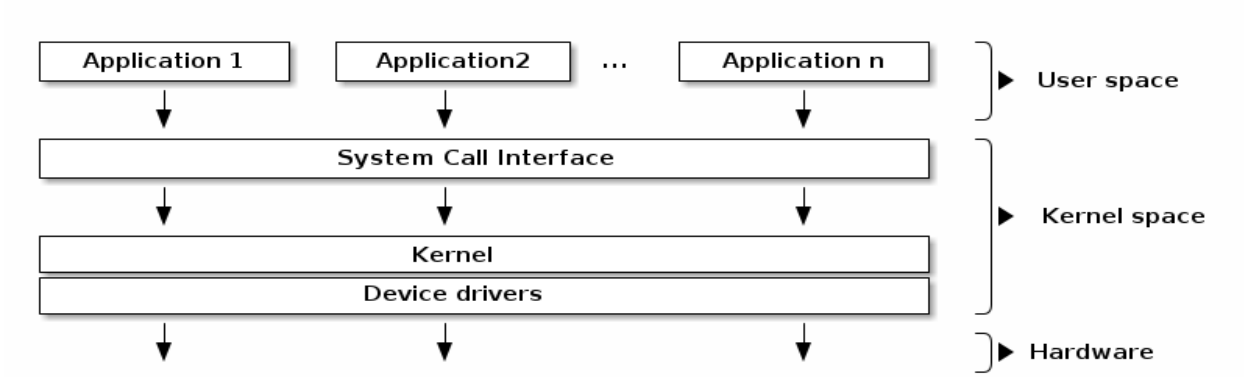
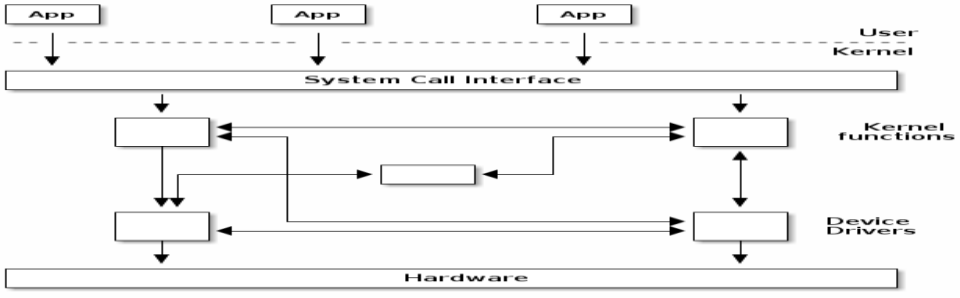
----------Linux kernel Introduction----------

Operating System: An operating system is a "resource allocator" and a "controlling of operations" program.

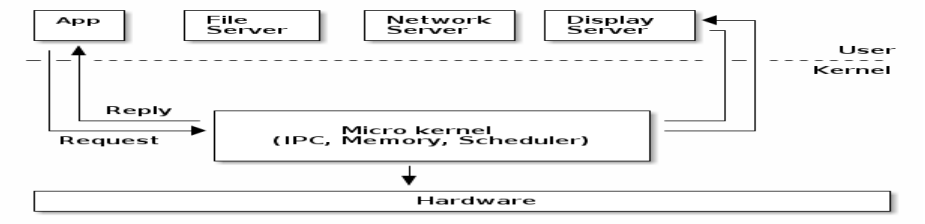


User space and Kernel space: User mode and kernel mode are the processor execution mode. Code that runs in kernel mode can fully control the CPU while code that runs in user mode has certain limitations. The kernel space is accessed protected so that user applications cannot access it directly, while user space can be directly accessed from code running in kernel mode.

Kernel has two components: core component (physical memory manager, virtual memory manager, file manager, Interrupt handler, process manager etc.), non-core component (compiler, libs etc.)
Monolithic: All the parts of a kernel components like the Scheduler, File System, Memory Management, Networking Stacks, Device Drivers, etc., are maintained in one unit within the kernel in Monolithic Kernel.
Faster processing(advantage) but **kernel becomes panic if any of module goes down(Disadvantages)**



Microkernel: Only the very important parts like IPC (Inter process Communication), basic scheduler, basic memory handling, basic I/O primitives etc., are put into the kernel. Communication happens via message passing. Others are maintained as server processes in User Space.
Slower Processing due to additional Message Passing.



Real mode: This is the only mode which was supported by the 8086 (the very first processor of the x86 series). The 8086 had 20 address lines, so it was capable of addressing "2 raised to the power 20" i.e. 1 MB of memory. No multi-tasking – no protection is there to keep one program from overwriting another program.

Protected mode: Multitasking and There is no 1 MB limit in protected mode. Support for virtual memory, which allows the system to use the hard disk to emulate additional system memory when needed.

Types of Operating Systems:

Batch operating system:
The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator.

Multiprogramming system: It is non preemptive system. If CPU is executing process P1, then it will finish execution of P1 and then only it will execute other process. If process goes for some I/O operation, then only CPU will switch to other process.

Multi-tasking/Timesharing system: It is preemptive and timesharing system. CPU will execute process P1 for fixed time and will switch to other process. So, CPU will be less idle as compared to multiprogramming system.

Distributed operating System: Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems.

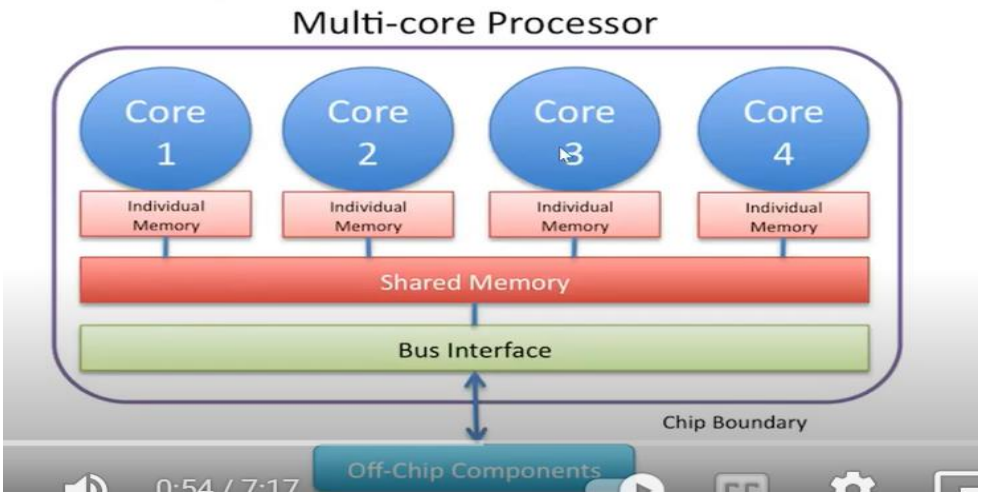
The advantages of distributed systems are as follows –

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Reduction of the load on the host computer. Reduction of delays in data processing.

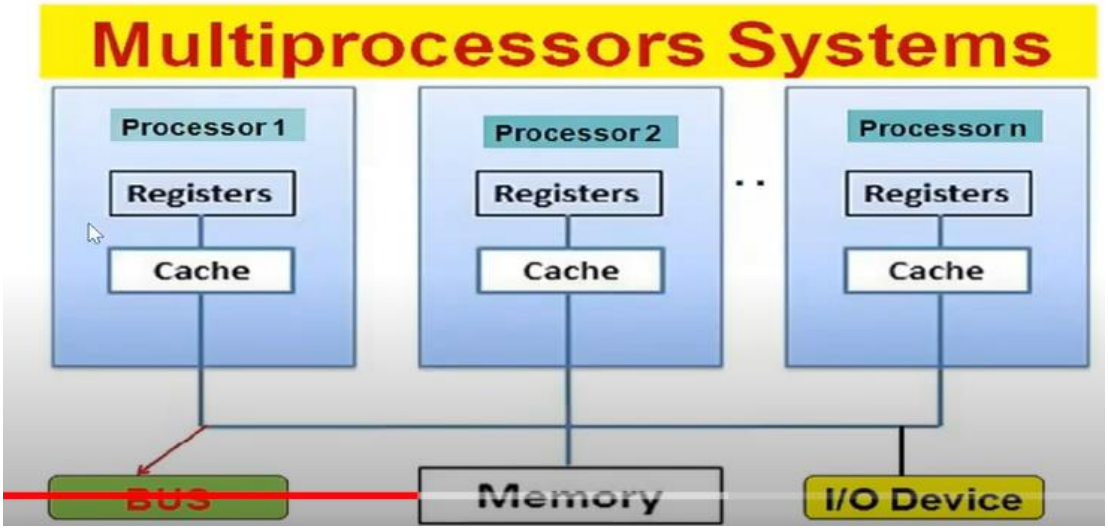
Network operating System: A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions.

Real Time Operating System: is an operating system (OS) for real-time applications that processes data and events that have critically defined time constraints.

Multicore system: A processor that has more than one core is called Multi core Processor while one with single core is called Unicore Processor or Uni processor. Nowadays, most of systems have four cores (Quad-core) or eight cores (Octa-core). These cores can individually read and execute program instructions, cores are integrated into single chip and will require less time.



Multiprocessor system: Systems which have more than one processor are called multiprocessor system. These systems are also known as parallel systems or tightly coupled systems.



Multiprocessor systems have the following advantages.

- **Increased Throughput:** Multiprocessor systems have better performance than single processor systems. It has shorter response time and higher throughput. User gets more work in less time.
- **Reduced Cost:** Multiprocessor systems can cost less than equivalent multiple single processor systems. They can share resources such as memory, peripherals etc.
- **Increased reliability:** Multiprocessor systems have more than one processor, so if one processor fails, complete system will not stop. In these systems, functions are divided among the different processors.

Differences b/w multiprocessor and multi core system:

MultiCore	MultiProcessor
A single CPU or processor with two or more independent processing units called cores that are capable of reading and executing program instructions.	A system with two or more CPU's that allows simultaneous processing of programs.
It executes single program faster.	It executes multiple programs Faster.
Not as reliable as multiprocessor.	More reliable since failure in one CPU will not affect other.
It has less traffic.	It has more traffic.
It does not need to be configured.	It needs little complex configuration.

It's very cheaper (single CPU that does not require multiple CPU support system).	It is Expensive (Multiple separate CPU's that require system that supports multiple processors) as compared to MultiCore.

Distributed System: A distributed environment refers to multiple independent CPUs or processors in a computer system. An operating system does the following activities related to distributed environment −

- The OS distributes computation logics among several physical processors.
- The processors do not share memory or a clock. Instead, each processor has its own local memory.
- The OS manages the communications between the processors. They communicate with each other through various communication lines.

Spooling: Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices.

Asymmetric Multiprocessing (ASMP): All processors are not identical as one behaves as master and rest others as slave, where each one holds own memory space. The master processor has the operating system within which all system calls are executed.

Symmetric Multiprocessing (SMP): All the processors are identical and forms connection with a shared memory.

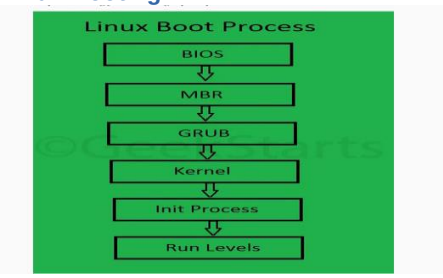
A single operating system can be accessed by each processor for task execution.

An Introduction to Linux: When installing Linux, the source code is usually stored in /usr/src/linux.
Kernel: kernel performs below main tasks:
Process management
- Device management
- Memory management
- Interrupt handling
- I/O communication
- File system management

The kernel exists as a physical file on the file system in Linux it is /boot directory and is usually called vmlinux.
/boot/vmlinuz-2.4.18-22

- At system boot time RAM only contains the boot loader, consuming a few kilobytes at the beginning of memory.
- The boot loader loads the kernel binary into memory from the hard disk, and places
- it at the beginning of memory.
- Once the kernel has been read in the boot loader tells the CPU to execute it by
- issuing a JMP (Jump) instruction.

Linux Booting:



steps:

CPU executes BIOS code first.
Stage 1: BIOS: First BIOS performs POST(power on self test) which ensures all hardware devices are working properly. BIOS responsibilities are to search boot loader (GRUB, LILO) program to load and execute it. BIOS search boot loader into a floppy, CD-rom, or hard drive.

Stage 2: MBR: MBR located into the 1st sector of the bootable drive. Generally, /dev/had, or /dev/sda. MBR stores in 512 bytes in size, there are three components of MBR.

- Primary boot loader information stored in 1st 446 bytes
- Partition table information stored in the next 64 bytes
- MBR validation check stored in the last 2 bytes.

Stage 3: GRUB: It stores all information about operating system image to load and execute.
If you have more than one operating system, all entry will be in this GRUB file, and you can choose to make the default one.

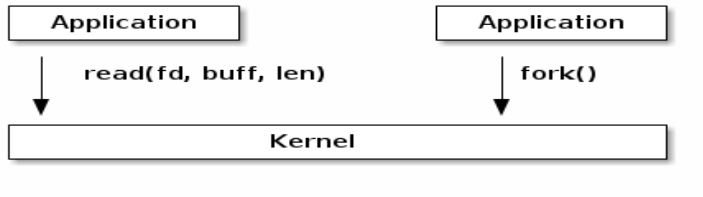
Stage 4: Kernel: When kernel loads, it mount the root file system and execute /sbin/init program. As init was the first program run by the Linux kernel, it has one(1) as process id (PID), which you can check by using below command:

Stage 5: Init: It checks file "/etc/inittab" to decide the Linux run level.
0 – halt, 1 – single-user mode, 2 – Multiuser, without NFS, 3 – Full multiuser mode, 4 – unused, 5 – X11, 6 – reboot,

Stage 6: Runlevel: One default run level identified; it will execute all required program for that run level.

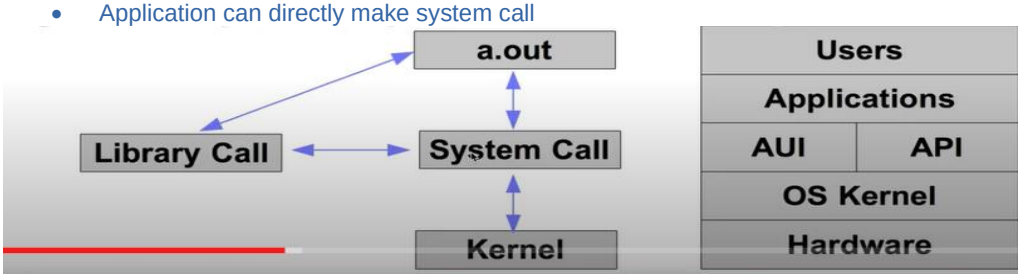
----------System Calls----------

The kernel offers a set of APIs that applications issue which are generally referred to as "System Calls". These APIs are different from regular library APIs because they are the boundary at which the execution mode switch from user mode to kernel mode.



Two ways for Application to communicate with system/kernel:

- Application can make library call and then library call to system call
- Application can directly make system call

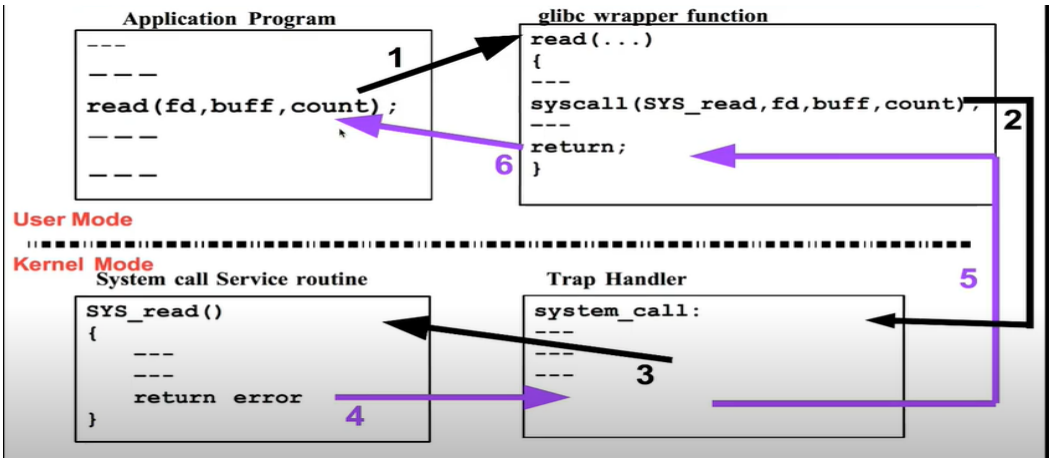


System call is controlled entry point into kernel code, allowing a process to request kernel to perform privileged operation. System call changes the CPU mode from user to kernel so that CPU can access protected memory. Each system call may have set of arguments that specify information to be transferred from user space to kernel space.

In Linux, system calls are identified by numbers and the parameters for system calls are machine word sized (32 or 64 bit). There can be a maximum of 6 system call parameters. Both the system call number and the parameters are stored in certain registers. For example, on 32bit x86 architecture, the system call identifier is stored in the EAX

register, while parameters in registers EBX, ECX, EDX, ESI, EDI, EBP. System libraries (e.g. libc) offers functions that implement the actual system calls in order to make it easier for applications to use them.

