General Information

Monolithic kernel

- Kernel size increases because kernel + kernel subsystems compiled as single binary
- Difficult to extension or bug fixing,
- Need to compile entire source code.
- Bad maintainability
- Faster, run as single binary
- Communication between services is faster.
- No crash recovery.
- More secure

Eg: Windows, Linux etc

Microlithic kernel

- Kernel size is small because kernel subsystems run as separate binaries.
- Easily extensible and bug fixing.
- Easily recover from crash
- Slower due to complex message passing between services
- Process management is complex
- Communication is slow
- Easily recoverable from crashing

Eg: MacOS, WinNT

RTOS:

-RTOS is an operating system that guarantees a certain capabilities within a specified time constraint.
-RTOS must also must able to respond predictably to unpredictable events.

Application Programming:

Compilation Stages

●Preprocessor
– Expands header files
– Substitute all macros
– Remove all comments
– Expands and all #directives

●Compilation
–Convert to assembly level instructions

●Assembly
– Convert to machine level instructions
– Commonly called as object files
– Create logical address

●Linking
– Linking with libraries and other object files

Linking - static:

-Static linking is the process of copying all library modules used in the program into the final executable image. This is performed by the linker and it is done as the last step of the compilation process.
-To create a static library first create intermediate object files. Eg: gcc -c fun1.c fun2.c.
⇒ Creates two object files fun1.o and fun2.o
-Then create a library by archive command: ar rcs libfun.a fun1.o fun2.o

Linking - Dynamic

-It performs the linking process when programs are executed in the system. During dynamic linking the name of the shared library is placed in the final executable file. Actual linking takes place at run time when both executable file and library are placed in the memory.
- The main advantage to using dynamically linked libraries is that the size of executable programs is reduced. To create a dynamic library (shared object file)

Eg: gcc -fPIC -shared fun1.c fun2.c -o libfun.so

Common errors with various memory segments:

-Stack Overflow / Stack Smashing:
●When the process stack limit is over. Eg: Call a recursive function infinite times.
● When you are trying to access an array beyond limits. Eg int arr[5]; arr[100];

-Memory Leak:
●When you never free memory after allocating. Eventually process heap memory will run-out.

-Segmentation Fault:

●When you try to change text segment, which is a read-only memory or try trying to access a memory beyond process memory limit (like NULL pointer)

Linux Components:

●Hardware Controllers: This subsystem is comprised of all the possible physical devices in a Linux installation - CPU, memory hardware, hard disks

●Linux Kernel: The kernel abstracts and mediates access to the hardware resources, including the CPU. A kernel is the core of the operating system

●O/S Services: These are services that are typically considered part of the operating system (e.g. windowing system, command shell)

●User Applications: The set of applications in use on a particular Linux system (e.g. web browser)

Linux Kernel Subsystem

● Process Scheduler (SCHED):

– To provide control, fair access of CPU to process, while interacting with HW on time

● Memory Manager (MM):

– To access system memory securely and efficiently by multiple processes. Supports Virtual Memory in case of huge memory requirement

● Virtual File System (VFS):

– Abstracts the details of the variety of hardware devices by presenting a common file interface to all devices

● Network Interface (NET):

– provides access to several networking standards and a variety of network hardware

● Inter Process Communications (IPC):

– supports several mechanisms for process-to- process communication on a single Linux system

Virtual File System:

- Presents the user with a unified interface, via the file- related system calls.

- The VFS interacts with file-systems which interact with the buffer cache, page-cache and block devices.

- Finally, the VFS supplies data structures such as the dcache, inodes cache and open files tables.

- *– Allocate a free file descriptor.*
- *– Try to open the file.*
- *– On success, put the new 'struct file' in the fd table of the process.On error, free the allocated file descriptor.*

System call:

- Communications are two types: Synchronous(Polling) & Asynchronous(Interrupts)

Interrupts:

Hardware: Generates whenever a Hardware change happens.

Software: Generates by instruction from code (eg: INT 0x80).

