standard file descriptors:  
Standard Input (stdin): File descriptor 0, typically representing keyboard input.   
Standard Output (stdout): File descriptor 1, typically representing text output.   
Standard Error (stderr): File descriptor 2, typically used for error messages.   
5)It's important for processes to close file descriptors using close( ) system call when they're no longer needed

4 Sockets:

1.Sockets are defined as end points of the communication channel.  
2.When 2 processes want to communicate with each other, they each create a socket and use it to establish a  
connection.  
3.A socket is associated with a specific process   
4.A process is associated with multiple sockets for example a webserver process may create multiple sockets to handle each client.  
5.A socket is identified with an IP address and port number  
 IP address - says about the device in the network  
 Port number - says about the process in a device   
 So, communication can be established in between two processes of same device or across a network  
6.Socket provides bidirectional **FIFO** Communication facility over the network

There are 2 types of sockets  
a)Stream sockets(TCP - transmission control protocol):  
It is reliable because   
 - Before data transfer begins, a communication channel is established between processes and this connection is  
 maintained throughout the completion of data transfer  
 - It ensures that data is delivered in the correct order and without any loss through acknowledgement,   
 retransmission of lost data.  
Ex: 1.When you request anything in the browser it establishes a TCP connection with the web server, this ensures that data is delivered reliably.  
 2. High quality video or voice calls

b)Datagram sockets(UDP - user datagram protocol):  
It is unreliable because  
 - No connection is established and each datagram is sent independently (like sending through mailbox)  
 - It doesnt guarantee ordering delivery and no retransmissions  
Here benefit is low latency i.e **short amount of time for data transmissions**  
Ex: In video streaming, its okay if few frames are lost and the audio, video transmission time should be minimized. So here UDP connection is useful.

socket life cycle:

1.socket( ) - a socket is created and return a file descriptor representing the socket  
2.bind( ) - it is bound to specific IP address and port number  
3.connect( ) - ready to connect with another socket  
4.send( ) and receive( ) - sending and receiving the data  
5.close( ) - close the socket

5. Remote procedural call:

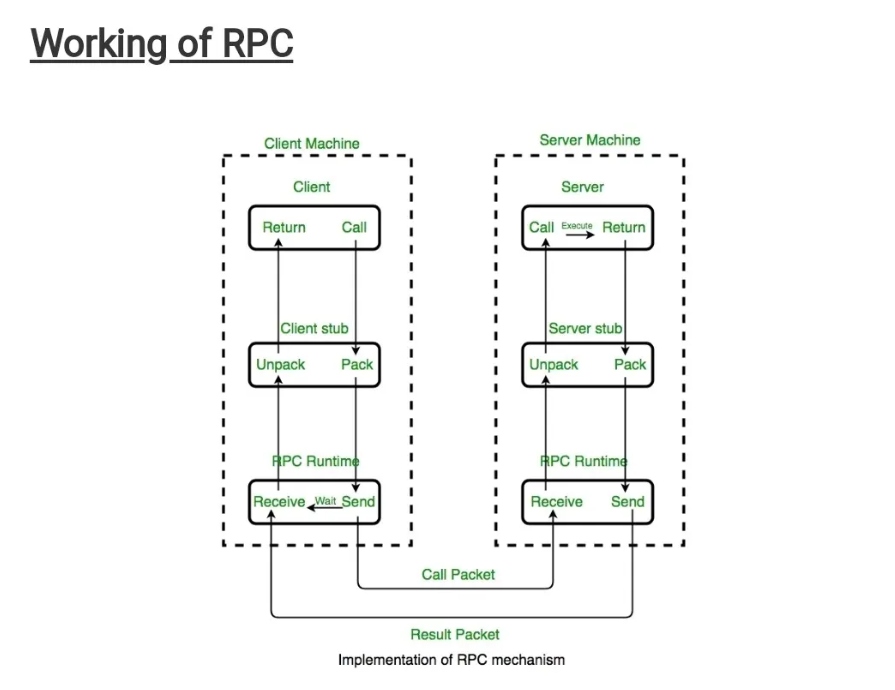
Basically remote procedural call means calling for a service in which server is present somewhere else.

Here client request a service from the server located in another computer without having to understand the network details. Ex: Google search

Includes mainly 5 elements:

1.client  
2.client stub   
3.RPC runtime   
4.server stub   
5.server

stub - piece of code used for packing and unpacking of parameters   
During RPC runtime, messages are transmitted in packed manner.



**CPU scheduling:**

Turnaround time: time between process creation and process termination

Throughput: no. of processes that are completed per unit time.