- Application makes system calls by invoking wrapper function of glibc (c library). It passes arguments by pushing them into stack in reverse order.
- Wrapper function copies system call number to specific CPU registers. It also Copies arguments from process stack to CPU registers.
- Now wrapper function executes the trap machine instruction, that causes CPU to switch from user mode to kernel mode. Trap handler saves system call no and arguments from CPU registers to process kernel stack, indexes system call table with system call no. to find address of system call routine. Trap handler transfers control to system call service routine.
- System call executes and returns success or error. Trap handler again executes trap machine instruction to switch from kernel to user mode and returns control to wrapper function.



Accessing user space from system calls:
As we mentioned earlier, user space must be accessed with special APIs (get_user(), put_user(), copy_from_user(), copy_to_user()) that check whether the pointer is in user space and also handle the fault if the pointer is invalid. In case of invalid pointers, they return a non-zero value.

Add the new system call to the system call table:
1) If you are on a 32-bit system you'll need to change 'syscall_32.tbl'. For 64-bit, change 'syscall_64.tbl'.

2) **Add new system call to the system call header file:** cd include/linux/ and update syscall.h

asmlinkage long sys_hello(void);

This defines the prototype of the function of our system call. "asmlinkage" is a key word used to indicate that all

parameters of the function would be available on the stack.

----------The Kernel versus Process Management----------

Process:
A process is basically a program in execution. On batch systems, it is called as a "job" while on time sharing systems, it is called as a "task". A summary of the resources a process has can be obtain from the */proc/<pid>* directory.

Important functions of process management are:
- Creation and deletion of system processes.
- Creation and deletion of users.

- CPU scheduling.
- Process communication and synchronization.

Process Priority: Each process has its priority in which it is executed**.** It can be checked using top command.
nice and renice command: nice command is used to start a process with specified nice value, which renice command is used to alter priority of running process.
nice -n -5 bash
renice value PID

This all works in priority based scheduling mechanism:

priority Inversion: when priority of process gets inverted.
Ex: There are three process of priority L,M and H.
1) L and H have to run critical section. L is executing critical section and H is waiting to execute critical section.
2) Now M came and preempted the process L, M is not going to execute critical section.
3) M will complete execution then L will resume and then H will get chance.

Solution:
Priority Inheritance: The basic idea of the priority inheritance protocol is that when a job blocks one or more high-priority jobs, it ignores its original priority assignment and executes its critical section at an elevated priority level.
Ex: Now L will execute with high priority of H and M can not preempt it because L is executing with high priority. Once L completes execution, will coma back to it's original priority.

Priority Ceiling: Priority ceiling protocol involves assigning a "priority ceiling level" to each resource or lock. Whenever a task works with a particular resource or takes a lock, the task's priority level is automatically boosted to that of the priority ceiling associated with the lock or resource.

Context switching: is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single central processing unit (CPU) and is an essential feature of a multitasking operating system**.**

Context Switching Triggers
There are three major triggers for context switching. These are given as follows −

- **Multitasking**: In a multitasking environment, a process is switched out of the CPU so another process can be run. The state of the old process is saved, and the state of the new process is loaded. On a pre-emptive system, processes may be switched out by the scheduler.
- **Interrupt Handling**: The hardware switches a part of the context when an interrupt occurs. This happens automatically. Only some of the context is changed to minimize the time required to handle the interrupt.
- **User and Kernel Mode Switching**: A context switch may take place when a transition between the user mode and kernel mode is required in the operating system

Context Switching Steps
The steps involved in context switching are as follows −

- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue etc.
- Select a new process for execution.
- Update the process control block of the selected process. This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor. This is done by loading the previous values of the process control block and registers**.**

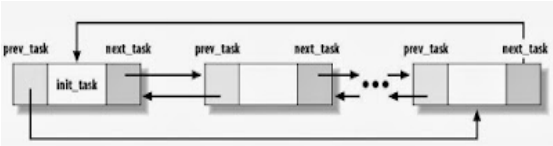
  Context Switching Cost:
  Context Switching leads to an overhead cost because of TLB flushes, sharing the cache between multiple tasks, running the task scheduler etc. Context switching between two threads of the same process is faster

than between two different processes as threads have the same virtual memory maps. Because of this TLB flushing is not required.

PCB contains:

- **Program counter:** It indicates the address of the next instruction to be executed for this process.
- **CPU Registers:** They include index registers, stack pointer and general-purpose registers. It is used to save process state when an interrupt occurs, so that it can resume from that state.
- **CPU-scheduling information:** it includes process priority, pointer to scheduling queue.
- **Memory management information:** value of the base and limit registers, page tables depending on the memory system.
- **Accounting information:** it contains an amount of CPU and real time used, time limits process number and so on.
- **I/O status information:** It includes a list of I/O devices allocated to the process, a list of open files and so on.

Process Descriptor and the Task Structure:
The kernel stores the list of processes in a circular doubly linked list called the task list. Process descriptor is nothing but each element of this task list of the type **struct task_struct** , which is defined in <linux/sched.h> . The process descriptor contains all the information about a specific process. The task_struct is a relatively large data structure, at around 1.7 kilobytes on a 32-bit machine.



Each thread has its own thread_info. There are two basic reasons why **there are two such structures.**

Accessing the current process:

- opening a file needs access to struct task_struct's file field
- mapping a new file needs access to struct task_struct's mm field
- Over 90% of the system calls needs to access the current process structure so it needs to be fast
- The current macro is available to access to current process's struct task_struct

Blocking the current thread:

- Set the current thread state to TASK_UINTERRUPTIBLE or TASK_INTERRUPTIBLE
- Add the task to a waiting queue
- Call the scheduler which will pick up a new task from the READY queue
- Do the context switch to the new task

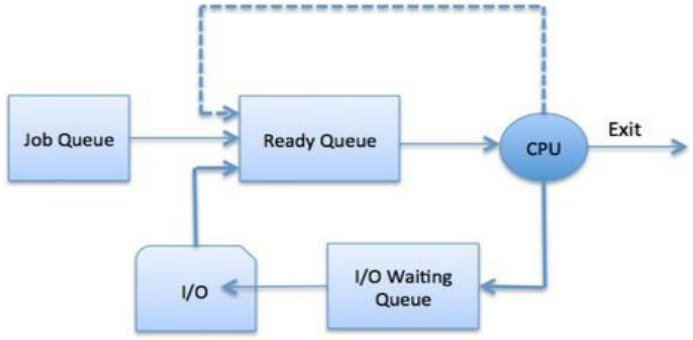
Waking up a task:

- Select a task from the waiting queue
- Set the task state to TASK_READY
- Insert the task into the scheduler's READY queue
- On SMP system this is a complex operation: each processor has its own queue, queues need to be balanced, CPUs needs to be signaled

---

Process Queue:

- The Operating system manages various types of queues for each of the process states. The PCB related to the process is also stored in the queue of the same state. If the Process is moved from one state to another state then its PCB is also unlinked from the corresponding queue and added to the other state queue in which the transition is made.



There are the following queues maintained by the Operating system:

1. Job Queue: Initially, all the processes get stored in the job queue. It is maintained in the secondary memory. The long-term scheduler (Job scheduler) picks some of the jobs and put them in the primary memory.

2. Ready Queue: Ready queue is maintained in primary memory. The short-term scheduler picks the job from the ready queue and dispatch to the CPU for the execution.

3. Waiting Queue: When the process needs some IO operation to complete its execution, OS changes the state of the process from running to waiting. The context (PCB) associated with the process gets stored on the waiting queue which will be used by the Processor when the process finishes the IO.

CPU Bound process: The term CPU-bound describes a scenario where the execution of a task or program is highly dependent on the CPU.

I/O bound process: Any application that involves reading and writing data from an input-output system, as well as waiting for information, is considered I/O bound.

- Programs that are I/O bound are often slower than CPU-bound programs.
- Due to the use of the input-output system, the time spent waiting for data to be read or written can be substantial. This is considerably slower than the time it takes a processor to complete operations.

Process State: When process executes, it changes the state.
2.1. Running or Runnable State ®: When a new process is started, it'll be placed into the running or runnable state. **In the running state, the process takes up a CPU core to execute its code and logic.** However, the thread scheduling algorithm might force a running process to give up its execution right. This is to ensure each process can have a fair share of CPU resources. In this case, the process will be placed into a run queue, and its state is now a runnable state waiting for its turn to execute.

2.2. Sleeping State: Interruptible (S) and Uninterruptible (D): During process execution, it might come across a portion of its code where it needs to request external resources. Mainly, the request for these resources is IO-based such as to read a file from disk or make a network request. Since the process couldn't proceed without the resources, it would stall and do nothing. In events like these, they should give up their CPU cycles to other tasks that are ready to run, and hence they go into a sleeping state.

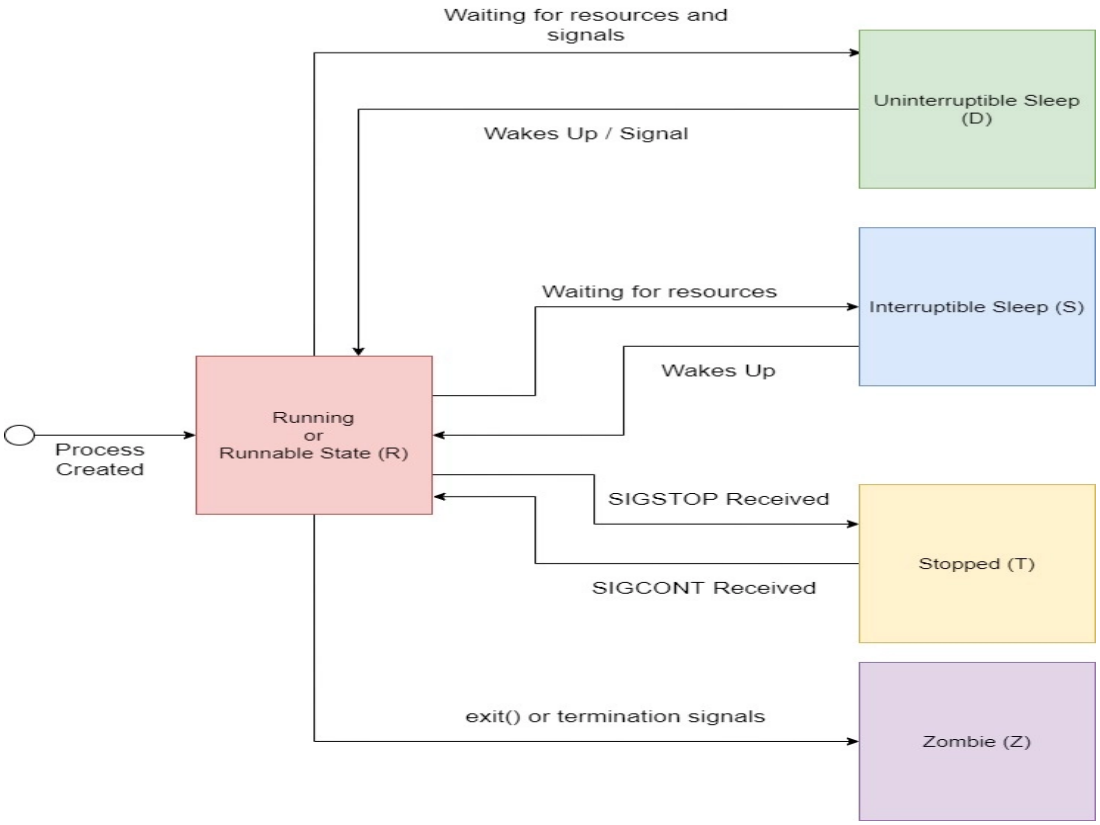
There are two different sleeping states: the uninterruptible sleeping state (D) and the interruptible sleeping state (S).The uninterruptible sleeping state will only wait for the resources to be available before it transit into a runnable state, and it doesn't react to any signals. On the other hand, the interruptible sleeping state (s) will react to signals and the availability of resources.

2.3. Stopped State (T): From a running or runnable state, we could put a process into the stopped state (T) using the SIGSTOP or SIGTSTP signals. The difference between both signals is that we send the SIGSTOP is

programmatic, such as running kill -STOP {pid}. Additionally, the process cannot ignore this signal and will go into the stopped state. On the other hand, we send the SIGTSTP signal using the keyboard CTRL + Z. Unlike SIGSTOP, the process can optionally ignore this signal and continue to execute upon receiving SIGTSTP.

2.4. Zombie State (Z):

When a process has completed its execution or is terminated, it'll send the SIGCHLD signal to the parent process and go into the zombie state. The zombie process, also known as a defunct process, will remain in this state until the parent process clears it off from the process table. To clear the terminated child process off the process table, the parent process must read the exit value of the child process using the wait() or waitpid() system calls



Process Context: When a program executes a system call or triggers an exception, it enters Kernel-space. At this point, the kernel is said to being "executing on behalf of the process" and are in process context. When in process context, the current macro is valid.

Interrupt context: When executing a interrupt handler or bottom half, the kernel is in interrupt context.

Code that runs in interrupt context has the following properties:

it runs because of an IRQ (not of an exception)

there is no well-defined process context associated, not allowed to trigger a context switch (no sleep, schedule, or user memory access)

Daemon process: Disk and execution monitor, is a process that runs in the background without user's interaction. They usually start at the booting time and terminate when the system is shut down. The name of daemons usually ends with 'd' at the end in Unix. **Ex: httpd, named, lpd.**

Orphan process: If a parent has terminated before reaping its children and child process is still running. These children are called orphans. They are adopted by Init process, which do the reaping.

Zombie process: is a process that has terminated but parent process has not collected the exit status of child process and not reaped child process. When a process terminates but still holds the system resources like PCB and various table maintained by OS. It's half alive and half dead because it's holding memory, but it's never scheduled on CPU. Zombie process cannot be killed by signals. The only way to kill them, to kill parent process and then it is adopted by Init process.

Schedulers: Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. The scheduler is invoked: when there is change in process state, new process is created, software interrupt, hardware interrupt etc. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long Term Scheduler: It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound, and processor bound.

Short Term Scheduler: It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler: Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes. A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out.

Why we use scheduler:

- Max CPU utilization
- Max throughput
- Min waiting time

Scheduling Algorithms:

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin (RR) Scheduling

First Come First Serve (FCFS): Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue.

Round robin: Each process is assigned a fixed time (Time Quantum/Time Slice) in cyclic way. It is designed especially for the time-sharing system. 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 up to 1-time quantum.

Threads: the units of execution within a program.

Each thread within a process has a
1. unique program counter
2. process stack,
3. set of processor registers.

Threads are light weight. They don't have their own memory spaces and other resources unlike processes. All processes start with a single thread. So they behave like lightweight processes but are always tied to a parent "thick" process. So, creating a new process is a slightly heavy task and involves allocating all these resources while cresting a thread does not. Killing a process also involves releasing all these resources while a thread does not. However, killing a thread's parent process releases all resources of the thread. A process is suspended by itself and resumed by itself. Same with a thread but if a thread's parent process is suspended then the threads are all suspended.

Multi-threading:
Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.
Threads operate faster than processes due to following reasons:
1) Thread creation is much faster.
2) Context switching between threads is much faster.
3) Threads can be terminated easily
4) Communication between threads is faster.

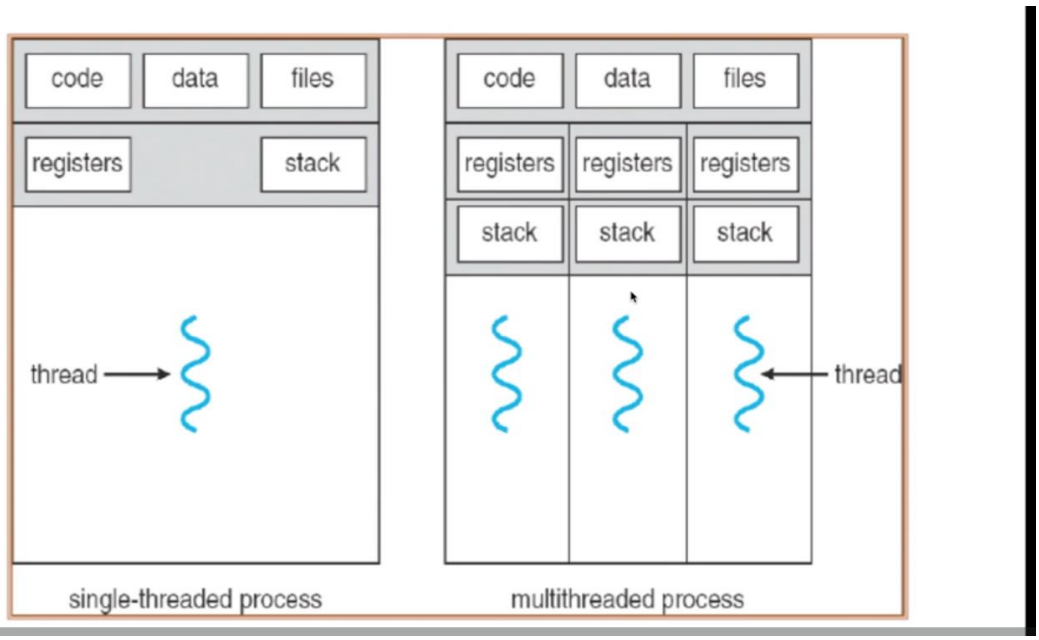
Types of Thread: Threads are implemented in following two ways –
User Level Threads – User managed threads. The thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts
Kernel Level Threads – There is no thread management code inthe application area. Kernel threads are supported directly by the operating system.