●Interrupt controller signals CPU that interrupt has occurred, passes interrupt number

● Basic program state saved

● Uses interrupt number to determine which handler to start

● CPU jumps to interrupt handler

● When interrupt done, program state reloaded and program resumes

systems call:

A set of interfaces to interact with hardware devices such as the CPU, disks, and printers.

Advantages:

– Freeing users from studying low-level programming

– It greatly increases system security

– These interfaces make programs more portable

⇒ For a OS programmer, calling a system call is no different from a normal function call. But the way system call is executed is way different.

⇒ Logically the system call and regular interrupt follow the same flow of steps. The source (I/O device v/s user program) is very different for both of them. Since system call is generated by user program they are called as 'Soft interrupts' or 'Traps'

System Call vs Library Function

-A library function is an ordinary function that resides in a library external to your program. A call to a library function is just like any other function call

-A system call is implemented in the Linux kernel and a special procedure is required in to transfer the control to the kernel

-Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ

System Call Example:

gettimeofday()

-Gets the system's wall-clock time. It takes a pointer to a struct timeval variable. This structure represents a time, in seconds, split into two fields.

- *tv_sec field - integral number of seconds*
- *tv_usec field - additional number of usecs*

nanosleep()

-A high-precision version of the standard UNIX sleep call. Instead of sleeping an integral number of seconds, nanosleep takes as its argument a pointer to a struct timespec object, which can express time to nanosecond precision.

- *tv_sec field - integral number of seconds*
- *tv_nsec field - additional number of nsecs*

Other Example: Read, Write, Open, Close, Exit

Process:

-Running instance of a program is called a PROCESS. When you invoke a command from a shell, the corresponding program is executed in a new process. The shell process resumes when that process complete

Process vs Program:

-A program is a passive entity, such as file containing a list of instructions stored on a disk

-Process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

-A program becomes a process when an executable file is loaded into main memory

⇒ Each Process will have its own Code, Data, Heap and Stack

Process State Transition Diagram

state:

New: The process is being created

Ready: The process is waiting to be assigned to processor

Running: Instructions are being executed

Wait: The process is waiting for some event to occur

Terminated: The process has finished execution

New --> Ready: admitted.

Ready --> Running: Schedule dispatch.

Running --> Ready: Interrupted.

Running --> Wait: I/O or event wait

Waiting --> Ready: I/O or event completion.

Running --> terminated: exit.

Descriptor - ID:

- Each process in a Linux system is identified by its unique process ID, sometimes referred to as PID. Process IDs are numbers that are assigned sequentially by Linux as new processes are created. Every process also has a parent process except the special init process.

- Processes in a Linux system can be thought of as arranged in a tree, with the init process at its root. The parent process ID or PPID, is simply the process ID of the process's parent.

Active Processes

The ps command displays the processes that are running on your system. By default, invoking ps displays the processes controlled by the terminal or terminal window in which ps is invoked

Context Switching

-Switching the CPU to another task requires saving the state of the old task and loading the saved state for the new task. The time wasted to switch from one task to another without any disturbance is called context switch or scheduling jitter. After scheduling the new process gets hold of the processor for its execution

Process Creation

Two common methods are used for creating new process:

- Using system(): Relatively simple but should be used sparingly because it is inefficient and has considerable security risks.

- Using fork() and exec(): More complex but provides greater flexibility, speed, and security.

-System(): It creates a sub-process running the standard shell. The system function in the standard C library is used to execute a command from within a program.
-fork(): fork makes a child process that is an exact copy of its parent process. When a program calls fork, a duplicate process, called the child process, is created. The parent process continues executing the program from the point that fork was called. The child process, too, executes the same program from the same place. All the statements after the call to fork will be executed twice, once, by the parent process and once by the child process.
- **fork() - How to Distinguish?**
First, the child process is a new process and therefore has a new process ID, distinct from its parent's process ID. One way for a program to distinguish whether it's in the parent process or the child process is to call getpid. The fork function provides different return values to the parent and child processes. One process "goes in" to the fork call, and two processes "come out," with different return values. The return value in the parent process is the process ID of the child. The return value in the child process is zero.

Process Zombie:
-Zombie process is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children.
-When a program exits, its children are inherited by a special process, the init program, which always runs with process ID of 1 (it's the first process started when Linux boots). The init process automatically cleans up any zombie child processes that it inherits.