FCFS - non preemptive

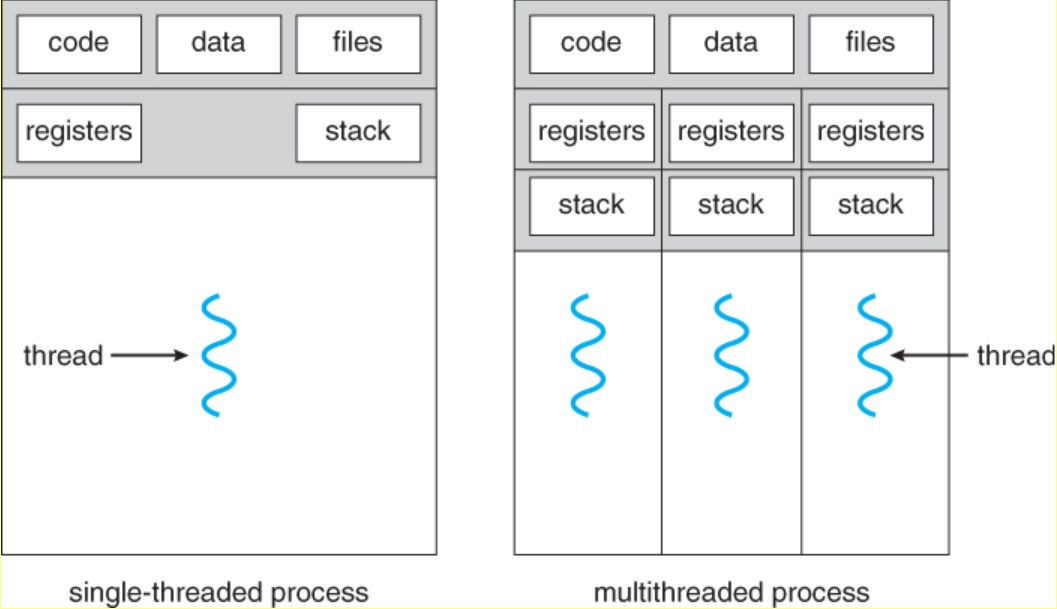
Shortest job scheduling first - can be preemptive or non preemptive

Priority scheduling - can be preemptive or non preemptive.

Round robin - same as FCFS but here there is a time quant so that if process burst time is more than time quant then it will be executed upto time quant time and remained part is added at the end of the ready queue.

Multilevel queue scheduling algorithm:   
Top levels(more priority) : interactive processes(generally use round Robin)   
Bottom levels(less priority) : batch processes(generally use FCFS)   
If a process takes too much CPU time it will be moves to lower priority level. Similarly if a process waits too long in a lower priority queue then it will be shifted to higher priority level

**Threads:** (user level threads)  
1.In a process, a thread is a single sequence of instructions being executed.  
2.Threads are **lightweight**. Here lightweight indicates that they share the same memory and resources with other threads within a process. This means creating and managing threads is easier than managing separate processes.  
Eg: Context switching is easier for threads than processes  
Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.



Like the processes, threads also have states like NEW, READY, RUNNING, TERMINATED, WAITING etc.  
Each thread has its own [Thread Control Block (TCB)](https://www.geeksforgeeks.org/thread-control-block-in-operating-system). Like the process, a context switch occurs for the thread, and register contents are saved in (TCB).  Here context switching between threads of same process.  
Priority can be assigned to the threads just like the process, and the highest priority thread is scheduled first.  
As threads share the same address space and resources, synchronization is also required for the various activities of the thread.

Types of threads:  
**User level threads and kernel level threads:**

1.ULT are implemented by users and implementation is easy.  
KLT are implemented by OS and implementation is complicated

2.In user-level threads, each thread has its own stack and registers, but they share the same address space.  
In kernel-level threads have their own stacks and their own separate address spaces, so they are independent and isolated from each other.

3.ULT: limited parallelism  
User-level threads are threads that are managed entirely by the application, without the operating system's direct involvement. They are "invisible" to the OS kernel. Since the OS only sees a single process, it doesn’t know there are multiple threads inside it. This means the OS cannot schedule those threads on multiple CPU cores. As a result, even if you have a multicore processor (multiprocessing capability), the OS can only assign one core to the entire process. The multiple threads inside the process have to share that one core.  
KLT: can be multithreaded effectively   
Kernel-level threads are managed by the operating system. The OS is fully aware of each thread within a process.  
Because the OS knows about the threads, it can schedule them across multiple CPU cores.

4. ULT - A blocking operation(like waiting for input/output) in one thread can halt the entire process -   
because as discussed earlier the OS only sees a single process, it doesn’t know there are multiple threads inside it. When one of the user-level threads performs a blocking operation, the entire process appears to be blocked to the OS. It doesn’t know that other threads inside the process might still be able to continue. As a result, it suspends the entire process, halting all other threads within that process too.  
If one kernel thread performs a blocking operation then other parallelly executing threads within the process can continue execution as they works independently.

5.context switch time = ULT < KLT  
Here context switch means switching of threads execution within a process. Time is more for KLT because OS support is needed and also ULTs share some common properties  
 context switch time = thread < process   
because state save and state restore in thread is relatively faster because threads within a process have some common properties. (Lightweight property)

6. difficulty in creating and managing = ULT < KLT  
because OS support is needed  
 difficulty in creating and managing = thread < process  
because threads are lightweight(have some common properties)

**Multithreading**:   
The idea is to achieve parallelism by dividing a process into multiple threads.  
Each thread is assigned to a CPU core for execution.  
Even though if processor can do only one task at a time, process is divided into multiple threads and context switch among them happens very rapidly so user feels that threads are running simultaneously.

Ex:MS Word uses multiple threads:  
 one thread to take inputs  
 one thread for formatting (adjusting font style, size, colour, and effects (bold, italics, underline))  
 one thread to check the grammatical errors  
 one thread for auto saving

1.Performance:  
Executing multiple threads concurrently, it can take advantage of parallelism and reduce overall execution time.

2.Responsiveness:  
Multiple tasks will be running concurrently.