Differences b/w user level threads and kernel level threads:

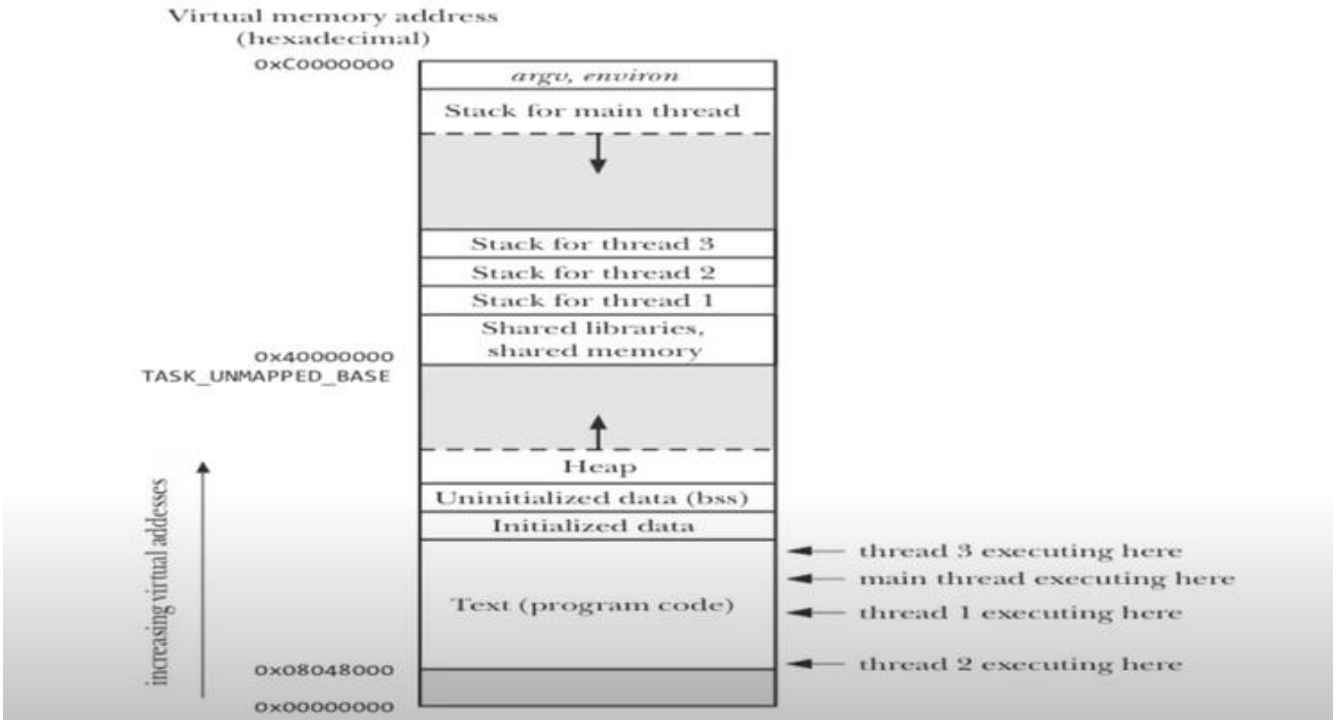
User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Multithreading Models:
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- **Many to many models**: It multiplexes any number of user threads to an equal or smaller number of kernel threads.
- **Many to one model**: It maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked.
- **One to one model:** This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.



Each thread has its own program counter, stack and registers, and thread shares the code section, data section, files, and Heap memory.



Threads within a process share :

- PID, PPID, PGID, SID, UID, GID
- Code and Data Section
- Global Variables
- errno variable
- Open files via PPFDT
- Signal Handlers
- Interval Timers
- CPU time consumed
- Resources Consumed
- Nice value
- Record locks (created using fcntl())

Threads have their own:

- Thread ID
- CPU Context (PC, and other registers)
- Stack
- State
- The `errno` variable
- Priority
- CPU affinity
- Signal mask

Pthread APIs:

```
int pthread_create(pthread_t *tid, const pthread_attr_t
                   *attr, void *(*start)(void *), void *arg) ;
```

- This function starts a new thread in the calling process. The new thread starts its execution by invoking the start function which is the 3rd argument to above function
- On success, the TID of the new thread is returned through 1st argument to above function
- The 2nd argument specifies the attributes of the newly created thread. Normally we pass `NULL` pointer for default attributes.
- The 4th argument is a pointer of type void which points to the value to be passed to thread start function. It can be `NULL` if you do not want to pass any thing to the thread function. It can also be address of a structure if you want to pass multiple arguments

```
void pthread_exit(void *status);
```

- This function terminate the calling thread
- The `status` value is returned to some other thread in the calling process, which is blocked on the `pthread_join()` call
- The pointer `status` must not point to an object that is local to the calling thread, since that object disappears when the thread terminates

Ways for a thread to terminate:

- The thread function calls the `return` statement
- The thread function calls `pthread_exit()`
- The main thread returns or call `exit()`
- Any sibling thread calls `exit()`

```
int pthread_join(pthread_t tid, void **retval);
```

- Any peer thread can wait for another thread to terminate by calling `pthread_join()` function, similar to `waitpid()`. Failing to do so will produce the thread equivalent of a zombie process
- The 1st argument is the ID of thread for which the calling thread wish to wait. Unfortunately, we have no way to wait for any of our threads like `wait()`
- The 2nd argument can be `NULL`, if some peer thread is not interested in the return value of the new thread. Otherwise, it can be a double pointer which will point to the status argument of the `pthread_exit()`

Returning value from thread function:

A thread function can return a pointer to its parent/calling thread, and that can be received in the 2nd argument of the `pthread_join()` function

The pointer returned by the `pthread_exit()` must not point to an object that is local to the thread, since that variable is created in the local stack of the terminating thread function

Making the local variable `static` will also fail. Suppose two threads run the same `thread_function()`, the second thread may over write the static variable with its own return value and return value written by the first thread will be over written

So the best solution is to create the variable to be returned in the heap instead of stack

Thread Attributes:

Every thread has a set of attributes which can be set before creating it. If we pass a `NULL` as second argument to `pthread_create()` function, the default thread attributes are used. The default value of thread attributes are shown in table below:

Attribute	Default Value	Description
`detachstate`	PTHREAD_CREATE_JOINABLE	Joinable by other threads
`stackaddr`	NULL	Stack allocated by system
`stacksize`	NULL	2 MB
`priority`	---	Priority of calling thread is used
`policy`	SCHED_OTHER	Determined by system
`inheritsched`	PTHREAD_INHERIT_SCHED	Inherit scheduling attributes from creating thread

Joinable Thread:

A joinable thread (like a process) is not automatically cleaned up by GNU/LINUX when it terminates. The thread's exit status hangs around in system until another thread calls `pthread_join()` to obtain its return value. Only then its resources are released. For example whenever we want to return data from child thread to parent thread the child thread must be a joinable thread

Detached Thread:

A detachable thread is cleaned up automatically when it terminates. Since a detached thread is immediately cleaned up, another thread may not wait for its completion by using `pthread_join()` to obtain its return value. For example suppose the main thread crates a child thread to do back up of a file and the main thread continue its execution. When the backup is finished , the second thread can just terminate. There is no need for it to rejoin the main thread. A thread can detach itself using `pthread_detach(pthread_self())`

Fork: The fork system call is used to create a new process. The newly created process is the child process. The process which calls fork and creates a new process is the parent process. The child and parent processes are executed concurrently.Fork returns 0 for child process and positive value for parent process and -1 for error.
//child process does not inherit parent's memory locks and timers. child process inherits mutex, condition variables, open file descriptor, message queue descriptor. page tables are copied, and page frames are shared.

Exec:

The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point. As a new process is not created, the process identifier (PID) does not change, but the machine code, data , heap , and stack of the process are replaced by those of the new program. Exec() replaces the current process with a the executable pointed by the function.

- A process may overwrite itself with another executable image. When a process calls one of the six **`exec()`** functions, it is completely replaced by the new program, and the new program starts executing its `main` function
- There are five library functions of `exec` family and all are layered on top of the **`execve()`** system call. Each of these functions provides a different interface to the same functionality
- There is no return after a successful `exec` call. The `exec()` functions return only if an error has occurred. The return value is -1, and `errno` is set to indicate the error

```
int execl (const char *pathname, const char* arg0,...,(char*)0);
int execlp (const char *filename, const char* arg0,...,(char*)0);
int execle (const char* pathname, const char* arg0,...,(char*)0,
                  char* const envp[]);
```

- The first argument to this family of `exec()` calls, is the name of the executable, which on success will overwrite the address space of the calling process with a new program from the secondary storage
- The **l** after the `exec` means that command line arguments to the new program will be passed as a comma separated list of strings with a `'\0'` character at the end
- The **p** stands for path. It means that the program specified as the first argument should be searched in all directories listed in the PATH variable. However, using absolute path to program is more secure than relying on PATH variable, which can be more easily altered by malicious users
- The **e** stands for environment. It means that after the command line arguments, the program should pass an array of pointers to null terminated strings, specifying the new environment of the program to be executed. Otherwise, the caller environment will be used

---

Vfork: The basic difference between vfork and fork is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues.

Clone: Clone, as fork, creates a new process. Unlike fork, these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. In Linux a new thread or process is create with the clone() system call. Both the fork()system call and the pthread_create() function uses the clone() implementation.

Check process details with process ID:
ls -l /proc/pid/exe

Wait system call:

```
pid_t wait(int *status)
```

- The process that calls the `wait()` system call gets blocked till any one of its child terminates
- The child process returns its termination status using the `exit()` call and that integer value is received by the parent inside the `status` argument (used for reaping and cleaning zombies from system). On the shell, we can check this value in the `$?` environment variable
- On success, the `wait()` system call returns PID of the terminated child and in case of error returns a -1
- If a process wants to wait for termination of all its children, then

```
while(wait(null) > 0);
```

Purposes of Wait system call:

1) Notify the parent process that child process has terminated.
2) Tell the parent how a child process finished.

A process can end in four ways:

- **Success /Failure:** On successful completion of the task, programs call `exit(0)` or `return 0` from `main()` function. In case of failure, programs call `exit()` with a non-zero value. The programmer need to document these error values in the manual page
- **Killed by a Signal:** A process might get killed by a signal generated from the keyboard, form an interval timer, from the kernel, or from another process
- **Stopped by a Signal:** A process might get SIGTSTP signal and temporary suspend its execution
- **Continued by a Signal:** A process might get SIGCONT signal and continue its execution

Instead of bit operators, we can use macros to decipher the **`status`** argument of **`wait()`** , defined in **`/usr/include/x86_64-linux-gnu/bits/waitstatus.h`**

WIFEXITED (status)	• This macro returns true if child process exited normally • **`WEXITSTATUS (status)`** returns exit status of the child process
WIFSIGNALED (status)	• This macro returns true if child process is killed by a signal • **`WTERMSIG(status)`** returns the number of signal that killed the process • **`WCOREDUMP(status)`** returns a non-zero value if the child process created a core dump file
WIFSTOPPED (status)	• This macro returns true if child process is stopped by a signal • **`WSTOPSIG(status)`** returns the number of signal that stopped the process
WIFCONTINUED (status)	• This macro returns true if child process was resumed by SIGCONT

With the passage of time, UNIX designers have added a number of variants of the `wait()` system call, like `waitpid()`, `waitid()`, `wait3()`, `wait4()`...

```
pid_t waitpid(pid_t pid, int* status,int options);
```

The **pid** argument enables the selection of the child to be waited for:

- **If pid > 0 :** waits for the child whose PID equals the value of pid
- **If pid == -1:** waits for any child

```
wait(&status) <=> waitpid(-1, &status, 0)
```

- **If pid == 0 :** waits for any child process whose process Group ID is the same as the calling/parent process
- **If pid < -1:** waits for any child process whose process Group ID equals the absolute value of **pid** argument

----------Interrupts----------

An interrupt is an event that alters the normal execution flow of a program and can be generated by hardware devices or even by the CPU itself. When in interrupt occurs the current flow of execution is suspended, and interrupt handler runs. After the interrupt handler runs the previous execution flow is resumed.

Synchronous: Exception: Divide by zero or a system call are examples of exceptions.

processor detected:

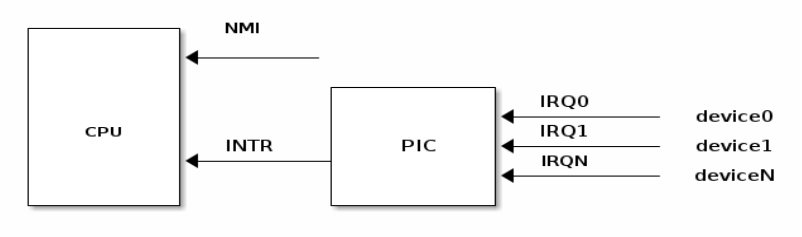
faults:A fault is a type of exception that is reported before the execution of the instruction and can be usually corrected**(page fault)**

traps: A trap is a type of exception that is reported after the execution of the instruction in which the exception was detected

Asynchronous: interrupts, are external events generated by I/O devices. For example, a network card generates an interrupts to signal that a packet has arrived.

Maskable: can be ignored and signalled via INT pin

Non-maskable: cannot be ignored and signalled via NMI pin



A device supporting interrupts has an output pin used for signalling an Interrupt ReQuest. IRQ pins are connected to a device named Programmable Interrupt Controller (PIC) which is connected to CPU's INTR pin.

A PIC usually has a set of ports used to exchange information with the CPU. When a device connected to one of the PIC's IRQ lines needs CPU attention the following flow happens:

1. device raises an interrupt on the corresponding IRQn pin
2. PIC converts the IRQ into a vector number and writes it to a port for CPU to read
3. PIC raises an interrupt on CPU INTR pin
4. PIC waits for CPU to acknowledge an interrupt before raising another interrupt
5. CPU acknowledges the interrupt then it starts handling the interrupt

Once the interrupt is acknowledged by the CPU the interrupt controller can request another interrupt, regardless if the CPU finished handled the previous interrupt or not. Thus, depending on how the OS controls the CPU it is possible to have nested interrupts.

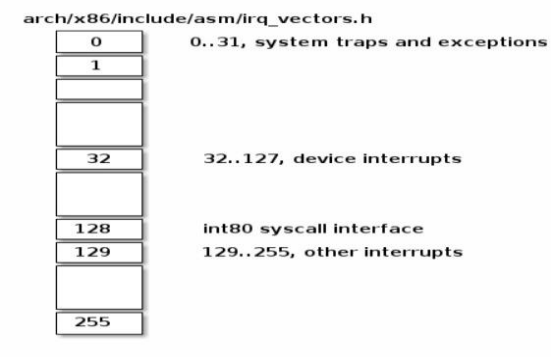
Handling an interrupt request:

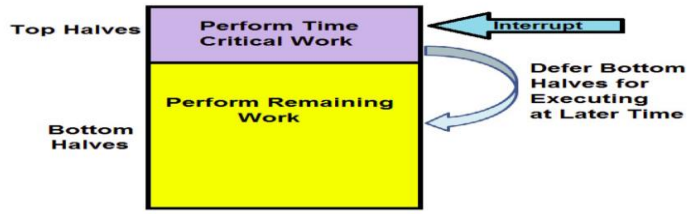
After an interrupt request has been generated the processor runs a sequence of events that eventually ends up with running the kernel interrupt handler:

- CPU checks the current privilege level, if need to change privilege level
- change stack with the one associated with new privilege
- save old stack information on the new stack
- save EFLAGS, CS, EIP on stack
- save error code on stack in case of an abort
- execute the kernel interrupt handler

Interrupts can be checked in systems: *proc*/interrupts

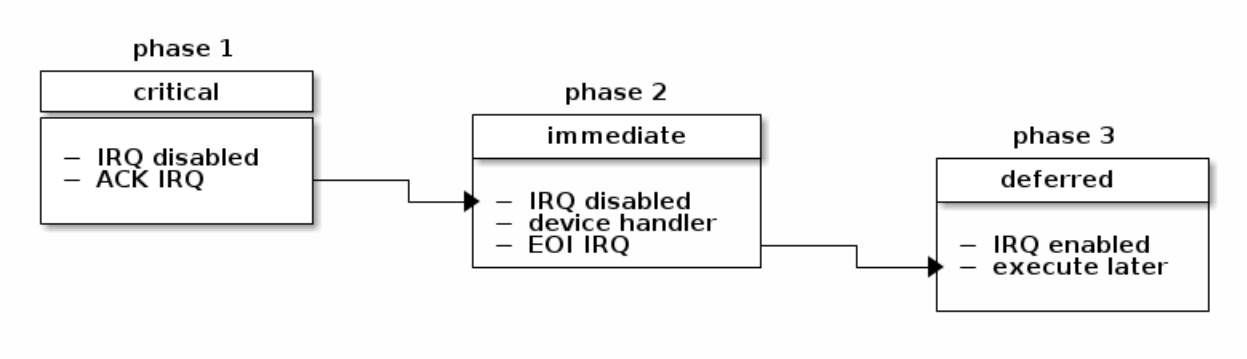
Interrupt Descriptor Table:





Top Halves and Bottom Halves
Limitations On interrupt handler: -
1) It runs asynchronously by interrupting the other code.
2)All interrupt on the current processor disabled.
3)Interrupts are often time critical as they deal with hardware.
4) We cannot block interrupt handler as they run in interrupt context.
Interrupt handling is divided into two parts:
1) **Top Halves**: - It is executed as immediate response to interrupt.
2) **Bottom Halves**: - It is executed sometime later when CPU get free time.