Process Orphan:
-An orphan process is a computer process whose parent process has finished or terminated, though it remains running itself. Orphaned children are immediately "adopted" by init .
-An orphan is just a process. It will use whatever resources it uses. It is reasonable to say that it is not an "orphan" at all since it has a parent but "adopted".
-Init automatically reaps its children (adopted or otherwise). So if you exit without cleaning up your children, then they will not become zombies.

Process Overlay - exec()
-The exec functions replace the program running in a process with another program.
- When a program calls an exec function, that process immediately ceases executing and begins executing a new program from the beginning. Because exec replaces the calling program with another one, it never returns unless an error occurs.
- This new process has the same PID as the original process, not only the PID but also the parent process ID, current directory, and file descriptor tables (if any are open) also remain the same. Unlike fork, exec results in still having a single process

Process Blending fork() and exec():
-If we want a calling program to continue execution after exec, then we should first fork() a program and then exec the subprogram in the child process.
-This allows the calling program to continue execution as a parent, while the child program uses exec() and proceeds to completion.
-This way both fork() and exec() can be used together.

Process COW – Copy on Write:
- Copy-on-write (called COW) is an optimization strategy.
- When multiple separate processes use the same copy of the same information it is not necessary to re-create it. Instead they can all be given pointers to the same resource, thereby effectively using the resources. However, when a local copy has been modified (i.e. write) , the COW has to replicate the copy, and has no other option.
- For example if exec() is called immediately after fork() they never need to be copied the parent memory can be shared with the child, only when a write is performed it can be re-created.

Process Termination:
- When a parent forks a child, the two processes can take any turn to finish themselves and in some cases the parent may die before the child. In some situations, though, it is desirable for the parent process to wait until one or more child processes have completed.
- This can be done with the wait() family of system calls. These functions allow you to wait for a process to finish executing, enabling the parent process to retrieve information about its child's termination.

Process Wait:
There are four different system calls in the wait family:
-wait(int *status): Blocks & waits the calling process until one of its child processes exits. Return status via simple integer argument.
-waitpid (pid_t pid, int* status, int options): Similar to wait, but only blocks on a child with specific PID .

-wait3(int *status, int options, struct rusage *rusage): Returns resource usage information about the existing child process.
-wait4 (pid_t pid, int *status, int options, struct rusage *rusage): Similar to wait3, but on a specific child.

II. Inter Process Communications (IPC):

- Inter-process communication (IPC) is the mechanism whereby one process can communicate, that is exchange data with another process.
- There are two flavors of IPC exist: System V and POSIX.
IPC can be categorized broadly into two areas:
-Data exchange (Communication):Pipes, FIFO, Shared memory, Signals, Sockets.
-Resource usage/access/control (Synchronization): Semaphores.
⇒ User vs Kernel Space: How can processes communicate with each other and the kernel? The answer is nothing but IPC mechanisms.

Pipe:

- A pipe is a communication device that permits unidirectional communication. Data written to the "write end" of the pipe is read back from the "read end". Pipes are serial devices; the data is always read from the pipe in the same order it was written.

Pipes - Creation:

- int pipe(int pipe_fd[2]);

⇒ Pipe read and write can be done simultaneously between two processes by creating a child process using fork() system call.

FIFO:

- A first-in, first-out (FIFO) file is a pipe that has a name in the file-system. FIFO file is a pipe that has a name in the file-system. FIFOs are also called Named Pipes. FIFOs are designed to let them get around one of the shortcomings of normal pipes.

FIFO - Creation:

-FIFO can also be created similar to directory/file creation with special parameters & permissions. After creating FIFO, read & write can be performed into it just like any other normal file.
- int mknod(const char *path, mode_t mode, dev_t dev);
⇒ To communicate through a FIFO, one program must open it for writing, and another program must open it for reading.
-Either low-level I/O functions (open, write, read, close and so on) or C library I/O functions (fopen, fprintf, fscanf, fclose, and so on) may be used.