**Threading issues:**

1. fork() and exec():

fork()-duplicating the process and new child will be created

When a particular thread called fork() there can be 2 cases, first one is duplicate all the threads in process and second one is to duplicate only that thread and remaining space is empty in the new process.

exec()-replace all the information with another information. Process ID remains same.

Solution is if exec() will be called just after fork() then duplicating that particular thread is enough because here no usage of other threads and in all other cases it is necessary to duplicate all the threads.

2. Thread cancellation:

Termination of thread during it's execution  
A thread that is to be cancelled is known as target thread

Ex1: when multiple number of threads are searching for an information in a database if one of the thread got it then all other threads that are searching have to be terminated.  
Ex2:when we opened some web browser in which something is loading and then we clicked into button on the top then all the threads that helps in loading have to be terminated.

Where the difficulty in cancellation lies?  
Resources have been allocated to a cancelled thread  
A thread is cancelled in the midst of updating data it is sharing with other threads.

Asynchronous cancellation: One thread terminates other thread  
Deferred cancellation: The target thread periodically checks whether it should terminate and gets terminated by   
 itself.

With deferred cancellation:   
It ensures that cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not

3. Signal handling: (try to explain completely about signal handlers as already discussed Above)

Signals are notifications sent to a process when a particular event occurs (e.g., division by zero, a keyboard interrupt).

**Signal handling:**  
Signal is generated by particular event  
Signal is sent to a process (issue occurs here)  
Signal is handled by signal handler

In multithreaded programs, determining to which thread the signal could be sent  
1. The signal could be sent to the **thread that caused the signal** (synchronous signals like division by zero).  
2. The signal could be sent to a **specific thread** that is designed to handle signals.  
3. The signal could be sent to **every thread** in the process.

**Synchronous signals**: signal is generated by current process event  
Ex: Suppose your program tries to divide a number by zero. This causes a **synchronous signal** (like **SIGFPE** for floating-point exceptions).  
**Asynchronous signals**: signal is generated by external event  
While running a program in the terminal, you press Ctrl+C to stop it. This generates an **asynchronous signal** (like **SIGINT** for interrupt).

Synchronous signals are sent to the **thread that caused the event** because that thread is directly responsible for it.  
Asynchronous signals, are not tied to any specific thread, so it is unclear which thread should handle them. In some systems, the signal might be sent to all threads or to a specific thread designed to handle signals.

Default signal handler: The operating system provides default behaviors (e.g., terminating the process).  
User defined signal handlers: The programmer can override the default behavior and define custom actions to handle the signal.

In UNIX, threads can choose whether they want to handle signals or not ie they can mask(ignore) the signal if they want. In windows, An APC is assigned to a specific thread for handling the signal. (APC = Asynchronous Procedure Calls)

4.thread pools: (declaring variables outside a for loop is better than declaring inside)

for each incoming request to server, creating a new independent thread(kernel level thread) is inefficient because:  
**1.Overhead of Thread Creation and Destruction**:  
**2.Resource Exhaustion**: Each thread consumes resources like CPU and memory. If there are too many threads, the   
 server might run out of these resources.

Some finite number of threads are created at the start of the program and remain idle until a task is assigned.  
They are not destroyed after they complete a task. Instead, they are returned to the pool to be **reused** for new tasks.  
Above mentioned pre created threads are known as thread pools

When there are more requests than available threads in the pool, the requests are added to a **task** **queue**. Threads pick up tasks from this queue when they become available.

5.Thread specified data:

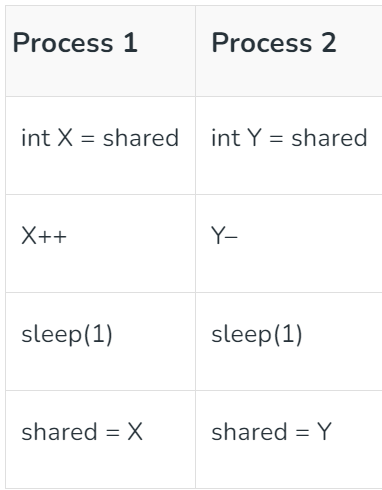
Thread-specific data means that each thread has its own private copy of certain data.   
It is particularly useful in thread pools where threads are reused because we cant predict what exact data is needed by a thread  
This ensures that each thread can work independently without worrying about shared data being modified by other threads

firstly discuss about what is critical section, conditions to satisfy by the solution of critical section problem and solution to critical section problem is nothing but process synchronization methods.   
Process synchronization methods also avoids the race condition and say about race condition.  
**Process Synchronization:**

The main objective of process synchronization is to ensure that multiple processes access share resources in a consistent manner.

It should be able to eliminate the race condition.

Synchronization Techniques made with help ofMutex, Semaphores, Monitors, Barriers**.**   
(solutions to critical section problem)  
  
  
**Race condition:**When multiple processes are accessing a common shared resources, the final output depends on the particular order in which access takes place.   
In general resources can be anything like printer, open files etc…

****Lets consider P1 access the shared memory first  
 P1 stores the value 10 in its local variable X (so X = 10).  
 Then X becomes X+1 ie 11  
 P1 goes into waiting state and CPU switches to Process P2  
 P2 stores the value 10 in its local variable Y (so Y = 10).  
 Then Y becomes Y-1 ie 9  
 P2 goes into waiting state and CPU switches to Process P1  
 Then value of shared becomes 11  
 P1 gets terminated and P2 gets executed  
 Then final value of shared member is 9  
  
If we consider P2 access the shared memory first then final value of shared member is 11proper process synchronization can prevent race conditions.