Top Halves: - Top halves executes as soon as CPU receives the interrupt. Following work are generally performed in top halves
1) Acknowledgment of receiving the interrupt
2) copy if some data is received
3) if the work is sensitive needs to perform in top halves.
4)If the work is related to hardware needs to perform in top halves.
5)If the work needs to be ensure that another interrupt does not interrupt it , should be perform in interrupt handler.



Deferrable actions:
Deferrable actions are used to run later. If deferrable actions scheduled from an interrupt handler, the associated callback function will run after the interrupt handler has completed.

There are two large categories of deferrable actions: those that run in interrupt context and those that run in process context.

The purpose of interrupt context deferrable actions is to avoid doing too much work in the interrupt handler function. Running for too long with interrupts disabled can have undesired effects such as increased latency or poor system performance due to missing other interrupts (e.g. dropping network packets because the CPU did not react in time to dequeue packets from the network interface and the network card buffer is full).

Soft IRQs: Soft IRQs is the term used for the low-level mechanism that implements deferring work from interrupt handlers but that still runs in interrupt context.

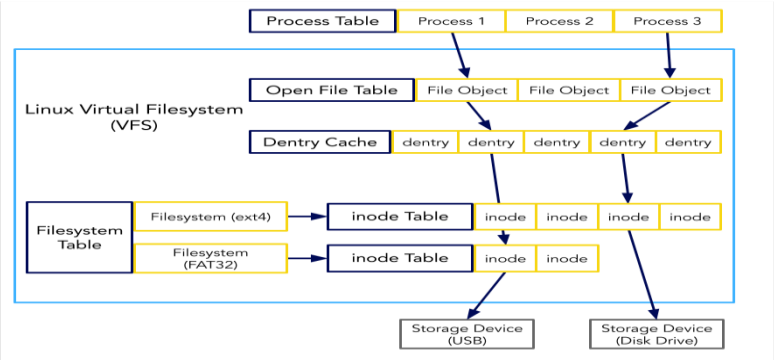
Soft IRQ APIs:
initialize: open_softirq()
activation: raise_softirq()
masking: local_bh_disable(), local_bh_enable()
Once activated, the callback function do_softirq() runs either: after an interrupt handler or from the ksoftirqd kernel thread

Tasklets: Tasklets are a dynamic type (not limited to a fixed number) of deferred work running in interrupt context.
Tasklets API:
initialization: tasklet_init()
activation: tasklet_schedule()
masking: tasklet_disable(), tasklet_enable()

Workqueues: Workqueues are a type of deferred work that runs in process context.
They are implemented on top of kernel threads.
Workqueues API:
init: INIT_WORK
activation: schedule_work()

---------- VFS: ----------

The VFS is sandwiched between two layers: the upper and the lower. The upper layer is the system call layer where a user space process traps into the kernel to request a service (which is usually accomplished via libc wrapper functions) -- thus catalyzing the VFS's processes. The lower layer is a set of function pointers, one set per filesystem implementation, which the VFS calls when it needs an action performed that requires information specific to a particular filesystem.



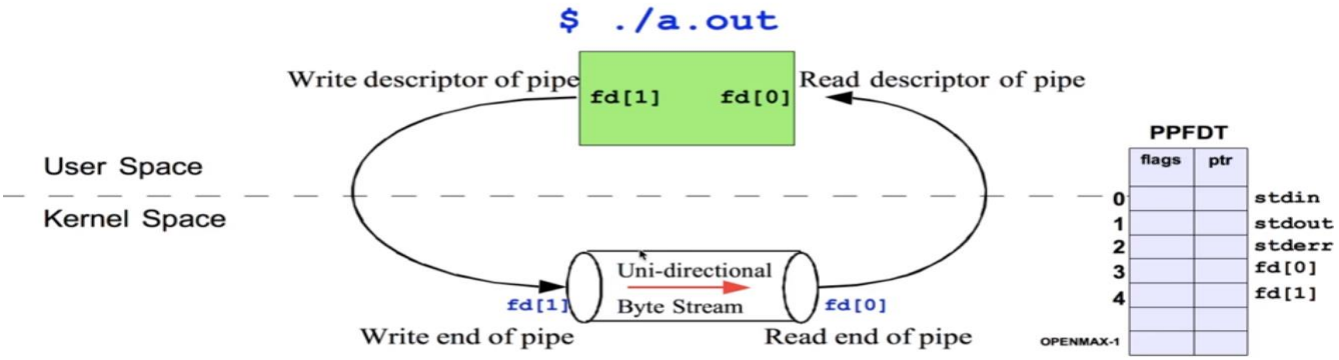
There are four important VFS objects:

1) **Superblock object**: represents a specific mounted filesystem.
2) **Inode object:** represents a specific file.
3) **Dentry object**: represents a directory entry, a single component of a path.
4) **File object:** represents an open file as associated with a process.

----------IPCS ----------

Unnamed pipes: pipe is created inside kernel space and it's unidirectional.

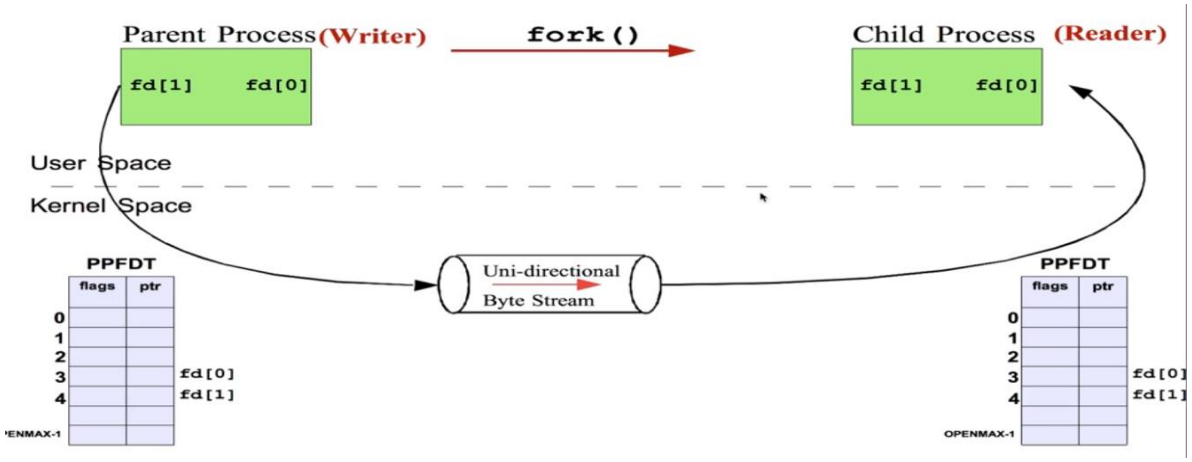
```
int fd[2];
pipe(fd);
int cw = write(fd[1],msg, strlen(msg));
int cr = read(fd[0], buf, cw);
write(1, buf, cr);
```



Ps -a | grep 1234 => example of unnamed pipe.

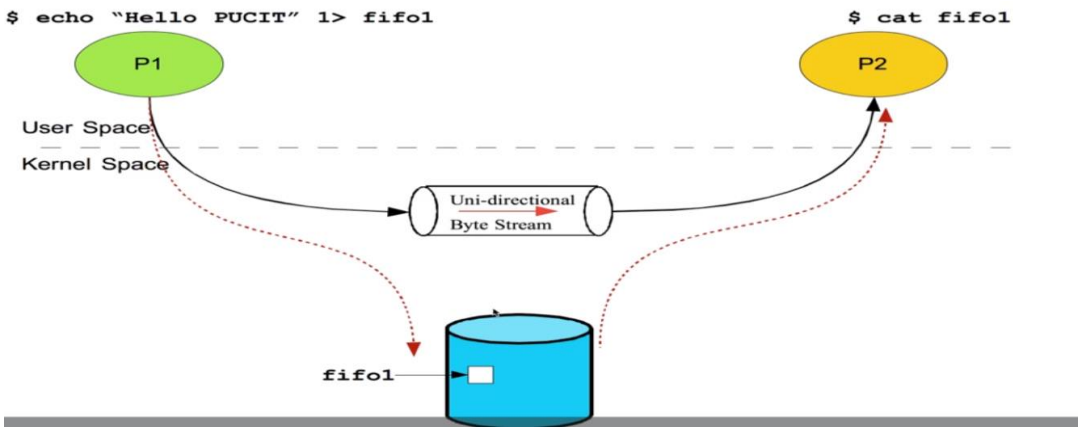
Unnamed pipes b/w parent and child:

Parent process is writer and child process is reader.



Named Pipes/FIFO:

- Pipes have no names, and their biggest disadvantage is that they can be only used between processes that have a parent process in common (ignoring descriptor passing)
- UNIX FIFO is similar to a pipe, as it is a one way (half duplex) flow of data. But unlike pipes a FIFO has a path name associated with it allowing unrelated processes to access a single pipe
- FIFOs/named pipes are used for communication between related or unrelated processes executing on the same machine
- A FIFO is created by one process and can be opened by multiple processes for reading or writing. When processes are reading or writing data via FIFO, kernel passes all data internally without writing it to the file system. Thus a FIFO file has no contents on the file system; the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system



When process writes into fifo file, kernel redirects writing to kernel buffer. When other process reads fifo file from hard disk, kernel redirects reading from pipe. Therefore, FIFO pipe is inside kernel memory having name.

Message Queues: **Message Queue:** A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget(). New messages are added to the end of a queue by msgsnd().
//Create a message queue or connect to an already existing message queue (msgget())
//Write into message queue (msgsnd())
//Read from the message queue (msgrcv())

//Perform control operations on the message queue (msgctl())

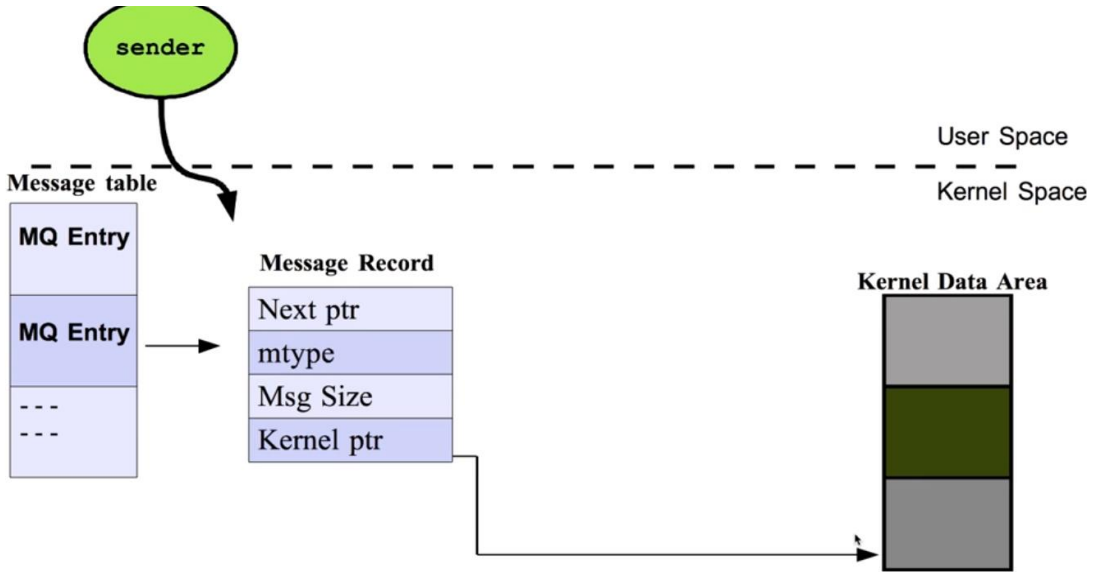
- **Message Queues** can be used to pass messages between related or unrelated processes executing on same machine. Message queues are somewhat like pipes and fifos, but differ in two important aspects:
  - **First**, message boundaries are preserved, so that readers and writers communicate in units of messages
  - **Second**, message queues are kernel persistent
- A Message Queue can be thought of as a linked list of messages in the kernel space. Processes with adequate permissions can put messages onto the queue & processes with adequate permissions can remove messages from the queue

Difference b/w FIFO and Message queues:

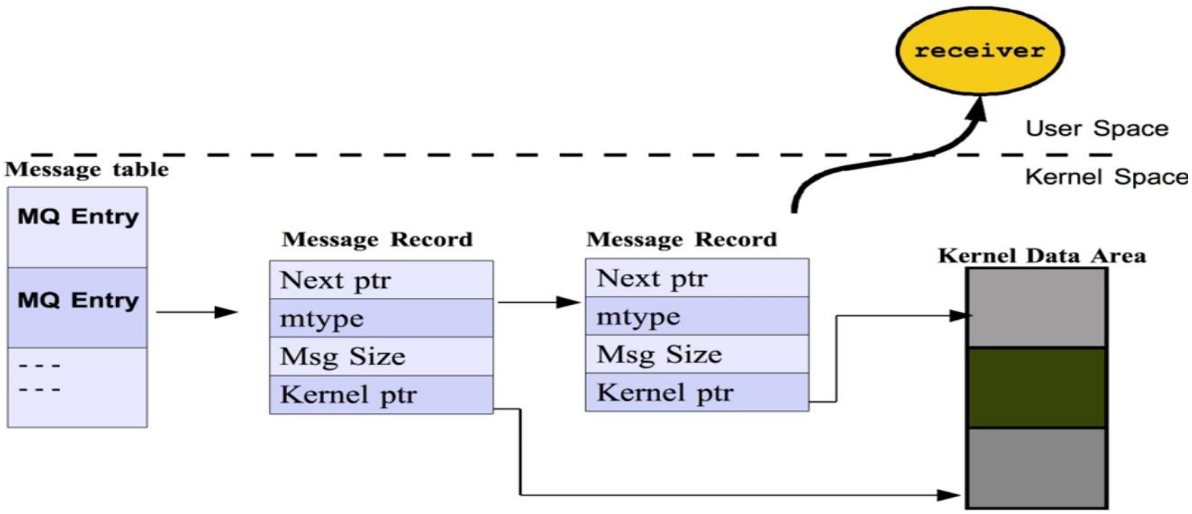
- Message Queues have Kernel persistence while FIFOS have process persistence
- In pipes, a process read or write stream of bytes, while in message queues a process read or write a complete delimited message; it is not possible to read a partial message leaving rest behind in IPC object
- In pipes a write makes no sense unless a reader also exists. In message queues there is no requirement that some reader must be waiting before a process writes a message to the queue
- Message queues are priority driven. Queue always remains sorted with the oldest message of the highest priority at front
- A process can determine the status of a message queue

Sender is sending data over message queue:

Kernel maintains table for each message queue in system. Sender process uses messageget(), now entry will be created for message queue in message table. Sender calls messageSend(), kernel creates entry in message queue and node contains pointer to next node. This node is linked list node. Kernelptr points to kernel data area. Kernel data area contains actual message copied from sender's virtual address space.



Receiver receives message: Other process call messageget() and call messageReceive(), kernel copies data to receiver's virtual address space.



Shared Memory:

shared memory: Two or more process can access the common memory and communication is done via this shared memory where changes made by one process can be viewed by another process. The problem with pipes, fifo and message queue – is that for two processes to exchange information. The information goes through the kernel. To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques.