Shared Memories:

-Shared memory allows two or more processes to access the same memory. When one process changes the memory, all the other processes see the modification. Shared memory is the fastest form of Inter process communication because all processes share the same piece of memory. It also avoids copying data unnecessarily.
Note:

- Each shared memory segment should be explicitly de-allocated
- System has limited number of shared memory segments
- Cleaning up of IPC is system program's responsibility

⇒ While shared memory is the fastest IPC, it will create synchronization issues as more processes are accessing the same piece of memory. Hence it has to be handled separately.

Shared Memories – Function calls

- Create a shared memory segment: int shmget(key_t key, size_t size, int shmflag)
- Attach to a particular shared memory location: void *shmat(int shmid, void *shmaddr, int shmflag)
- Detach from a shared memory location: int shmdt(void *shmaddr)
- shmctl(shmid, IPC_RMID, NULL): shmid: Shared memory ID

Synchronization - Semaphores:

● Semaphores are similar to counters Process semaphores synchronize between multiple processes, similar to thread semaphores.
● The idea of creating, initializing and modifying semaphore values remain same in between processes also
● However there are different set of system calls to do the same semaphore operations

Synchronization – Semaphore Functions:

- Create a process semaphore: int semget(key_t key, int nsems, int flag)
- Wait and Post operations: int semop(int semid, struct sembuf *sops, unsigned int nsops)
- Semaphores need to be explicitly removed: semctl(semid, 0, IPC_RMID)

Synchronization - Debugging:

- The ipcs command provides information on inter-process communication facilities, including shared segments.

- Use the -m flag to obtain information about shared memory.

Signals:

- Signals are used to notify a process of a particular event. Signals make the process aware that something has happened in the system. Target process should perform some predefined actions to handle signals

- This is called 'signal handling'. Actions may range from 'self termination' to 'clean-up'

Get Basics Right: Function pointers

What is a function pointer?

- *Datatype *ptr; normal pointer*
- *Datatype (*ptr)(datatype,..); Function pointer*

How does it differ from a normal data pointer?

Function Pointer

- Holds address of function

- Pointing to an address from the code segment.

- Dereference to execute the function

- Pointer arithmetic not valid

Data Pointer

- Holds address of an object

- Pointing to a address from stack/heap/data

- Dereference to get value from address

- Pointer arithmetic is valid

Call back functions:

- In computer programming, a callback is a reference to executable code, or a piece of executable code, that is passed as an argument to other code. This allows a lower- level software layer to call a subroutine (or function) defined in a higher-level layer.

Signals Origins: The kernel

- A Process may also send a Signal to another Process

- A Process may also send a Signal to itself

- User can generate signals from command prompt: 'kill' command:

$ kill <signal_number> <target_pid>

$ kill –KILL 4481

Signal Handling:

- When a process receives a signal, it processes: Immediate handling

- For all possible signals, the system defines a default disposition or action to take when a signal occurs.

There are four possible default dispositions:

- *– Exit: Forces process to exit*
- *– Core: Forces process to exit and create a core file*
- *– Stop: Stops the process*
- *– Ignore: Ignores the signal*

- The signal() function can be called by the user for capturing signals and handling them accordingly

- First the program should register for interested signal(s)

- Upon catching signals corresponding handling can be done:

*⇒ signal (int signal_number, void *(fptr) (int))*

- A signal handler should perform the minimum work necessary to respond to the signal. The control will return to the main program (or terminate the program).

- In most cases, this consists simply of recording the fact that a signal occurred or some minimal handling. The main program then checks periodically whether a signal has occurred and reacts accordingly

- Its called as asynchronous handling

Signals vs Interrupt:

- Signals can be described as soft-interrupts. The concept of 'signals' and 'signals handling' is analogous to that of the 'interrupt' handling done by a microprocessor. When a signal is sent to a process or thread, a signal handler may be entered.

- This is similar to the system entering an interrupt handler.