**Critical section: (From here main matter)**The **critical section** is a part of the code where shared resources, like variables or data, are accessed or modified by multiple processes.  
So don’t think like critical section as shared variables, it’s an area where modifications of shared resources took place  
For example in producer consumer buffer:  
shared variables = mutex, full, empty (semaphores)Shared resources = buffer  
critical region = /\* adds data to the buffer \*/  
In dining philosophers problem:  
shared variables = chopsticks (semaphores)  
critical region = /\*eating\*/

Since many processes tries to access the critical section at the same time, the **critical section problem** is about finding a way to control access, so the data remains consistent and no conflicts arise.  
solutions for critical section problem = process synchronization methods

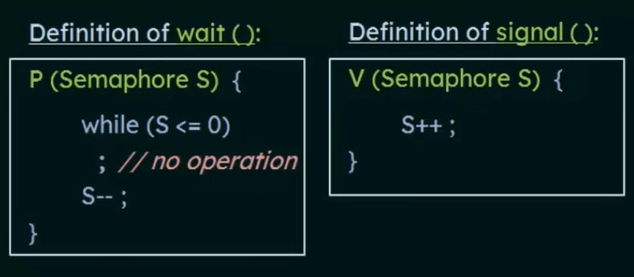
Any solution to the critical section problem must satisfy three requirements:  
a)**Mutually exclusive**: at an instant only single process can access critical section.

b)**Progress**: let's consider there is no process inside critical section and 2 processes P1 and P2 are competing to enter into critical section. Then it should not happen such that P1 is blocking P2 to enter inside CS and P2 is blocking P1 to enter inside CS. Because if it happens then there is no progress.   
(Dining philosopher solution using semaphore have a chance of no progress)

c)**Bounded wait**: there should be a limit for the no. of times a process can access the critical section because if there is no limit then there is a chance that a particular process is accessing critical section infinity number of times continuously and other processes are not getting chance.

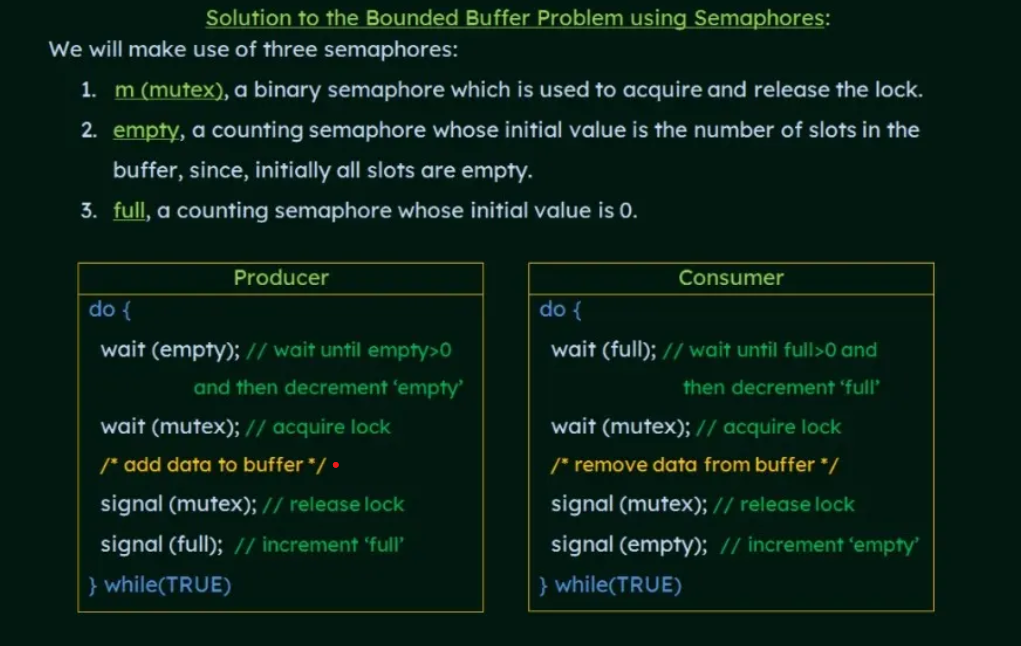
with help ofMutex, Semaphores, Monitors, Barriers, Petersons algorithmwe will find solution to critical section problems like producer consumer, dining philosopher, reader writers problem(not understood)