//Create the shared memory segment or use an already created shared memory segment (shmget())
//Attach the process to the already created shared memory segment (shmat())
//Detach the process from the already attached shared memory segment (shmdt())
//Control operations on the shared memory segment (shmctl())

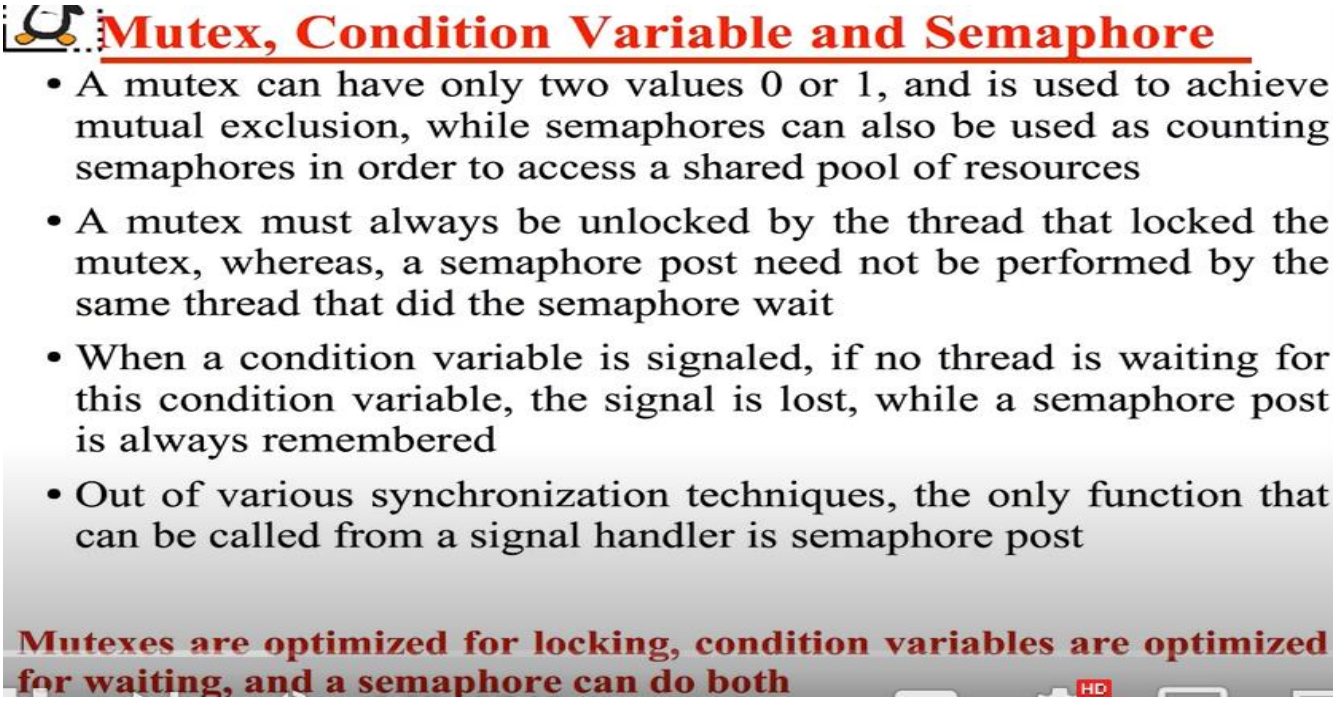
- **Shared Memory** allows two or more processes to share a memory region or segment of memory for reading and writing purposes
- The problem with pipes, fifo and message queue is that mode switches are involved as the data has to pass from one process buffer to the kernel buffer and then to another process buffer
- Since access to user-space memory does not require a mode switch, therefore, shared memory is considered as one of the quickest means of IPC

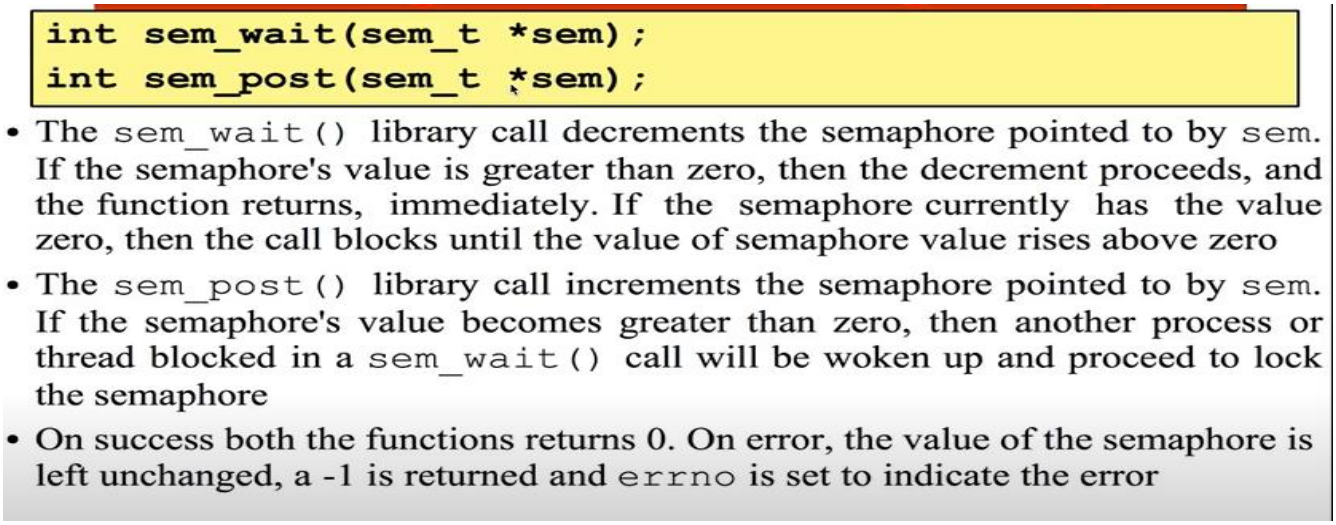
Memory Mapped files:

- A memory-mapped file is mostly a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file on disk
- Memory mapped I/O let us map a file on disk into a buffer in process address space, so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without using `read()` or `write()` system calls
- There are two types of memory mapped files:
  - Persisted / File Mapping
  - Non-Persisted / Anonymous Mapping

Semaphore:

- Semaphores are a kind of generalized locks first defined by Dijkstra in late 1960s. A primitive used to provide synchronization between various processes or between various threads of a process. It can be considered as an integer variable, with three differences:
  - When you create a semaphore, you can initialize it to any integer value, but after that you can perform two operations on it, increment (verhogen, post, signal) and decrement (proberen, wait)
  - When a process/thread decrements the semaphore, if the semaphore currently has the value zero, then the thread blocks until the value of semaphore value rises above zero
  - When a process/thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked. Which one? (strong semaphore, weak semaphores)





Signals: Signals are the interrupts that are sent to the program to specify that an important event has occurred. There are two kinds of signals:

1. Maskable
2. Non-Maskable

Maskable: - Maskable Signals are the signals that the user can change or ignore, for example, Ctrl +C.

Non-Maskable: - Non-Maskable Signals are the signals that the users cannot change or ignore. Non-Maskable signals mainly occur if a user is signaled for non-recoverable hardware errors.

Every process has process descriptor and process descriptor has three fields:
1) Signal pending field
2) Signal masking/unmasking field
3) Signal handler table.

Let's assume process is trying to access invalid logical virtual address and memory exception will be generated. Exception handler will generate SIGSEGV to current process, when system scans for pending signal, this pending signal will be addressed, and appropriate action will be taken.

Default Actions:

- Stop the process.
- Terminate the process
- Continue a stopped process.
- Ignore the signal
- Dump core: This default action generates a file named core which comprise the process's memory image when the process received the signal.

//**Shared_memory vs message queues**: As understood, once the message is received by a process it would be no longer available for any other process. Whereas in shared memory, the data is available for multiple processes to access. Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.

//**Socket programming:** Socket is used in a client-server application framework.
//Socket types: Stream Sockets (TCP/IP): It's reliable protocol and also data integrity is maintained. It's connection oriented. (Transmission control protocol)
//dataGram socket: UDP:It's not reliable.It's connection less.
//**How to make a Server**
//Create a socket with the socket() system call.
//Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
//Listen for connections with the listen() system call.
//Accept a connection with the accept() system call. This call typically blocks the connection until a client connects with the server.
//Send and receive data using the read () and write() system calls.
//**Make client**:
//Create a socket with the socket () system call.
//Connect the socket to the address of the server using the connect () system call.
//Send and receive data. There are a number of ways to do this, but the simplest way is to use the read() and write() system calls.
//Blocking and Non-Blocking Socket I/O:TCP sockets are placed in a blocking mode. This means that the control is not returned to your program until some specific operation is complete. For example, if you call the connect () method, the connection blocks your program until the operation is complete.
//we should make non-blocking calls and can be done by calling socket. setblocking(0)
//TCP/IP nad OSI model:

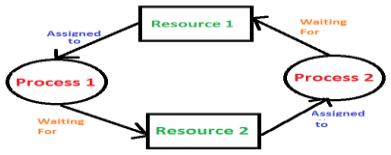
TCP/IP MODEL
Application Layer
Transport Layer
Internet Layer
Network Access Layer

OSI MODEL
Application Layer
Presentation Layer
Session Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer

BASIS FOR COMPARISON	SEMAPHORE	MUTEX
Basic	Semaphore is a signalling mechanism.	Mutex is a locking mechanism.
Existence	Semaphore is an integer variable.	Mutex is an object.
Function	Semaphore allow multiple program threads to access a finite instance of resources.	Mutex allow multiple program thread to access a single resource but not simultaneously.
Ownership	Semaphore value can be changed by any process acquiring or releasing the resource.	Mutex object lock is released only by the process that has acquired the lock on it.
Categorize	Semaphore can be categorized into counting semaphore and binary semaphore.	Mutex is not categorized further.

Semaphore: A semaphore is a hardware or a software tag variable whose value indicates the status of a common resource.
- Its purpose is to lock the common resource being used. A process which needs the resource will check the semaphore to determine the status of the resource followed by the decision for proceeding.
- In multitasking operating systems, the activities are synchronized by using the semaphore techniques is a better option in case there are multiple instances of resources available. In the case of single shared resource mutex is a better choice.

Deadlock: A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Is it possible to have a deadlock involving only one process: No

A deadlock situation can arise if the following four conditions hold simultaneously in a system.
- **Mutual Exclusion:** The resources available are not sharable. This implies that the resources used must be mutually exclusive.
- **Hold and Wait:** Any process requires some resources in order to be executed. In case of insufficient availability of resources, a process can take the available resources, hold them and wait for more resources to be available.
- **No Preemption:** The resources that a process has on hold can only be released by the process itself voluntarily. This resource cannot be preempted by the system.
- **Circular Waiting:** A special type of waiting in which one process is waiting for the resources held by a second process. The second process is in turn waiting for the resources held by the first process.

Methods for handling deadlock:

There are three ways to handle deadlock
1) **Deadlock prevention or avoidance**: The idea is to not let the system into a deadlock state.

One can zoom into each category individually; Prevention is done by negating one of above-mentioned necessary conditions for deadlock.
Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we must assume. We need to ensure that all information about resources which process will need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) to avoid deadlock.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred. (**Killing the process a**nd **Preemption of resource**)

3) Ignore the problem altogether: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Critical section: is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e., only one process can execute in its critical section at a time. All the other processes must wait to execute in their critical sections.

Difference between Hard link and soft link:
Hard Link:
A hard link acts as a copy (mirrored) of the selected file. It accesses the data available in the original file. If the earlier selected file is deleted, the hard link to the file will still contain the data of that file.
Soft Link:
A soft link (also known as Symbolic link) acts as a pointer or a reference to the file name. It does not access the data available in the original file. If the earlier file is deleted, the soft link will be pointing to a file that does not exist anymore.

Ldconfig: ldconfig is used to create, update and remove symbolic
links for the current shared libraries based on the lib directories present in the /etc/ld.so.conf

----------Memory Management----------

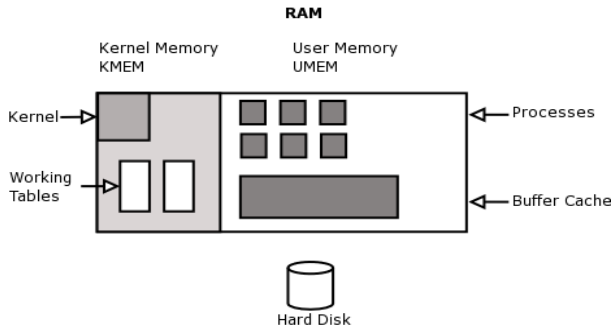
Memory Management: Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each memory location, regardless of either it is allocated to some process, or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Process Address Space:
The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is,2^31 possible numbers, for a total theoretical size of 2 gigabytes. The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated –

- **Symbolic addresses**: The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
- **Relative addresses:** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
- **Physical addresses:** The loader generates these addresses at the time when a program is loaded into main memory.

Memory is divided into two areas, kernel memory and user memory.

Kernel memory is also known as kmem, kernel space and kernel land.
This contains the kernel binary itself, working tables to keep track of status on the system and buffers.
Examples of working tables that the kernel keeps in kernel memory for the operation of the system are the Global Open File Table, the Process Table and the Mount Table.

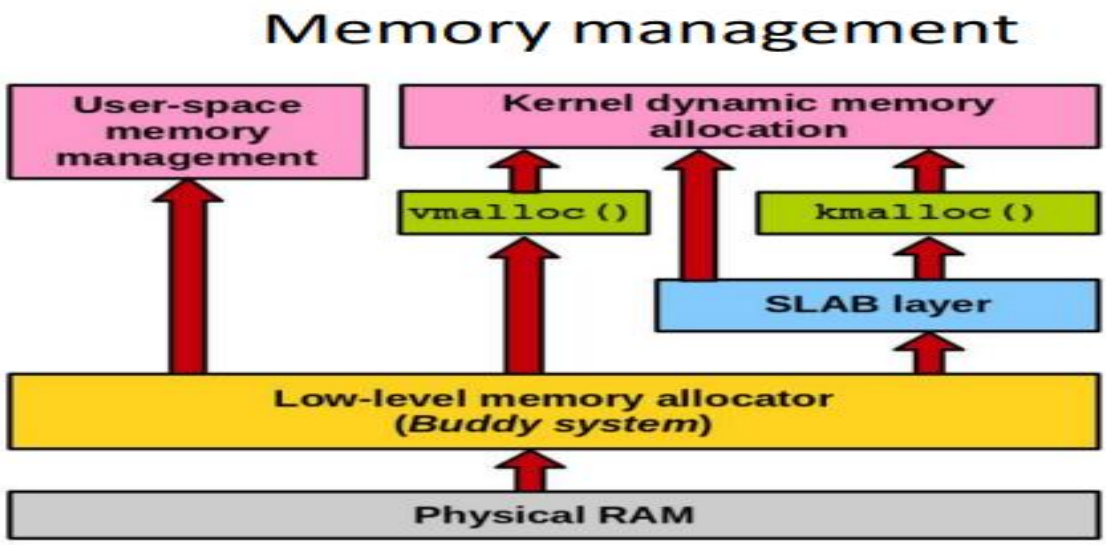


A modern operating system manages physical memory as page frames. A page frame can be allocated to a user process or for the kernel to use for its own purposes, such as to allocate its own data structures.

A kernel data object (e.g. a structure representing an "open file" or something) is typically less than the size of a page. So it makes sense to have a two-level allocation hierarchy: one to manage page frames, and one to allocate data structures inside allocated page frames.

Linux uses a buddy allocator to allocate page frames, and a slab allocator to allocate kernel data structures. When the slab allocator needs more memory, it obtains it from the buddy allocator.

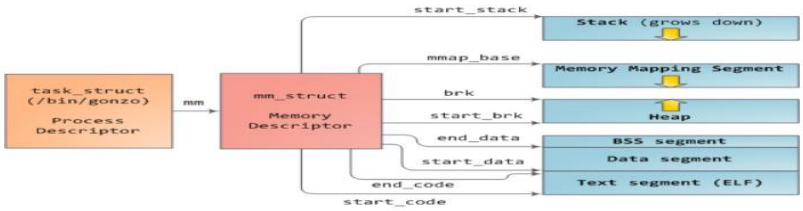
This approach works well for Linux, since it supports different page sizes; x86-64 CPUs support 4kB, 2M, and sometimes 2GB pages, and a binary buddy allocator can support this with very little modification.



Linux memory management is a complex subsystem that deals with:

- Management of the physical memory: allocating and freeing memory
- Management of the virtual memory: paging, swapping, demand paging, copy on write
- User services: user address space management (e.g. mmap(), brk(), shared memory)

- Kernel services: SL*B allocators, vmalloc



Why Memory Management is required:

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

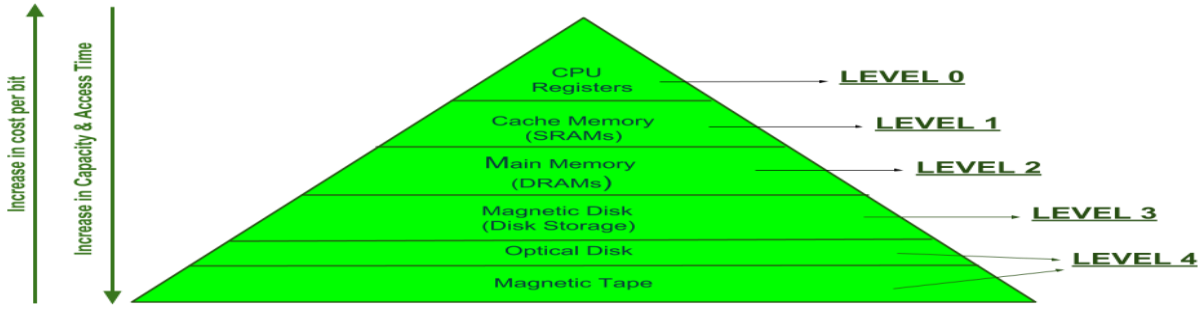
Levels of memory:

Level 1 or Register –
It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, Program counter, address register etc.

Level 2 or Cache memory –
It is the fastest memory which has faster access time where data is temporarily stored for faster access.

Level 3 or Main Memory –
It is memory on which computer works currently. It is small and once power is off data no longer stays in this memory.

Level 4 or Secondary Memory –
It is external memory, which is not as fast as main memory, but data stays permanently in this memory.