Signals Advanced Handling

- The signal() function can be called by the user for capturing signals and handling them accordingly

- It mainly handles user generated signals (ex: SIGUSR1), will not alter default behavior of other signals (ex: SIGINT)

- In order to alter/change actions, sigaction() function to be used
- Any signal except SIGKILL and SIGSTOP can be handled using this

⇒ sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)

Advanced Handling – sigaction structure

```
struct sigaction
{
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restorer)(void);
}
```

- sa_handler: SIG_DFL (default handling) or SIG_IGN (Ignore) or Signal handler function for handling
- Masking and flags are slightly advanced fields
- Try out sa_sigaction during assignments/hands-on session along with Masking & Flags

Signals Self Signaling:

- A process can send or detect signals to itself. There are three functions available for this purpose, apart from 'kill'.

raise (int sig): Raise a signal to the currently executing process. Takes signal number as input.

alarm (int sec): Sends an alarm signal (SIGALRM) to the currently executing process after a specified number of seconds.

pause(): Suspends the current process until expected signal is received. This is much better way to handle signals than sleep, which is a crude approach

Networking Fundamentals:

- IP layer: IP address
  - Dotted decimal notation ("192.168.1.10")
  - 32 bit integer is used for actual storage
  - IP address must be unique in a network
  - Two modes IPv4 (32 bits) and IPv6 (128 bits)
  - Total bits divided into two parts: Network + Host
  - Host part obtained using subnet mask
- Commands related to networking
  - Ifconfig (/sbin/ifconfig) command to find the ip-address of system
  - Ping – To check the connectivity using ICMP protocol
  - Host – To convert domain name to ip-address
- TCP/UDP layer: Port numbers
  - Well known ports [ex: HTTP (80), Telnet (23)]
  - System Ports (0-1023)
  - User Ports (1024-49151)
  - Dynamic and/or Private Ports (49152-65535)
- Port number helps in multiplexing and demultiplexing the messages
- To see all port numbers used in system by opening a file /etc/services

Socket:

- Sockets is another IPC mechanism, different from other mechanisms as they are used in networking
- Apart from creating sockets, one need to attach them with network parameter (IP address & port) to enable it communicate it over network
- Both client and server side socket needs to be created & connected before communication
- Once the communication is established, sockets provide 'read' and 'write' options similar to other IPC mechanisms
- In order to attach (called as "bind") a socket to network address (IP address & Port number), a structure is provided
- This (nested) structure needs to be appropriately populated
- Incorrect addressing will result in connection failure

```
struct sockaddr_in
{
        short int sin_family;
        unsigned short int sin_port;
        struct in_addr sin_addr;
        unsigned char sin_zero[8];
};
```

```
/* IP address structure for historical reasons */
struct in_addr
{
        unsigned long s_addr; /* 32 bit IP address */
};
```

Calls - socket

- Create Socket: `int socket(int domain, int type, int protocol)`
- Bind function: `int bind(int sockfd, struct sockaddr *my_addr, int addrlen)`
- Connect function: `int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)`
- Listen function: `int listen(int sockfd, int backlog)`
- Accept function: `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
    - The accept() returns a new socket ID, mainly to separate control and data sockets
    - By having this servers become concurrent
    - Further concurrency is achieved by fork() system call
- Recv function: `int recv (int sockfd, void *buf, int len, int flags)`
- Send function: `int send(int sockfd, const void *msg, int len, int flags)`
- Close function: `close (int sockfd)`

Sockets UDP – Functions calls

- Send data through a UDP socket: `int sendto( int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, socklen_t length);`
- Receive data through a UDP socket: `int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *length);`

Client – Server Models:

● Iterative Model

– The Listener and Server portion coexist in the same task

– So no other client can access the service until the current running client finishes its task.

Steps:

1- Create a socket
2- Bind it to a local address
3- Listen (make TCP/IP aware that the socket is available)
4- Accept the connection request
5- Do data transaction
6- Close

● Concurrent Model

– The Listener and Server portion run under control of different tasks

– The Listener task is to accept the connection and invoke the server task

– Allows higher degree of concurrency

Steps:

1- Create a Listening socket
2- Bind it to a local address
3- Listen (make TCP/IP aware that the socket is available)
4- Accept the connection request in loop
5- Create a new process and passing the new sockfd
6- Do data transaction
7- Close (Both process depending on the implementation)

Synchronization - Concepts:

In a multitasking system the most critical resource is CPU. This is shared between multiple tasks / processes with the help of 'scheduling' algorithm When multiple tasks are running simultaneously:

– Either on a single processor, or on

– A set of multiple processors

They give an appearance that:

– For each process, it is the only task in the system.