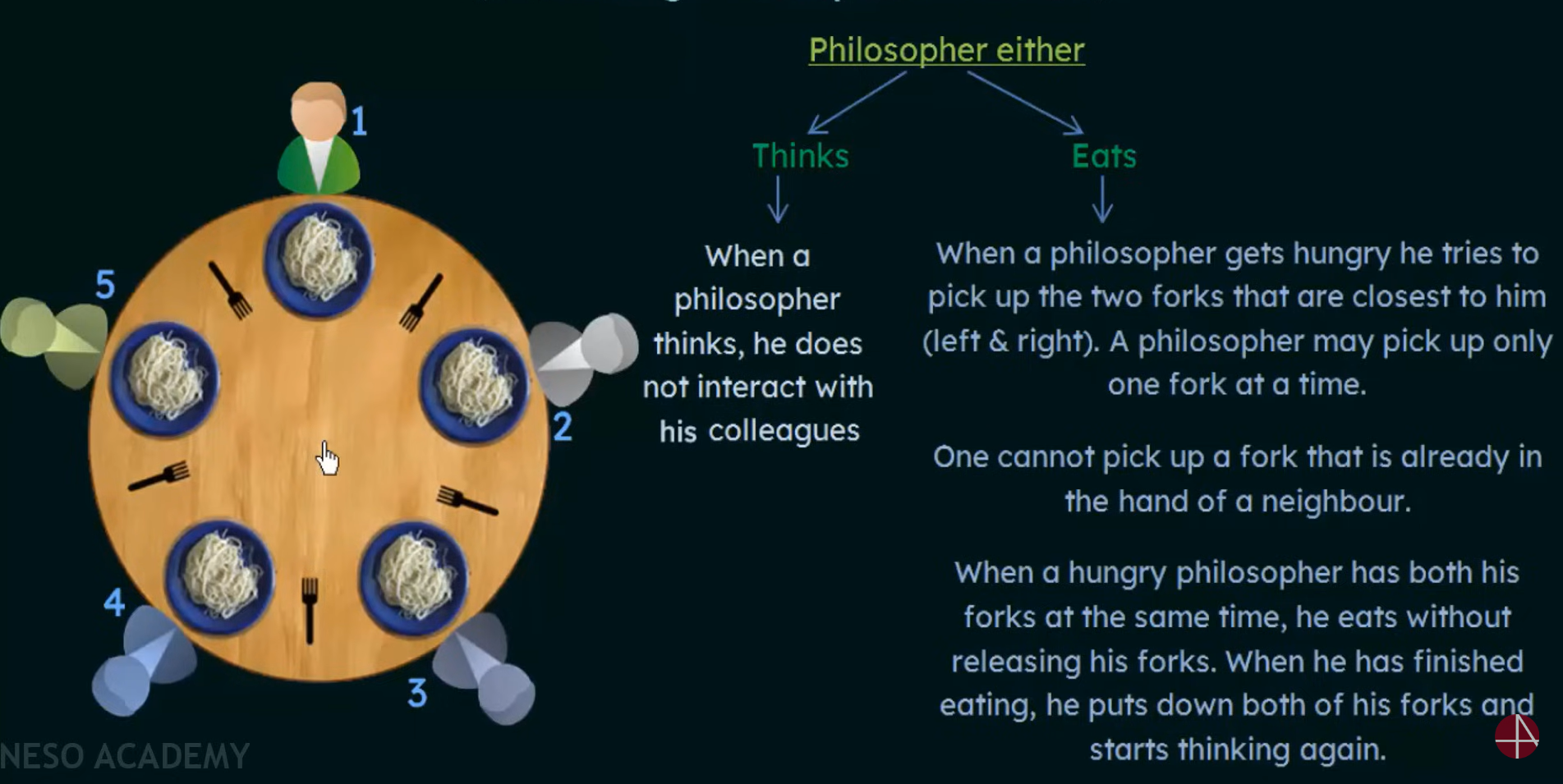
**Semaphore:**  
Semaphore is a non -ve variable shared between processes.  
This variable is used to solve the critical section problem and to achieve process synchronization

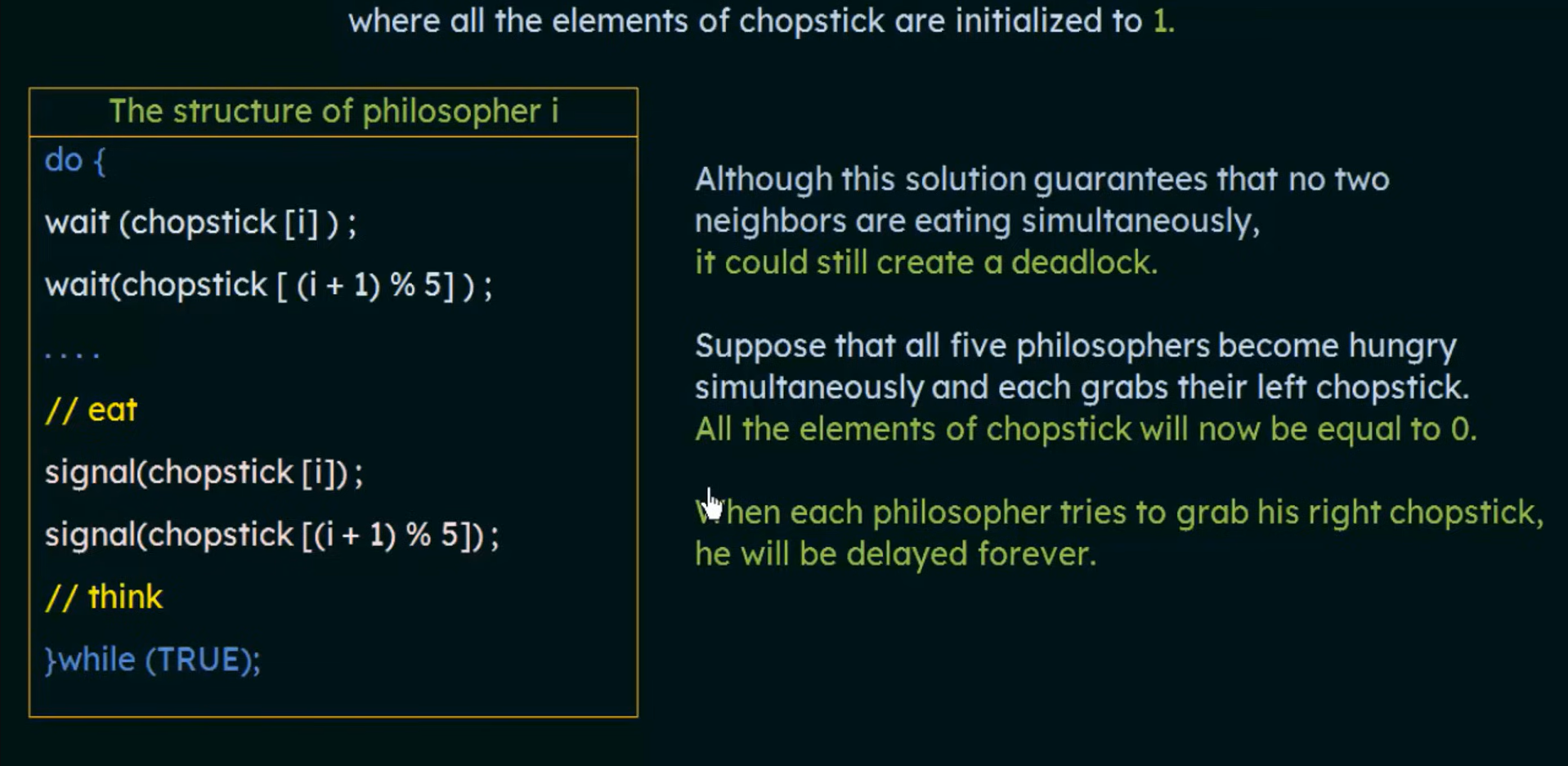
A semaphore S is accessed through 2 standard operations  
they are wait( ) and signal( )  