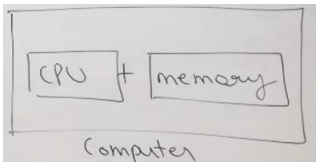


MEMORY HIERARCHY DESIGN

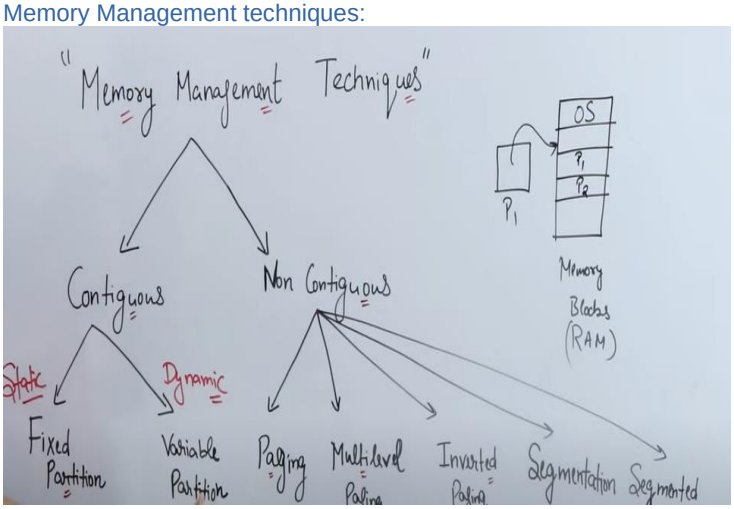
Cache Memory: is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently re-quested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various independent caches in a CPU, which store instructions and data.

Cache Performance:
When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. If the processor finds that the memory location is in the cache, a cache hit has occurred, and data is read from cache. If the processor does not find the memory location in the cache, a cache miss has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.



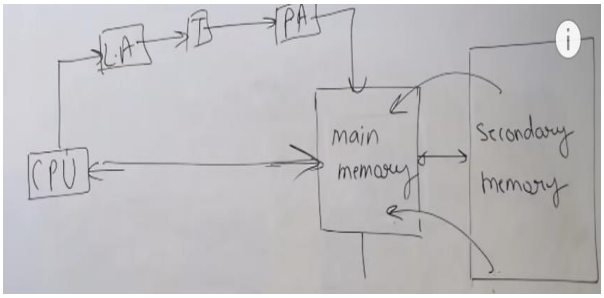
Memory Management techniques:



3 things are important when we talk about memory management:

1) Size (should be more to store more data)
2) Access time (should be less)
3) Per unit cost (should be less)

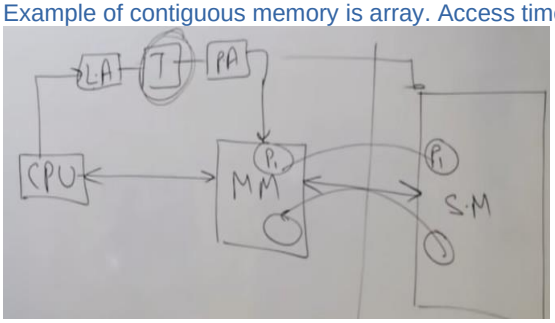
Locality of reference: is the tendency of a processor to access the same set of memory locations repetitively over a short period of time. Since Access time of Main memory is faster than secondary memory, we move process from secondary memory to main memory. CPU generates logical address, but main memory understands physical address, so address translation mechanism is required. CPU interacts with main memory, but main memory interacts with secondary memory in back ground.



Contiguous vs Noncontiguous allocation policy:

Contiguous memory allocation policy:

Example of contiguous memory is array. Access time will be fast.



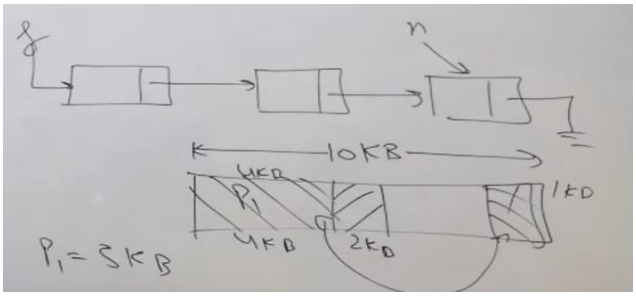
Problems with contiguous memory:

Suppose you want to run process p1 and you load it from secondary memory to main memory. If you load it in continuous fashion, then there will be problem of external fragmentation. We want to allocate memory of 5kb for process p1. We have memory of 5 kb available but not contiguously so cannot allocate 5 kb process. This problem is called **external fragmentation.**

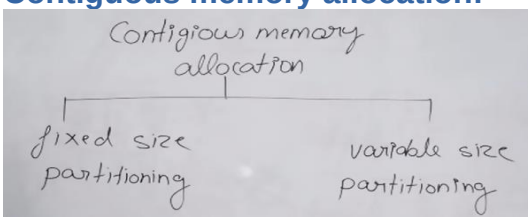
Fragmentation: As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Non-Contiguous memory allocation policy:

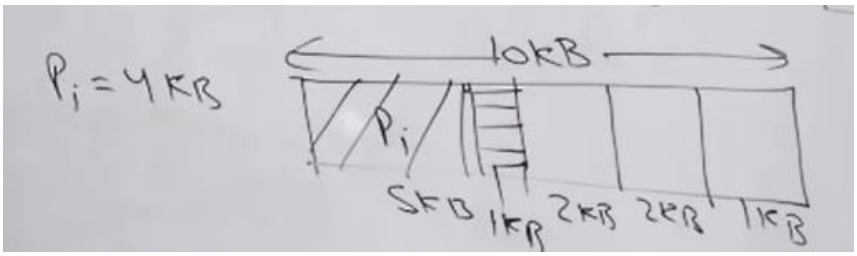
For example, linked list is example of non-contiguous policy. There will not be problem of external fragmentation. If we want to search nth node then we need to traverse it in linear way one by one so accessing is not fast.



Contiguous memory allocation:



Fixed size partioning: Memory is partitioned into fixed size, it does not mean that Every partitioned size would be same but size will be partitioned before memory allocation.



Problem with fixed size partioning:

If we want to allocate memory to 4k process, then will assign 5kb block of memory. There will be waste of 1 kb memory that cannot be assigned to any other process. This problem is called internal fragmentation.

Internal fragmentation:

Internal fragmentation occurs when memory blocks are allocated to the process more than their requested size. Due to this some unused space is leftover and creates an internal fragmentation problem.

Example: Suppose there is a fixed partitioning is used for memory allocation and the different size of block 3MB, 6MB, and 7MB space in memory. Now a new process p4 of size 2MB comes and demand for the block of memory. It gets a memory block of 3MB but 1MB block memory is a waste, and it cannot be allocated to other processes too. This is called internal fragmentation.

Example of internal fragmentation:

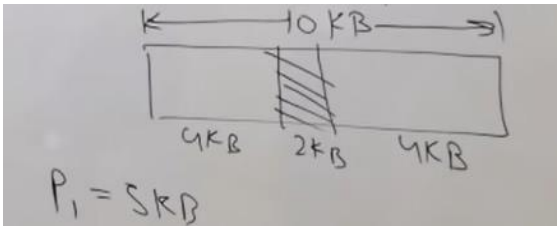
text book: Last page of story is not filled but new story is started from new page.

Variable size partioning:

allocate space to process when it comes for allocation and predefined partioning is not done.

Problem with Variable size partioning: **External fragmentation:**

In external fragmentation, we have a free memory space, but we cannot assign it to process because blocks are not contiguous.



Example: Suppose (consider above example) three process p1, p2, p3 comes with size 2MB, 4MB, and 7MB respectively. Now they get memory blocks of size 3MB, 6MB, and 7MB allocated respectively. After allocating process p1 process and p2 process left 1MB and 2MB. Suppose a new process p4 comes and demands a 3MB block of memory, which is available, but we cannot assign it because free memory space is not contiguous. This is called external fragmentation.

Solution of external fragmentation: Compaction

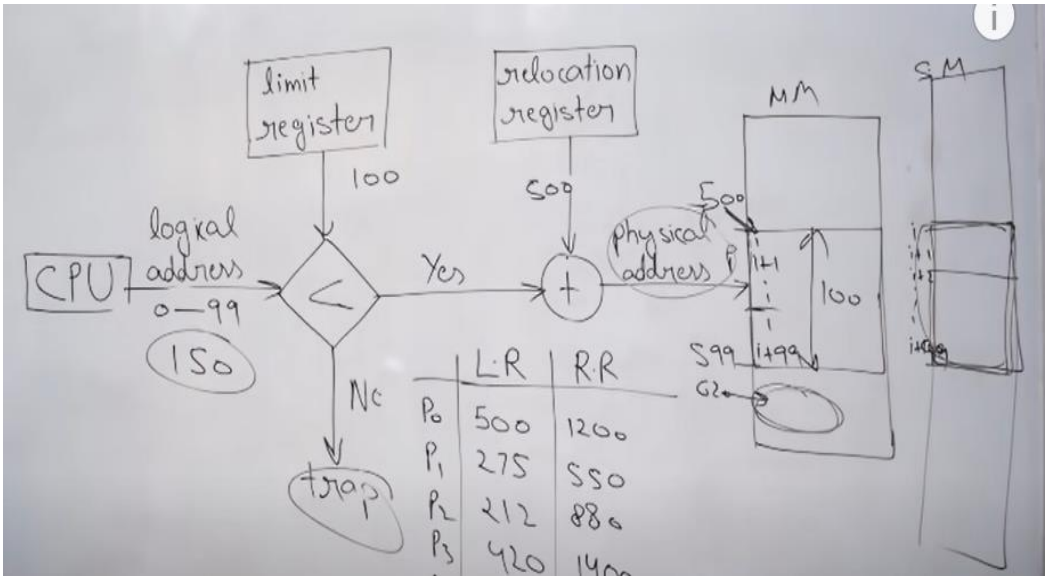
Compaction: is a process in which the free space is collected in a large memory chunk to make some space available for processes.

Address Translation is case of contiguous memory:

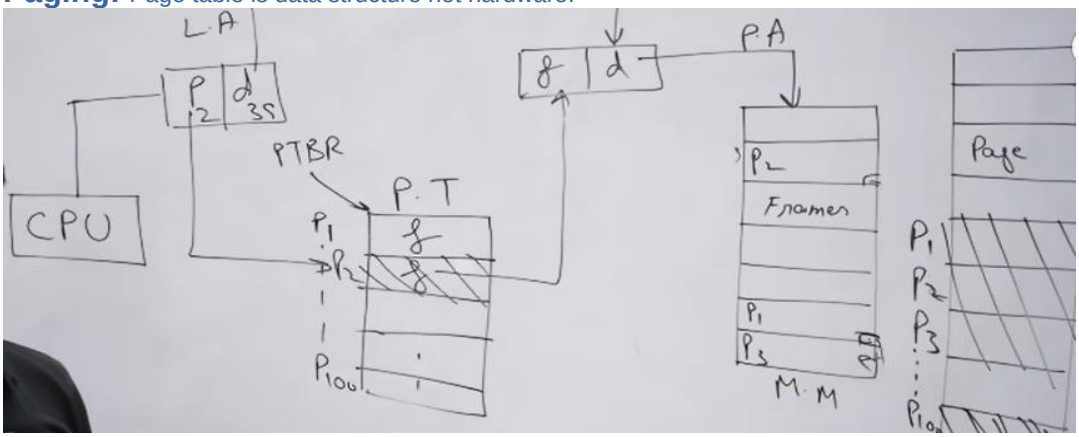
Limit register: it is the index values for instructions for every process.

Relocation register: is responsible for base address of process in main memory.

Ex: Process p0 has base address 500 and index value 100. Relocation register value will be 500 and limit register value will be 100.



Paging: Page table is data structure not hardware.



CPU generates logical address which contains **page no. & instruction offset**. With this page no, we access page table. Page table has PTBR (page table base register), holds the value of page tables which is stored in PCB. Now page 2 of page table is accessed, page table contains page frames of main memory. Now we can access the page frames of main memory. We don't need to map instruction offset as it would be same for each process.
We must remember the base address of each frame of main memory for that process.

Paging: Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non-contiguous(dividing fixed size pages)

- Logical Address or Virtual Address (represented in bits): An address generated by the CPU
- Logical Address Space or Virtual Address Space (represented in words or bytes): The set of all logical addresses generated by a program
- Physical Address (represented in bits): An address available on a memory unit
- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses

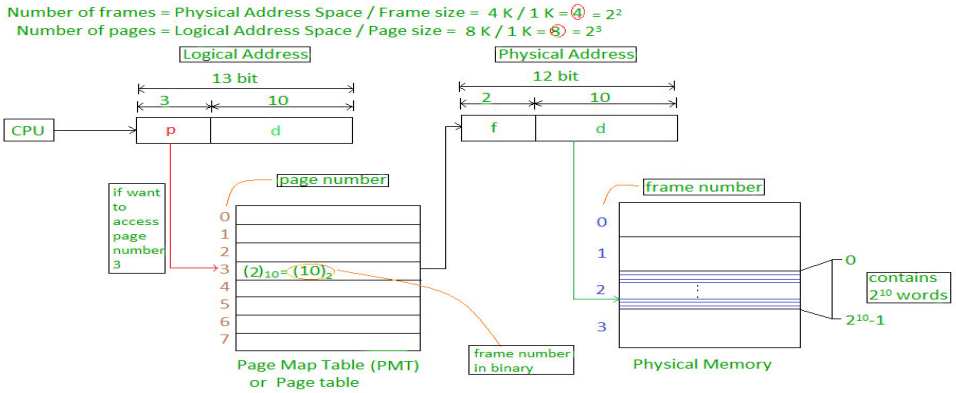
Example:

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device, and this mapping is known as the paging technique.

- The Physical Address Space is conceptually divided into several fixed-size blocks, called frames.
- The Logical Address Space is also split into fixed-size blocks, called pages.
- Page Size = Frame Size

Let us consider an example:

- Physical Address = 12 bits, then Physical Address Space = 4 K words (2^12)
- Logical Address = 13 bits, then Logical Address Space = 8 K words (2^13)
- Page size = frame size = 1 K words (assumption)



The address generated by the CPU is divided into:

- **Page number(p):** Number of bits required to represent the pages in Logical Address Space or Page number
- **Page offset(d):** Number of bits required to represent a particular word in a page or page size of Logical Address Space or word number of a page or page offset.

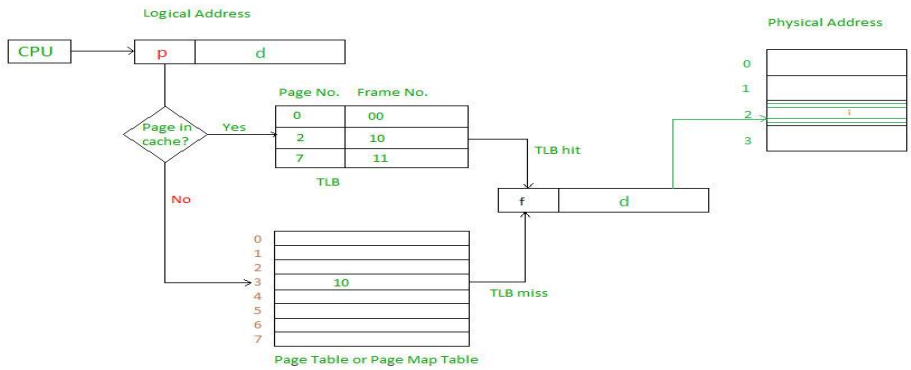
Physical Address is divided into:

- **Frame number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number frame
- **Frame offset(d):** Number of bits required to represent a particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.
  The hardware implementation of the page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if the page table is small. If the page table contains many entries, then we can use TLB (translation Look-aside buffer), a special, small, fast look-up hardware cache.

Now we need to further improve the access time taken using page table data structure. If we every time access same page from page tables, will eb waste of time. We have hardware solution called TLB.

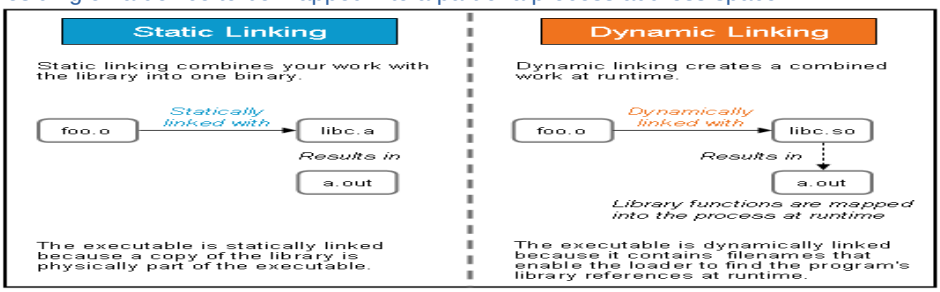
Translation Lookaside Buffer (i.e. TLB): In short, TLB speeds up the translation of virtual addresses to a physical address by storing page-table in faster memory. In fact, TLB also sits between CPU and Main memory.

- The TLB is an associative, high-speed memory.
- Each entry in TLB consists of two parts: a tag and a value.
- When this memory is used, then an item is compared with all tags simultaneously. If the item is found, then the corresponding value is returned.



Reentrant Kernels:
All Unix kernels are reentrant: this means that several processes may be executing in Kernel Mode at the same time.

Process Address Space: Each process runs in its private address space. A process running in User Mode refers to private stack, data, and code areas. Linux supports the mmap( ) system call, which allows part of a file or the memory residing on a device to be mapped into a part of a process address space.



