//
**********Linux_notes**********\\

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 component 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.

//***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 happen via message passing. Others are maintained as server processes in User Space.Slower Processing due to additional Message Passing.

//****Fork**:The fork system call is used to create a new processes. 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. chiled process inherits mutex, condition variables, open file descriptor, message queue descriptor.
//page tables are copied and page frames are shared.

//*****Exec**: is used to execute commands given on bash.It replaces the current process image with new process image.Current virtual address space and page table is freed and new virtual address sapce and page tables are created.

//***sched_yield**() causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.

//**clone:** is similar to fork but it is used to create thread(pthead_create). new process is not independent of parent process.

//**Signals**: one process can raise the signal and can deliver to others. When signal is sent to process: Every process has process descriptor and process descriptor has three fields:signal pending field, signal masking/unmasking field, 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.

//
**********IPC**********\\

unnamed pipe:It is used for related process(within process)unidirectional byte stream which connects o/p of one proces into i/p of other process.Pipes are implemented using file descriptor. ex: int pfd[2], pipe(pfd);, write(), read().

Named/fifo pipes are used for two way communication. pipe can be created using mknod(const char *pathname, mode_t mode, dev_t dev);

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 process to exchange information. The information has to go 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())

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())

//**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.

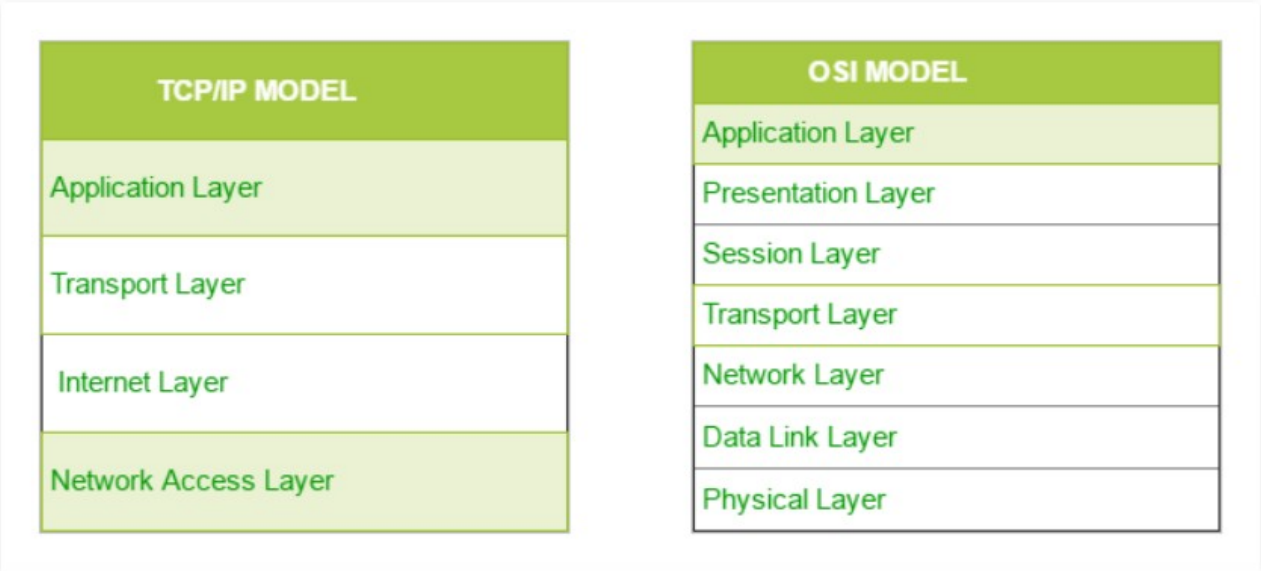
//**Socket programming:** Socket is used in a client-server application framework.
//Socket types: Stream Sockets(TCP/IP): It's reliable protocol and also data integrity is maintained.It's connection oriented.(Transmission control protocol)
//dataGram socket: UDP:It's not reliable.It's connection less.

//**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.