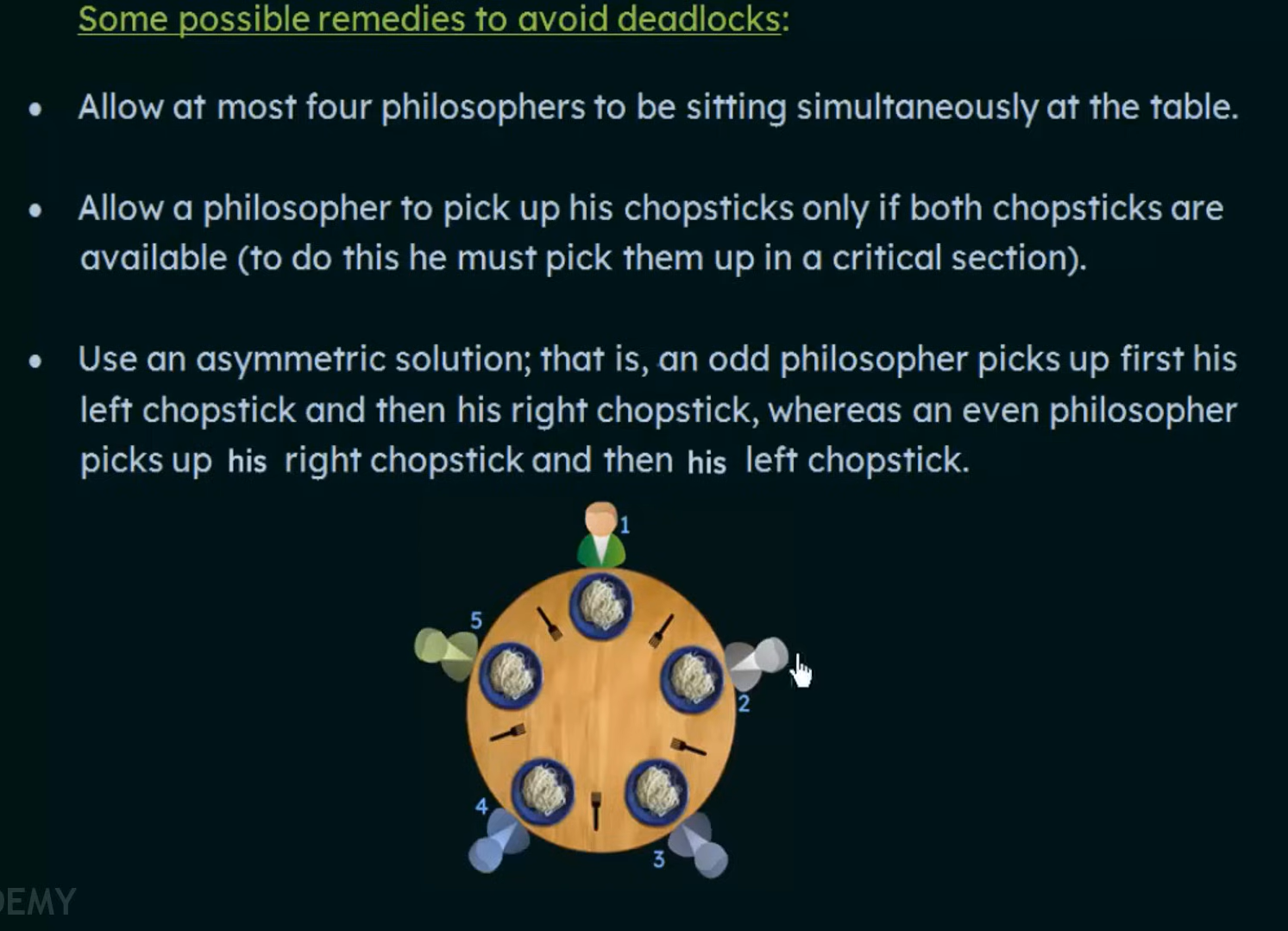
Binary semaphore : either 0 or 1. We can use it as a lock for the process.  
Counting semaphore : can be any non -ve integer. checks whether instance of a resource is available or not