– At a higher level, all these processes are executing efficiently.

– Process sometimes exchange information:

– They are sometimes blocked for input or output (I/O).

Whereas multiple processes run concurrently in a system by communicating, exchanging information with others all the time. They also have very close dependency with various I/O devices and peripherals.

- Considering resources are lesser and processes are more, there is a contention going between multiple processes

- Hence resources need to be shared between multiple processes. This is called as "Critical section"

- Access / Entry to the critical section is determined by scheduling, however exit from the critical section needs to be done when activity is completed properly. Otherwise it will lead to a situation called "Race condition".

- Synchronization is defined as a mechanism which ensures that two or more concurrent processes do not simultaneously execute some particular program segment known as critical section

- When one process starts executing the critical section (serialized segment of the program) the other process should wait until the first process finishes

- If not handled properly, it may cause a race condition where, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes

- If any critical decision to be made based on variable values (ex: real time actions – like medical system), synchronization problems will create a disaster as it might trigger totally opposite action than what was expected.

Race Condition in Embedded Systems:

- Embedded systems are typically lesser in terms of resources, but having multiple processes running. Hence they are more prone to synchronization issues, thereby creating race conditions

- Most of the challenges are due to shared data conditions. Same pathway to access common resources creates issues

- Debugging race conditions and solving them is a very difficult activity because you cannot always easily re-create the problem as they occur only in a particular timing sequence.

-Asynchronous nature of tasks makes race condition simulation and debugging as a challenging task, often spend weeks to debug and fix them

Critical Section

- The way to solve race condition is to have the critical section access in such a way that only one process can execute at a time

-If multiple process try to enter a critical section, only one can run and the others will sleep (means getting into blocked / waiting state)

- Only one process can enter the critical section; the other two have to sleep. When a process sleeps, its execution is paused and the OS will run some other task.

- Once the process in the critical section exits, another process is woken up and allowed to enter the critical section. This is done based on the existing scheduling algorithm.

- It is important to keep the code / instructions inside a critical section as small as possible (say similar to ISR) to handle race conditions effectively.

Synchronization Priority Inversion

- One of the most important aspects of a critical section is to ensure whichever process is inside it, has to complete the activities at one go. They should not be done across multiple context switches. This is called Atomicity

- Assume a scenario where a lower priority process is inside the critical section and higher priority process tries to enter

- Considering atomicity the higher priority process will be pushed into a blocking state. This creates some issues with regular priority algorithms.

-In this juncture if a medium priority task gets scheduled, it will enter into the critical section with a higher priority task that is made to wait. This scenario is further creating a change in priority algorithm

⇒ This is called as 'Priority Inversion' which alters the priority schema.

Critical Section Solutions:

Solution to critical section should have following three aspects
into it:

-Mutual Exclusion: If process P is executing in its critical section, then no other processes can be executing in their critical sections.

- Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

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

Critical Section Solutions - Mutual Exclusion:

- A Mutex works in a critical section while granting access. You can think of a Mutex as a token that must be grabbed before execution can continue.

- During the time that a task holds the mutex, all other tasks are waiting on the mutex to sleep.

- Once a task has finished using the shared resource, it releases the mutex. Another task can then wake up and grab the mutex.

Mutual Exclusion – Locking / Blocking

- A process may attempt to get a Mutex by calling a lock method. If the Mutex was unlocked (means already available), it becomes locked (unavailable) and the function returns immediately

- If the Mutex was locked by another process, the locking function blocks execution and returns only eventually when the Mutex is unlocked by the other process.

- More than one process may be blocked on a locked Mutex at one time

- When the Mutex is unlocked, only one of the blocked processes is unblocked and allowed to lock the Mutex. Other tasks stay blocked.

Critical Section Semaphores

- A semaphore is a counter that can be used to synchronize multiple processes. Typically semaphores are used where multiple units of a particular resource are available.