Whenever there is change in static lib, code must be recompiled as well but it is secured no other app can corrupt it.
While dynamic lib is not part of executable but only one copy can be shared by multiple apps.
To create a dynamic library, write the following command:

```
gcc -g -fPIC -Wall -Werror -Wextra -pedantic *.c -shared -o liball.so
```

The **-fPIC** flag allows the following code to be referenced at any virtual address at runtime. It stands for Position Independent Code.

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Static library creation:

```
ar rc liball.a dog.o cat.o bird.o
ar is for archieving and -rc is for replace and create….
```

Logical and Physical Address Space: Logical Address space: An address generated by the CPU is known as “Logical Address”. It is also known as a Virtual address. Logical address space can be defined as the size of the process. A logical address can be changed.

Physical Address space: An address seen by the memory unit (i.e the one loaded into the memory address register of the memory) is commonly known as a “Physical Address”. A Physical address is also known as a Real address. The set of all physical addresses corresponding to these logical addresses is known as Physical address space. A physical address is computed by MMU. The run-time mapping from virtual to physical addresses is done by a hardware device Memory Management Unit (MMU). The physical address always remains constant.

Static and Dynamic Loading: To load a process into the main memory is done by a loader. There are two different types of loading:

Static loading: - loading the entire program into a fixed address. It requires more memory space.

- **Dynamic loading**: - The entire program and all data of a process must be in physical memory for the process to execute. So, the size of a process is limited to the size of physical memory. To gain proper memory utilization, dynamic loading is used. In dynamic loading, a routine is not loaded until it is called. All routines are residing on disk in a relocatable load format. One of the advantages of dynamic loading is that unused routine is never loaded. This loading is useful when a large amount of code is needed to handle it efficiently.

Static and Dynamic linking: To perform a linking task a linker is used. A linker is a program that takes one or more object files generated by a compiler and combines them into a single executable file.

Static linking: In static linking, the linker combines all necessary program modules into a single executable program. So there is no runtime dependency. Some operating systems support only static linking, in which system language libraries are treated like any other object module.

- **Dynamic linking**: The basic concept of dynamic linking is like dynamic loading. In dynamic linking, "Stub" is included for each appropriate library routine reference. A stub is a small piece of code. When the stub is executed, it checks whether the needed routine is already in memory or not. If not available, then the program loads the routine into memory.

Swapping: Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes. It is used to improve main memory utilization. Swapping is also known as roll-out, roll in, because if a higher priority process arrives and wants service, the memory manager can swap out the lower priority process and then load and execute the higher priority process. After finishing higher priority work, the lower priority process swapped back in memory and continued to the execution process.

Memory Allocation:

Main memory usually has two partitions −

- Low Memory − Operating system resides in this memory.
- High Memory − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.

2	**Multiple-partition allocation** In this type of allocation, main memory is divided into several fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Both the first fit and best-fit systems for memory allocation affected by external fragmentation. To overcome the external fragmentation problem Compaction is used. In the compaction technique, all free memory space combines and makes one large block. So, this space can be used by other processes effectively.

Another possible solution to the external fragmentation is to allow the logical address space of the processes to be noncontiguous, thus permit a process to be allocated physical memory wherever the latter is available.

Thrashing: - In virtual memory system, thrashing is a high page fault scenario. It occurs due to under-allocation of pages required by a process. The system becomes extremely slow due to thrashing leading to poor performance.

Belady's anomaly occur?

The Belady's anomaly is a situation in which the number of page faults increases when additional physical memory is added to a system.

Advantages and Disadvantages of Paging:

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

- **Segmentation:** is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is a different logical address space of the program. When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory. Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size
- **Paging vs Segmentation:**

Virtual memory:

A computer can address more memory than the amount physically installed on the system. This extra memory is called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

Virtual memory advantages:

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging:

A demand paging system is like a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program’s pages out to the disk or any of the new program’s pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program’s pages as they are referenced.

Advantages:

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

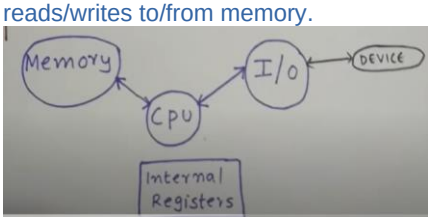
Disadvantages:
Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques

---

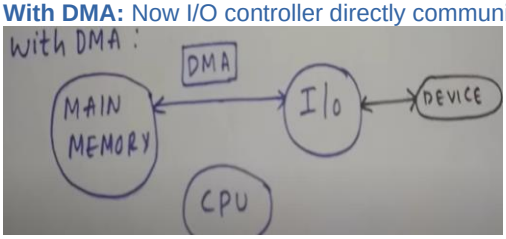
Direct Memory Access (DMA):

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So, a typical computer uses direct memory access (DMA) hardware to reduce this overhead. Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Without DMA: Device will send data to I/O controller and I/O controller will communicate with CPU then CPU reads/writes to/from memory.



With DMA: Now I/O controller directly communicate with memory**.**



Preemptive kernel:
A preemptive kernel is one that can be interrupted in the middle of executing code - for instance in response for a system call - to do other things and run other threads, possibly those that are not in the kernel.

I2C Protocol: I2C stands for Inter-Integrated Circuit. I2C is a simple two-wire serial protocol used to communicate between two devices or chips in an embedded system. I2C has two lines SCL and SDA, SCL is used for clock and SDA is used for data.
SPI Protocol: SPI stands for Serial Peripheral interface. SPI is a four wire serial communication protocol. SPI follows master-slave architecture. The four lines of SPI are MOSI, MISO, SCL and SS. SCL is a serial clock that is used for entire data communication. Slave Select(SS) is used to select the slave. Master out Slave In (MOSI) is the output data line from the master and Master in Slave out(MISO) is the input data line for the Master.

What is the difference between I2C vs SPI?

I2C is half duplex communication and SPI is full duplex communication.
I2C supports multi master and multi slave and SPI supports single master.
I2C is a two wire protocol and SPI is a four wire protocol.

Operating System - I/O Hardware: One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.
An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories

Block devices – A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.

Character devices: A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

Device Controllers: Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.
The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller. There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction, as necessary.
Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.

Synchronous vs asynchronous I/O:
Synchronous I/O – In this scheme CPU execution waits while I/O proceeds
Asynchronous I/O – I/O proceeds concurrently with CPU execution

Communication to I/O Devices:
The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device:

- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O: This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.
Memory-mapped I/O: When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.
While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.
The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

Device Management: An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management: A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources. Allocates the resources.
- De-allocates the resources.

Device driver programming

Can we load module statically?
Yes, It needs a bit of hacking to move into kernel source tree. Tweak the Makefile or Kconfig so that the code is built in. and then rebuild kernel image. For ex: you move code to drivers/ directory and update Makefile of drivers.

How to configure and install kernel:

1) Download the kernel source code.
2) Extract the Source Code: tar xvf linux-5.9.6.tar.xz
3) Configure Kernel:
   - Navigate to the code directory using the cd command (cd linux-5.9.6)
   - Copy the existing configuration file using the cp command (cp -v /boot/config-$(uname -r) .config)
   - To make changes to the configuration file, run the make command (make menuconfig)
4) Build the Kernel (make -j 4)
5) Install the required modules with this command (sudo make modules_install)
6) Install the kernel (sudo make install)
7)Update initramfs
8)Update grub.

Check installed kernel version: uname -r
check installed kernel date: uname -v
Kernel is installed in path: /lib/modules/$(uname-r)

lsmod – List Modules that Loaded Already

lsmod command will list modules that are already loaded in the kernel as shown beblow.

```
# lsmod
Module                  Size  Used by
ppp_deflate            12806  0
zlib_deflate           26445  1 ppp_deflate
bsd_comp               12785  0
..
```

insmod – Insert Module into Kernel

insmod command will insert a new module into the kernel as shown below.

```
# insmod /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko

# lsmod | grep "squash"
squashfs                35834  0
```

modinfo – Display Module Info

modinfo command will display information about a kernel module as shown below.

```
# modinfo /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko

filename:       /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko
license:        GPL
author:         Phillip Lougher
description:    squashfs 4.0, a compressed read-only filesystem
srcversion:     89B46A0667BD5F2494C4C72
depends:
intree:         Y
vermagic:       3.5.0-19-generic SMP mod_unload modversions 686
```

4. rmmod – Remove Module from Kernel

rmmod command will remove a module from the kernel. You cannot remove a module which is already used by any program.

```
# rmmod squashfs.ko
```

5. modprobe – Add or Remove modules from the kernel

modprobe is an intelligent command which will load/unload modules based on the dependency between modules. Refer to modprobe commands for more detailed examples.

II. Write a Simple Hello World Kernel Module

1. Installing the linux headers

You need to install the linux-headers-.. first as shown below. Depending on your distro, use apt-get or yum.

```
# apt-get install build-essential linux-headers-$(uname -r)
```

2. Hello World Module Source Code

Next, create the following hello.c module in C programming language.

```
#include <linux/module.h>    // included for all kernel modules
#include <linux/kernel.h>    // included for KERN_INFO
#include <linux/init.h>      // included for __init and __exit macros

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ankit");
MODULE_DESCRIPTION("A Simple Hello World module");

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello world!\n"); //printing to kernel buffer.
    return 0;    // Non-zero return means that the module couldn't be loaded.
```

```
}

static void __exit hello_cleanup(void)
{
    printk(KERN_INFO "Cleaning up module.\n"); //printing to kernel buffer.
}

module_init(hello_init);
module_exit(hello_cleanup);
```

Warning: All kernel modules will operate on kernel space, a highly privileged mode. So be careful with what you write in a kernel module.

3. Create Makefile to Compile Kernel Module

The following makefile can be used to compile the above basic hello world kernel module.

```
obj-m += hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Use the make command to compile hello world kernel module as shown below.

```
# make

make -C /lib/modules/3.5.0-19-generic/build M=/home/lakshmanan/a modules
make[1]: Entering directory `/usr/src/linux-headers-3.5.0-19-generic'
  CC [M]  /home/lakshmanan/a/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/lakshmanan/a/hello.mod.o
  LD [M]  /home/lakshmanan/a/hello.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.5.0-19-generic'
```

The above will create hello.ko file, which is our sample Kernel module.

4. Insert or Remove the Sample Kernel Module

Now that we have our hello.ko file, we can insert this module to the kernel by using insmod command as shown below.

```
# insmod hello.ko

# dmesg | tail -1
[ 8394.731865] Hello world!

# rmmod hello.ko
```

```
# dmesg | tail -1
[ 8707.989819] Cleaning up module.
```

When a module is inserted into the kernel, the **module_init** macro will be invoked, which will call the function hello_init. Similarly, when the module is removed with rmmod, **module_exit** macro will be invoked, which will call the hello_exit. Using dmesg command, we can see the output from the sample Kernel module.

Output file generated when we compile our module:

modules.order

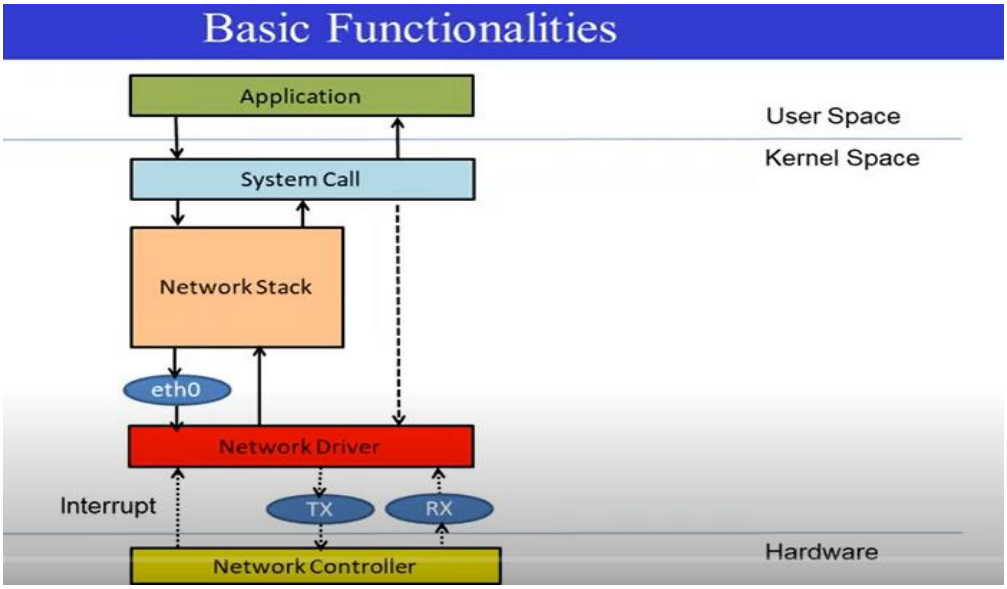
```
-------------------------------------------------
This file records the order in which modules appear in Makefiles. This
is used by modprobe to deterministically resolve aliases that match
multiple modules.

-------------------------------------------------------------------------

If the files are present the source file is compiled to a "modulname.o", and
"modulename.mod.c" is created which is compiled to "modulename.mod.o".
The modulename.mod.c is a file that basically contains the information about
the module (Version information etc).
The modulename.o and the modulename.mod.o are linked together by modpost in
the next stage to create the "modulename.ko" .
```

Network Device driver stack:

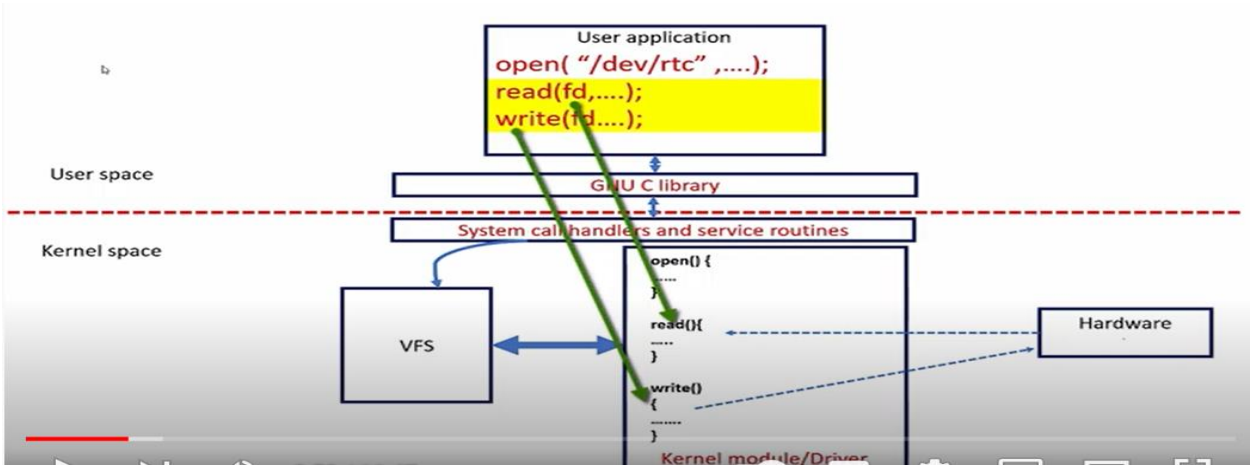
Struct net_dvice is used for network device.

Device Allocation: alloc_netdev()

Device Registration: register_netdevice()



Character device driver:



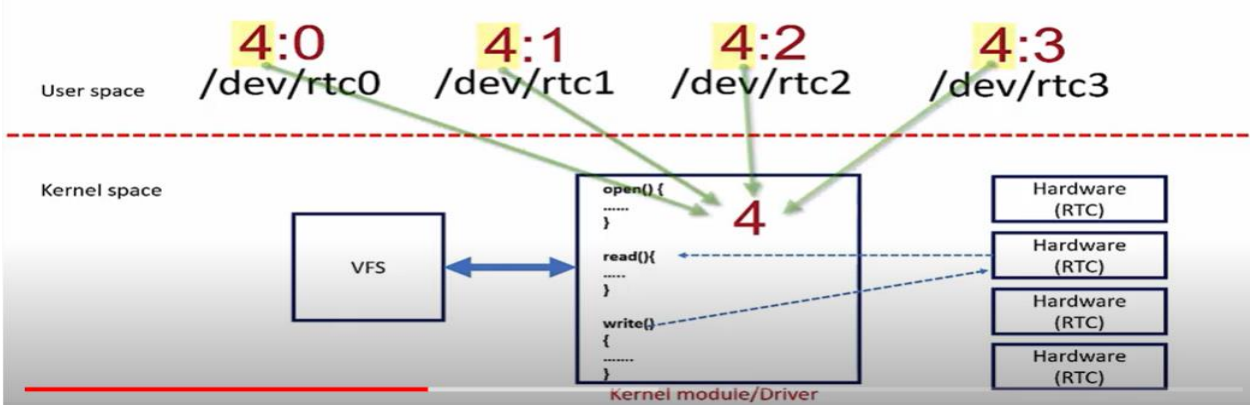
When we want to communicate(open/read/write) to device. Application calls system call (open/read/write)

User space system call is connected to driver system call implementation is taken care by VFS. Our device driver has to get registered with vfs.

When you use open system call on device file, how does the kernel connect to open system call to intended driver's open call?

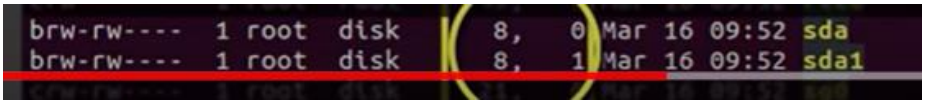
Kernel uses device no. Assign no. to driver

Below there are 4 instances of rtc type devices but driver will be same for all devices. There are 4 files created by driver. Communication with device will be done using device files.



Device number is combination of major number and minor number.

Major no. and minor no. can be checked in system by cd to dev and check ls-l command:

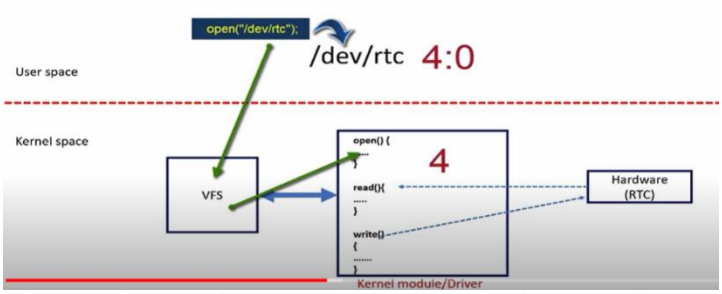


Connection establishment b/w device file access and driver:

Driver creates the device no., device file, makes char device registration with VFS using CDEV_ADD and

Implements the driver's file operation methods for open. Read, write etc.

When user program uses open system call on device file(dev/rtc), system call is handled by VFS first. VFS gets the device no. and compares it with driver registration list, that means this driver to get registered with VFS using device no. that we call character device add, CDEV_ADD (). When VFS gets open call from application, it opens a file by creating new file object and linking it to corresponding inode object.



Kernel APIs and utilities to be used in driver code

alloc_chrdev_region();	1.Create device number
cdev_init(); cdev_add();	2. Make a char device registration with the VFS
class_create(); device_create();	3. Create device files

Kernel APIs and utilities to be used in driver code

alloc_chrdev_region();	1.Create device number
cdev_init(); cdev_add();	2. Make a char device registration with the VFS
class_create(); device_create();	3. Create device files

Below creation calls are written in Init function of driver and deletion calls in Exit function of driver.

Creation	Deletion
alloc_chrdev_region();	unregister_chrdev_region();
cdev_init(); cdev_add();	cdev_del();
class_create(); device_create();	class_destroy(); device_destroy();

Kernel Header file details

Kernel functions and data structures	Kernel header file
alloc_chrdev_region() unregister_chrdev_region()	include/linux/fs.h
cdev_init() cdev_add() cdev_del()	include/linux/cdev.h
device_create() class_create() device_destroy() class_destroy()	include/linux/device.h
copy_to_user() copy_from_user()	include/linux/uaccess.h
VFS structure definitions	inclue/linux/fs.h

Device number representation

- The device number is a combination of major and minor numbers
- In Linux kernel, dev_t (typedef of u32) type is used to represent the device number.
- Out of 32 bits, 12 bits to store major number and remaining 20 bits to store minor number
- You can use the below macros to extract major and minor parts of dev_t type variable