Producer consumer buffer using semaphores :



Dining philosopher using semaphores:  


  
wait(chopstick[i]); line of all processes got executed. Then it leads to deadlock



Allow a philosopher to pick up his chopsticks only if both chopsticks are available:  
  
while(chopstick[i]<=0 || chopstick[(i+1)%5]<=0);  
chopstick[i]--;  
chopstick[(i+1)%5]--;  
//eat  
chopstick[i]++;  
chopstick[(i+1)%5]++;

Do questions in neso academy  
in problems for checking mutual exclusion first assume a particular process to be in critical section and note down the variable values for that process to be in critical section. And with these noted values check whether another process can enter into critical section or not  
Topics left in neso( petersons solution, readers writers problem, monitors, dining philosophers using monitors)

**Binary semaphore vs mutex(mutual exclusion):**

BS - works under signalling mechanisms  
Here signalling mechanism means, when some process performs an event such as releasing a resource or completing a task, it can signal this event by changing the value of the semaphore  
Mutex - works under locking mechanism

BS - The process which is having higher priority than current process can also release binary semaphore and take the lock.  
Mutex - The current process inside critical section only can release Mutex when it exits from critical section.

BS - Binary semaphore have no ownership.  
Mutex - There is ownership associated with mutex because only owner can release the lock.

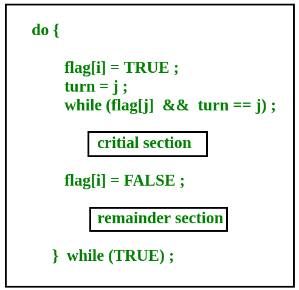
BS - value is changed according to wait () and signal () operations.  
Mutex - Mutex values can be modified just as locked or unlocked.

Petersons solution: (a synchronization technique like semaphore)  
Its limited to 2 processes  
It may not suits for modern computers

It is a **humble** algorithm ie one process tries to give chance to other process

flag[i] = true means Pi wants to enter inside critical section  
turn = j indicating whose turn to enter inside CS

For Pi: (somewhat difficult to memorize)  
When a process wants to enter inside critical section it gives chance or turn to other process  
It can enter inside CS only when other doesn’t want to enter



**Hardware Solutions : Test and set, swap**

LOCK:

while(lock==1) ;  
lock = 1  
{

//Critical section

}

lock = 0

Here there is no guarantee for mutual exclusion. Because lets consider P1 process is done with line 1 and not yet entered into critical section. At this time a process P2 can execute it's line 1 and 2 and enter into critical section and then P1 also can enter into critical section.

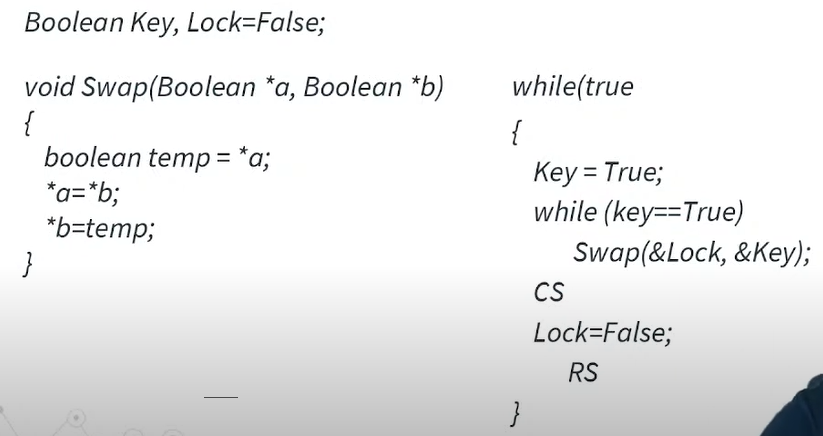
**Test and set:**

In order to avoid limitation in lock, line 1 and line 2 are combined in test and set.