- Each semaphore has a counter value, which is a non-negative integer. It can take any value depending on the number of resources available.

- The 'lock' and 'unlock' mechanism is implemented via 'wait' and 'post' functionality in semaphore. Where the wait will decrement the counter and post will increment the counter.

- When the counter value becomes zero that means the resources are no longer available hence remaining processes will get into a blocked state.

Semaphores – 2 basic operations:

Wait operation:

– Decrements the value of the semaphore by 1

– If the value is already zero, the operation blocks until the value of the semaphore becomes positive

– When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns

Post operation:

– Increments the value of the semaphore by 1

– If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore.

– One of those threads is unblocked and its wait operation completes (brings the semaphore's value back to 0).

Critical Section Mutex & Semaphores

- Semaphores which allow an arbitrary resource count (say 25) are called counting semaphores.

- Semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

- A Mutex is essentially the same thing as a binary semaphore, however the differences between them are in how they are used.

- While a binary semaphore may be used as a Mutex, a Mutex is a more specific use-case, in that only the process that locked the Mutex is supposed to unlock it.

- This constraint makes it possible to implement some additional features in Mutexes.

Critical Section Practical Implementation

- The problem is critical section / race condition is common in multi-threading and multi-processing environments. Since both of them offer concurrency & common resource facilities, it will raise race conditions.

- However the common resource can be different. In case of multiple threads a common resource can be a data segment / global variable which is a shared resource between multiple threads.

- In case of multiple processes a common resource can be a shared memory.

Synchronization Threads - Mutex

- pthread library offers multiple Mutex related library functions

- These functions help to synchronize between multiple threads:

*int pthread_mutex_init(pthread_mutex_t *mutex const pthread_mutexattr_t *attribute)*

*int pthread_mutex_lock(pthread_mutex_t *mutex)*

*int pthread_mutex_unlock(pthread_mutex_t *mutex)*

*int pthread_mutex_destroy(pthread_mutex_t *mutex)*

Synchronization Threads - Semaphores:

- pthread library offers multiple Semaphore related library functions

- These functions help to synchronize between multiple threads:

*int sem_init (sem_t *sem, int pshared, unsigned int value)*

*int sem_wait(sem_t *sem): Wait on the semaphore (Decrements count)*

*int sem_post(sem_t *sem): Post on the semaphore (Increments count)*

*int sem_destroy(sem_t *sem): Destroy the semaphore*

Process Management - Concepts

- Concepts Scheduling

- It is a mechanism used to achieve the desired goal of multitasking. This is achieved by SCHEDULER which is the heart and soul of the operating System.

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue. Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.

Scheduling - Types:

*- Co-operative: **First Come First Serve (FCFS):** Round Robin Time slice(TS) based, Round Robin: Priority based.*

*- Pre-emptive: **Priority based:** Static: Rate Monotonic (RM), Dynamic: Earliest Deadline First(EDF)*

Co-operative vs Pre-emptive

- In Co-operative scheduling, processes co-operate in terms of sharing processor timing. The process voluntarily gives the kernel a chance to perform a process switch.

- In Preemptive scheduling, processes are preempted a higher priority process, thereby the existing process will need to relinquish CPU.

Scheduling – Types – FCFS:

- First Come First Served (FCFS) is a Non-Preemptive scheduling algorithm. FIFO (First In First Out) strategy assigns priority to process in the order in which they request the processor. The process that requests the CPU first is allocated to the CPU first. This is easily implemented with a FIFO queue for managing the tasks. As the process comes in, they are put at the end of the queue. As the CPU finishes each task, it removes it from the start of the queue and heads on to the next task.

Scheduling – Types – RR: Time Sliced

- Processes are scheduled based on time-slice, but they are time-bound.

- This time slicing is similar to FCFS except that the scheduler forces the process to give up the processor based on the timer interrupt.

- It does so by preempting the current process (i.e. the process actually running) at the end of each time slice. The process is moved to the end of the priority level.

Scheduling – Types – RR: Priority

- Processes are scheduled based on RR, but priority attached to it.