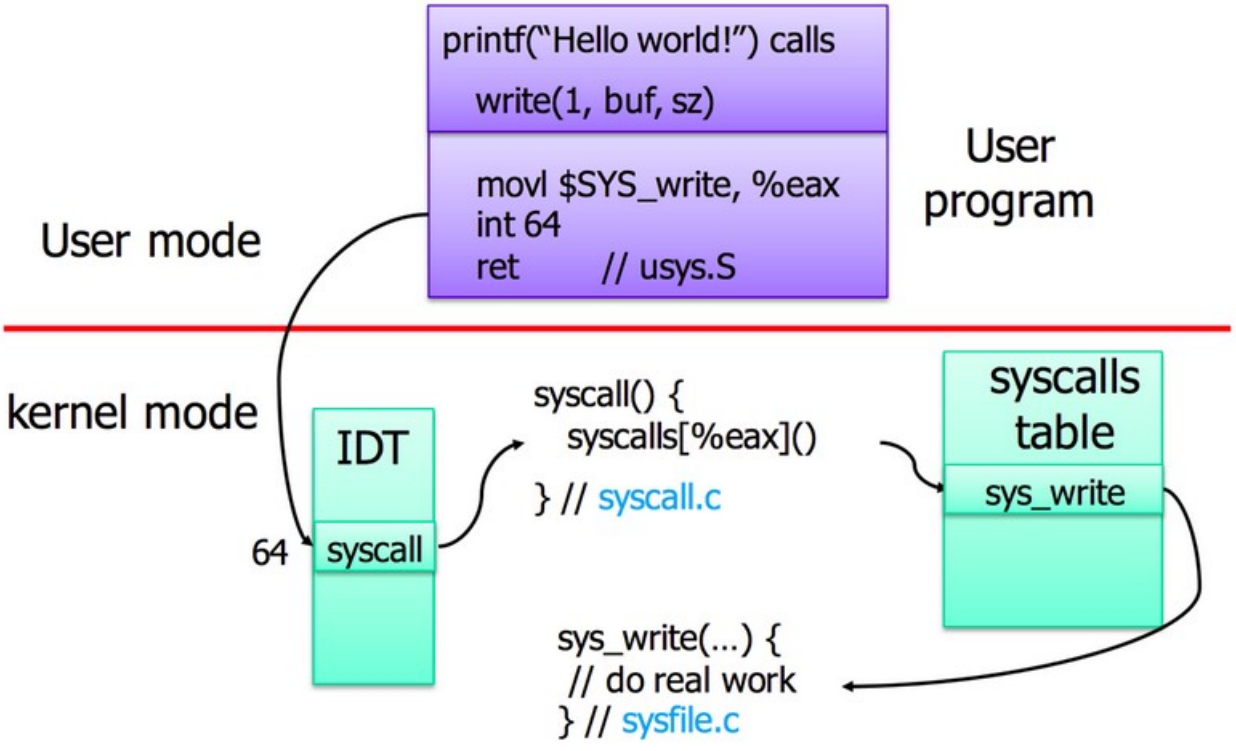
//**Make client**:
//Create a socket with the socket() system call.
//Connect the socket to the address of the server using the connect() system call.
//Send and receive data. There are a number of ways to do this, but the simplest way is to use the read() and write() system calls.

//**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:



//**system call**: open/read/write/fork. read(c program in user space)--> printf calls write and write assign interrupt no. To sys_write. Now interrupt no. Is mached in IDT to call sysem call table and then from there correct system call is called.



Scheduler:
The scheduler is invoked: when ther is change in process state, new process is created, software interrupt, hardware interrupt etc.

Thread: we use clone() to create thread, thread is different from process. Thread uses process address space and threads communicate through shared memory. In linux, we call ptheread_create for creating the thread.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg); -->can pass argument to routine
which will be called when thread is called.