```
dev_t device_number;
int minor_no = MINOR(device_number);
int major_no = MAJOR(device_number);
```

- You can find these macros in linux/kdev_t.h
- If you have, major and minor numbers , use the below macro to turn them into dev_t type device number

```
MKDEV(int major, int minor);
```

Device Files: The device file allows transparent communication b/w user-space application and hardware.

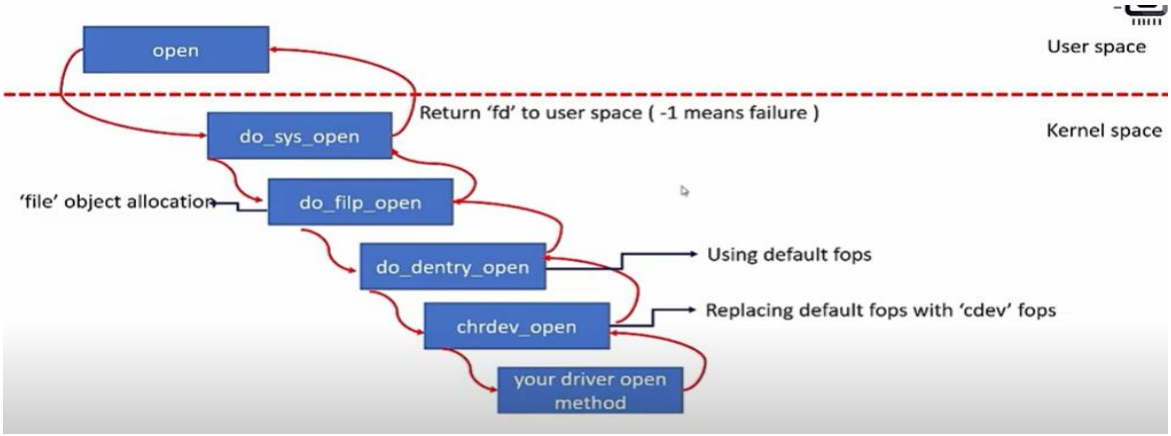
Device file creation:

1) **Manually:** We can create device file manually by mknod:
   mknod -m <permissions> <name> <device type> <major> <minor>
   sudo mknod -m 666 /dev/etx_device c 246 0

2) **Automatically:** The automatic creation of device files can be handled with udev. Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically.

   When device file is created -> vfs calls special_init_inode() function and this function checks type of device. inode object created in memory ->
   special_init_inode() does: in memory inode object's **device no** is initialized with device number of newly created device file. and inode object's file ops is initialized with dummy default file operation methods.
   When we call open method ()->

file object is created and now it calls do_dentry_open(). do_dentry_open() uses default ops. It calls default chrdev_open() which replaces default fops with driver's open system call.

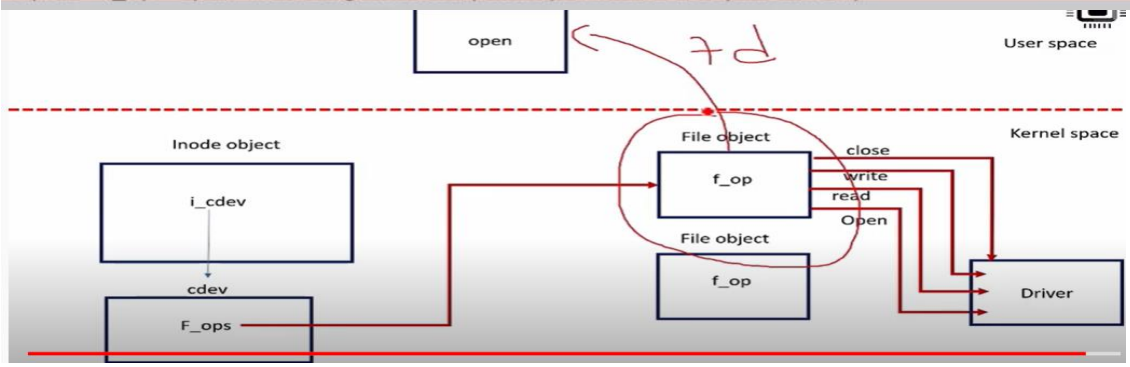


When device file gets created

1) create device file using udev
2) inode object gets created in memory and inode's i_rdev field is initialized with device number
3) inode object's i_fop field is set to dummy default file operations (def_chr_fops)

When user process executes open system call

1) user invokes open system call on the device file
2) file object gets created
3) inode's i_fop gets copied to file object's f_op (dummy default file operations of char device file)
4) open function of dummy default file operations gets called (chrdev_open)
5) inode object's i_cdev field is initialized with cdev which you added during cdev_add (lookup happens using inode-> i_rdev field)
6) inode->cdev->fops (this is a real file operations of your driver) gets copied to file->f_op
7) file->f_op->open method gets called (read open method of your driver)



---

mknod /dev/rama c 12 5

To deploy a module inside kernel, what are the possible methods.? Mention actual difference among them.

insmod requires you to pass it the full pathname and to insert the modules in the right order, while **modprobe** just takes the name, without any extension, and figures out all it needs to

know by parsing /lib/modules/version/modules.dep.
Explain about about ksets, kobjects and ktypes. How are they related?
Kobjects have a name and a reference count.
- A ktype is the type of object that embeds a kobject. Every structure that embeds a kobject needs a corresponding ktype. The ktype controls what happens to the kobject when it is created and destroyed.
- A kset is a group of kobjects. These kobjects can be of the same ktype or belong to different ktypes. The kset is the basic container type for collections of kobjects. Ksets contain their own kobjects, but you can safely ignore that implementation detail as the kset core code handles this kobject automatically.

1. As kernel can access user space memory, why should copy_from_user is needed?
Disables SMAP (Supervisor Mode Access Prevention) while copying from user space

2. how many ways we can assign a major minor number to any device?
There are two ways of a driver assigning major and minor number.
1. **Static Assignment:**
register_chrdev_region is the function to allocate device number statically.
2. **Dynamic Assignment**: alloc_chrdev_region is the kernel function to allocate device numbers dynamically

3. How is container_of( ) macro implemented?

4. Main Advantages and disadvantages of having separate user space and kernel space?
system calls might be **faster** (i.e. **lower latencies**), as the CPU doesn't have to switch from application mode into kernel. you might get **direct access to the system's hardware** via memory and I/O ports.

5. What is re entrant function: It can be reentered by another thread.

6. How will you insert a module statically in to linux kernel:
you just need to do a bit of hacking to move the external module into the kernel source tree, tweak the Makefiles/Kconfig a bit so that the code is built-in, and then build your kernel image.

7. how the device files are created in Linux:
They're called **device** nodes, and are **created** either manually with mknod or automatically by udev

8. How can a static driver runs? Without doing any insmod?

9. **What is the path of your driver inside kernel? /lib/modules/$(uname -r)**

10. Diff b/w SLAB and Vmalloc
Kmalloc is similar to malloc function, we use in our C program to allocate memory in user space. kmalloc allocates memory in kernel space. kmalloc allocates contiguous memory in physical memory as well as virtual memory. vmalloc is the other call to allocate memory in kernel space as like kmalloc.
vmalloc allocates contiguous memory in virtual memory but it doesn’t guarantee that memory allocated in physical memory will be contiguous.

11. How do you pass a value to a module as a parameter? ->**module_param()**

12. What is the functionality of PROBE function
The purpose of the probe routine is to detect devices residing on the bus and to create device nodes corresponding to these device

13. How do you get the list of currently available drivers ?

14. What is the use of file->private_data in a device driver structure ?
Private data to driver.

15. What is a device number ?

16. What are the two types of devices drivers from VFS point of view ?

17. How to find a child process in linux/unix.?
using the -P option of pgrep(pgrep -P pid)

18. What is the difference between fork( ) and vfork( )?
The primary **difference between** the **fork**() and **vfork**() system call is that the child process created using **fork** has separate address space as that of the parent process. On the other hand, child process created using **vfork** must share the address space of its parent process.

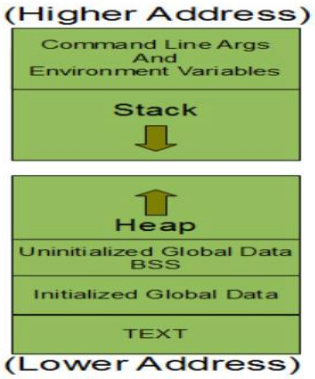
19. **What are the processes with PID 0 is Sched and PID 1 is init(**process primarily responsible for starting and shutting down the system**)**

20. What is the difference between interruptible and uninterruptible task states?

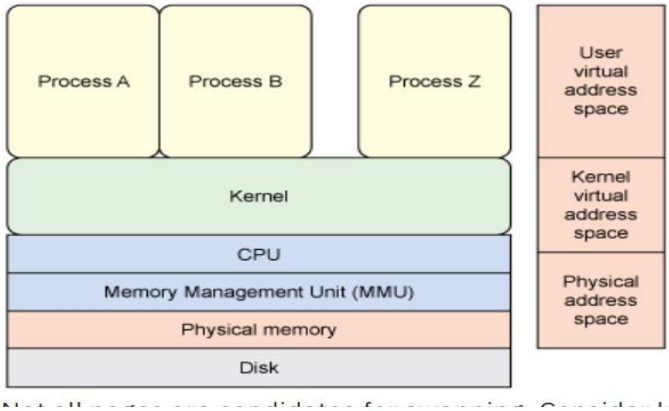
21. How processes and threads are created? (From user level till kernel level)

22. How to determine if some high priority task is hogging CPU: top

23. Priority inversion, priority inheritance, priority ceiling24. Process Memory Layout

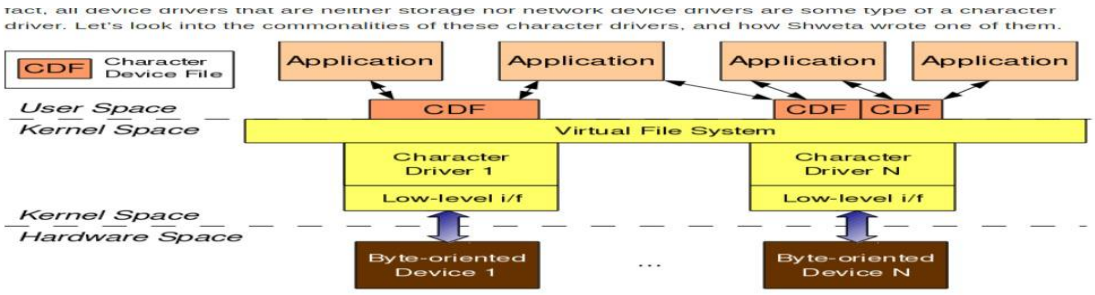


25. how much memory is occupied by process address space.
4 GB
26. When a same executable is executed in two terminals like terminal 1 execute ./a.out and terminal 2 executed ./a.out what will the program address space look like on RAM
27. **what is diff b/w process and threads?**
A **process is** a program under execution i.e an active program. A **thread is** a lightweight **process** that can be managed independently by a scheduler. **Processes** require more time for context switching as they are more heavy. **Threads** require less time for context switching as they are lighter than **processes**
28. Will threads have their own stack space?
Yes
29. **Can one thread access the address space of another thread?**
In general, each *thread* has its own registers (including its own program counter), its own stack pointer, and its own stack. Everything else is shared between the threads sharing a process.
A *process* is generally considered to consist of a set of threads sharing an address space, heap, static data, and code segments, and file descriptors*.
30. What is task_struct and how are task states maintained?
Task_struct is structure used to instantiate for each process.
Task_states are: Running, uninterruptible sleep(D), interruptible sleep(S), Zombies



Virtual address space to Physical address space.
Mutex: When want to provide atomic access to critical section. A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.
Semaphore: we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.
Spinlock: Use a spinlock when you really want to use a mutex, but your thread is not allowed to sleep. e.g.: An interrupt handler within OS kernel must never sleep.
Deadlock: If a thread which had already locked a mutex, tries to lock the mutex again, it

will enter the waiting list of that mutex, which results in deadlock.
Scheduling methods such as First Come First Serve, Round Robin, Priority-based scheduling
Steps to invoke device driver:
1) User space process tries to write to character device.
2)**Device file:** All data will be communicated through device file will be in *dev*.
3) **Device driver:** This is the software interface for the device and resides in the kernel space.
4) Device: This can be the actual device present at the hardware level, or a pseudo device.



Inode:
The inode (index node) keeps information about a file in the general sense (abstraction): regular file, directory, special file (pipe, fifo), block device, character device, link, or anything that can be abstracted as a file.

Interview questions:

As kernel can access user space memory, why should copy_from_user is needed?
If kernel directly accesses the user data structure, system will panic if it's not a valid address (eg NULL pointer). To avoid this situation, copy_from_user(..) is used. This function will fail, if proper user address is not provided and does not bring down the system.

To deploy a module inside kernel, what are the possible methods.? Mention actual difference among them?
1. dynamic insertion as a kernel module (modeprobe and insmod)
2. linking statically to the kernel code But everyone will prefer modprobe because insmod have no capability to resolve dependency issue. But modprobe can do that.

What is cache ? How it is used and mapped the physical address cache and virtual address cache ?
Cache is a component that improves performance by transparently storing data such that future requests for that data can be served faster.

The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparably faster.

How one can measure time spent in context switch?
record the timestamps of the first and last instructions of the swapping processes. The context switch time is the difference between the two processes.

How to debug kernel?
printks ! Simple and very convenient. KDB and kernel probes can also be used.

What is the difference between kill-6 and kill -9
Kill-9: SIGKILL
Kill-6: SIGABRT

How would you handle sleeping or blocking instructions in an Interrupt Service Routine (if unavoidable) or basically if the length of ISR is long?
Sleep or blocking cannot be allowed in ISR(Interrupt context), Task lets(soft interrupt context)
But allowed in Work Queues(kernel context).

Explain device tree concepts in linux. : is a data structure describing the hardware components of a particular computer so that the operating system's kernel can use and manage those components, including the CPU or CPUs, the memory, the buses and the integrated peripherals.

Given a pid, how will you distinguish if it is a process or a thread ?

ps -AL | grep pid
1st column is parent id and the second column is thread (LWP) id. if both are same then its a process id otherwise thread.

How to install software in Linux:

Installing RPM packages:
command: ***rpm -i RPMPackage.rpm***
Installing DEB packages: ***apt-get install DEBPackage.deb***
apt-get remove DEBPackage.deb
Installing from tarballs(esp. Source code):
./configure
make
make install