- While processes are allocated based on RR (with specified time), when a higher priority task comes in the queue, it gets preempted.

- The time slice remains the same.

Scheduling – Types – Preemptive

Pre-emption means while a lower priority process is executing on the processor another process higher in priority than comes up in the ready queue, it preempts the lower priority process.

Rate Monotonic (RM) scheduling:

- *The highest Priority is assigned to the Task with the Shortest Period*
- *All Tasks in the task set are periodic*
- *The relative deadline of the task is equal to the period of the Task*
- *Smaller the period, higher the priority*

Earliest Deadline First (EDF) scheduling:

- *This kind of scheduler tries to give execution time to the task that is most quickly approaching its deadline*
- *This is typically done by the scheduler changing priorities of tasks on-the-fly as they approach their individual deadlines*

Memory Management - Concepts

Requirements - Relocation

- Programmer does not know where the program will be placed in memory when it is executed.

- Before the program is loaded, address references are usually relative addresses to the entry point of the program. These are called logical addresses, part of logical address space.

- All references must be translated to actual addresses. It can be done at compile time, load time or execution.

- Mapping between logical to physical address mechanisms is implemented as "Virtual memory".

- Paging is one of the memory management schemes where the program retrieves data from the secondary storage for use in main memory.

Virtual memory – Why?

- *If programs access physical memory we will face three problems*
  - *Don't have enough physical memory.*

– *Holes in address space (fragmentation).*
– *No security (All program can access same memory)*

● *These problems can be solved using virtual memory.*
– *Each program will have their own virtual memory.*
– *They separately map virtual memory space to physical memory.*
– *We can even move to disk if we run out of memory (Swapping)*

Virtual memory – Paging & page table

- Virtual memory divided into small chunks called pages.
- Similarly physical memory is divided into frames.
- Virtual memory and physical memory mapped using a page table.

Virtual memory – TLB

- For faster access page tables will be stored in CPU cache memory. But limited entries are only possible.
- If a page entry is available in TLB (Hit), control goes to the physical address directly (Within one cycle).
- If page entry is not available in TLB (Miss), it uses the page table from main memory and maps to physical address(Takes more cycles compared to TLB).

Page fault

- When a process tries to access a frame using a page table and that frame is moved to swap memory, it generates an interrupt called page fault.

Page fault – Handling

1.Check an internal table for this process, to determine whether the reference was a valid or it was an invalid memory access.
2.If the reference was invalid, terminate the process. If it was valid, but the page has not yet been brought in, page in the latter.
3.Find a free frame.
4.Schedule a disk operation to read the desired page into the newly allocated frame.
5.When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6.Restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

MMU: Memory Management Unit

- MMU is responsible for all aspects of memory management. It is usually integrated into the processor, although in some systems it occupies a separate IC (integrated circuit) chip.
The work of the MMU can be divided into three major categories:
● *Hardware memory management, which oversees and regulates the processor's use of RAM (random access memory) and cache memory.*
● *OS (operating system) memory management, which ensures the availability of adequate memory resources for the objects and data structures of each running program at all times.*
●*Application memory management, which allocates each individual program's required memory, and then recycles freed- up memory space when the operation concludes.*

MMU - Relocation

- The logical address of a memory allocated to a process is the combination of base register and limit register. When this logical address is added to the relocation register, it gives the physical address.

MMU Requirements - Protection

- Processes should not be able to reference memory locations in another process without permission.
- Impossible to check absolute addresses in programs since the program could be relocated.
- Must be checked during execution.

MMU Requirements - Sharing

- Allow several processes to access the same portion of memory.
- For example, when using shared memory IPC, we need two processes to share the same memory segment

MMU Requirements – Logical Organization

- Logical Organization Memory is organized linearly (usually) In contrast, programs are organized into modules..
- Modules can be written and compiled independently. Different degrees of protection can be given to different modules (read-only, execute-only).
- Modules can be shared among processes Segmentation helps here. In Linux, Code Segment has a read-only attribute

MMU Requirements – Physical Organization

Processes in the user space will be leaving & getting in
● *Each process needs the memory to execute, So, the memory needs to be partitioned between processes*