While( Test and set(&lock) );   
{   
 //critical section  
}  
lock = 0

Boolean Test and set(boolean \*target)   
{   
 Boolean r = \*target  
 \*target = TRUE   
 return r  
}

**Swap:**

****

If lock is true, then key will never become false and therefore while loop wont breaks  
Internal working is almost similar to test and lock

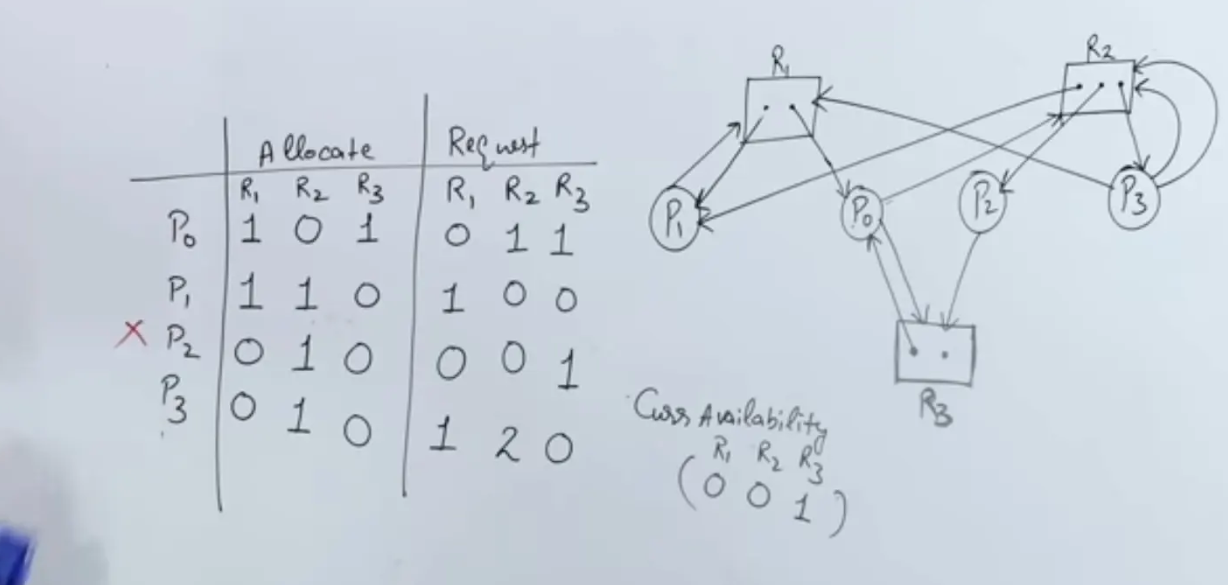
If lock is false, then after swapping key will be false and while loop breaks.. meanwhile lock becomes true

Deadlock:

Deadlock is like a locked state.

Let's consider 2 processes A and B. For process A to execute it will wait for process B to complete for getting required resources. For process B to execute it will wait for process A to complete for getting required resources. So both processes get stucked.   
  
Deadlock happens when all of these conditions are present simultaneously in a system:  
Mutually exclusive:  
At an instant, an instance of a resource can be allocated to single process.  
Non preemptive:  
After allocating an instance of a resource to a process, it will be released only after process completes its execution  
Hold and wait:  
Process is holding some resources and waiting for some other resources  
Circular wait:  


Bankers algorithm(used to detect the deadlock)



Methods for deadlock handling:  
 Deadlock Prevention or Avoidance  
 Deadlock Recovery  
 Deadlock Ignorance

**Deadlock prevention:**In this we prevent at least one of the four necessary conditions that lead to deadlock

**Mutual exclusion**:  
The idea here is that mutual exclusion should be applied only to resources that **cannot** be shared, such as printers or other physical devices. However, if a resource can be shared (like a read-only file), then it does not need a lock.   
So this may avoid deadlock up to some extent.  
**Problem**: Deadlock cannot be avoided completely as some resources can not be shared

**Non preemptive:**If a process is holding some resource and requesting other resources which are already allocated to some other processes. Then resources that current process is holding are released/preempted. After some time process again request for the previously allocated resources and extra required resources. **Problem: LIVE LOCK PROBLEM**Even though processes are changing their states continuously expecting some progress, still there is no progress  
according to above method- Both p1 and p2 detect that they can’t acquire second resource, so they release resource that they are holding and then try after some time.Again after some time p1 again acquired r1 and requesting to r2 p2 again acquired r2 and requesting to r1 so there is no overall progress still process are changing their state**.**

**Hold and wait:**To neglet Hold and wait, we have to make sure that holding and waiting for a process shouldn’t occurs at same time  
So we allocate required resources for the process before even starting execution.  
**Problem**: Some process wont be able to get resources and this leads to starvation

**Circular wait:**  
To remove the circular wait in system we can give the ordering of resources for any process to acquire.  
  
we can fix the resource acquiring order like the process first need to acquire resource r1 and then resource r2.  
So R2 cannot be directly allocated to P2

These are the Deadlock prevention methods but practically only fourth method is used as all other three condition removal method have some disadvantages with them.

**Deadlock recovery: Not studied seems difficult and useless**Manual Intervention - inform the operator and let them handle Automatic Recovery   
Process TerminationResource Preemption

**Deadlock ignorance:**If a deadlock is very rare, then let it happen and reboot the system. Why to create overhead on handling it if it occurs very rarely.  
Deadlock ignorance performance is better than the above two methods but the correctness of data is not there.

For single instance resource

If cycle is there then deadlock

If cycle is not there then no deadlock

If a process waits for finite time to get executed - starvation

If a process waits for infinite time to get executed - Deadlock.

Given 8 instances of resource and each process requires 3 instances to execute  
At max how many processes can be executed without deadlock.  
**Sol:**Lets first find out where deadlock starts occuring  
Firstly assign (3-1) instances to each process (each process is deficit of 1 instance)  
No. of processes = 8/(3-1) = 4  
So deadlock starts from 4 processes  
So at max (4-1) processes can be executed without deadlock.  
[ N/(m-1) - 1 ]