int pthread_join(pthread_t thread, void **value_ptr);--> shall suspend execution
of the calling thread until the target thread terminates
```

Mutex lock for Linux Thread Synchronization

int pthread_mutex_init(pthread_mutex_t ****mutex*, const pthread_mutexattr_t ****mutexattr***);**

int pthread_mutex_lock(pthread_mutex_t ****mutex*));**

int pthread_mutex_trylock(pthread_mutex_t ****mutex*);**

int pthread_mutex_unlock(pthread_mutex_t ****mutex*);**

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.

Condition variable: pthread_cond_init()
pthread_cond_wait()
pthread_cond_signal()

**********scenario for cond_variable**********
Two threads reading and writing to buffer. Writer thread locs mutex object and perform write operation to buffer and reader thread locs mutex object once it is released by writer thread and read it.
We can implement it through conditon variable, like write thread sends signal pthread_cond_signal to reader thread after writing 5 lines.

****

Semaphore 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.

ldd: load dynamic dependency--> `ldd executable name will listout dynamic library dependencies for that executable.`

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

Hardware interrupt handling:

1) The keyboard sends a scan code of the key to the keyboard controller (Scan code for key pressed and key released is different)

2) The keyboard controller interprets the scan code and stores it in a buffer

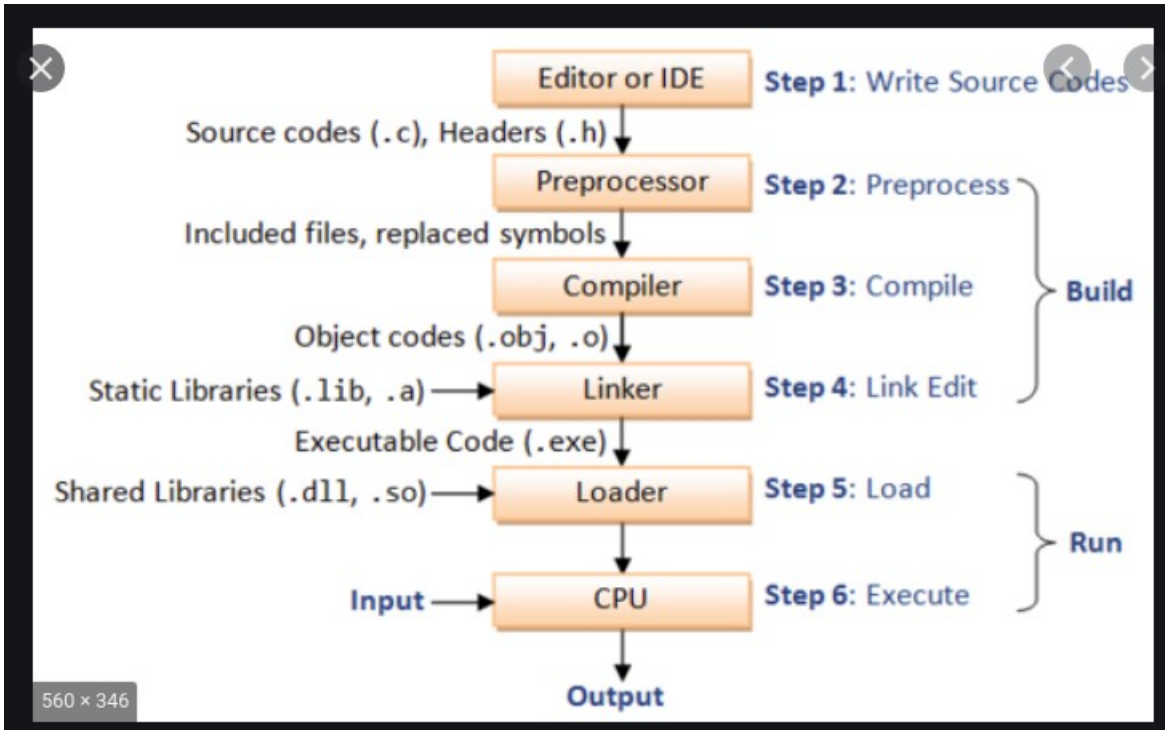
3) The keyboard controller sends a hardware interrupt to the processor.This is done by putting signal on “interrupt request line”: IRQ 1

4) The interrupt controller maps IRQ 1 into INT 9

5) An interrupt is a signal which tells the processor to stop what it was doing currently and do some special task

6) The processor invokes the “Interrupt handler” CPU fetches the address of “Interrupt Service Routine” (ISR) from “Interrupt Vector Table” maintained by the OS (Processor use the IRQ number for this)

7) The ISR reads the scan code from port 60h and decides whether to process it or pass the control to program for taking action.



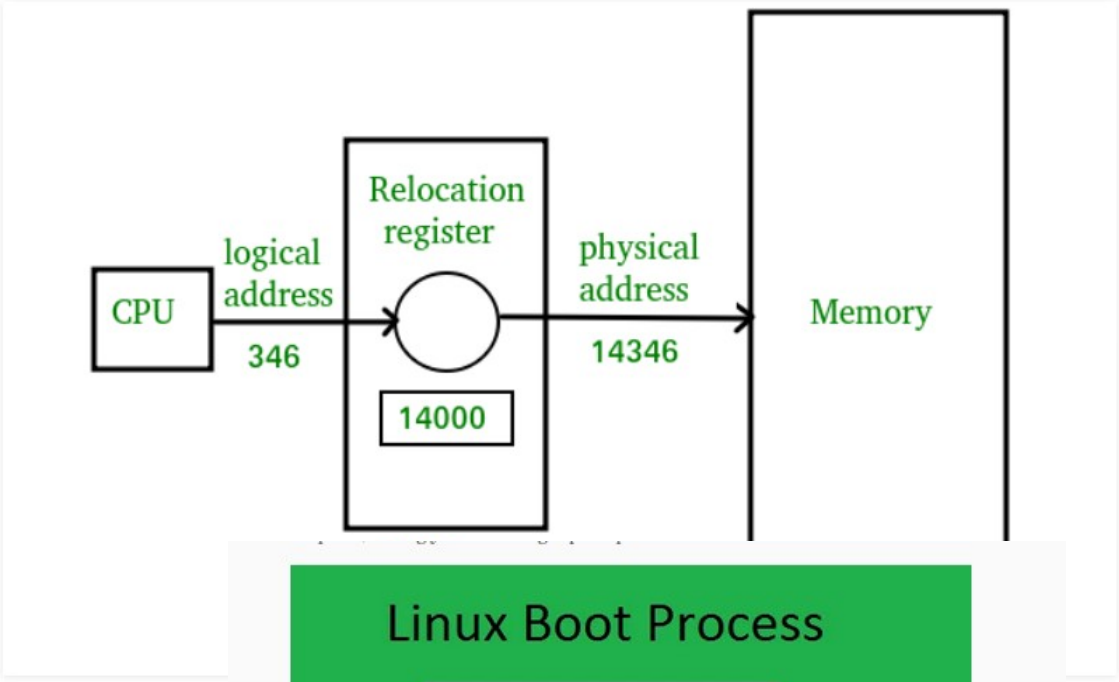
**********OS Concepts**********

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

MMU scheme –

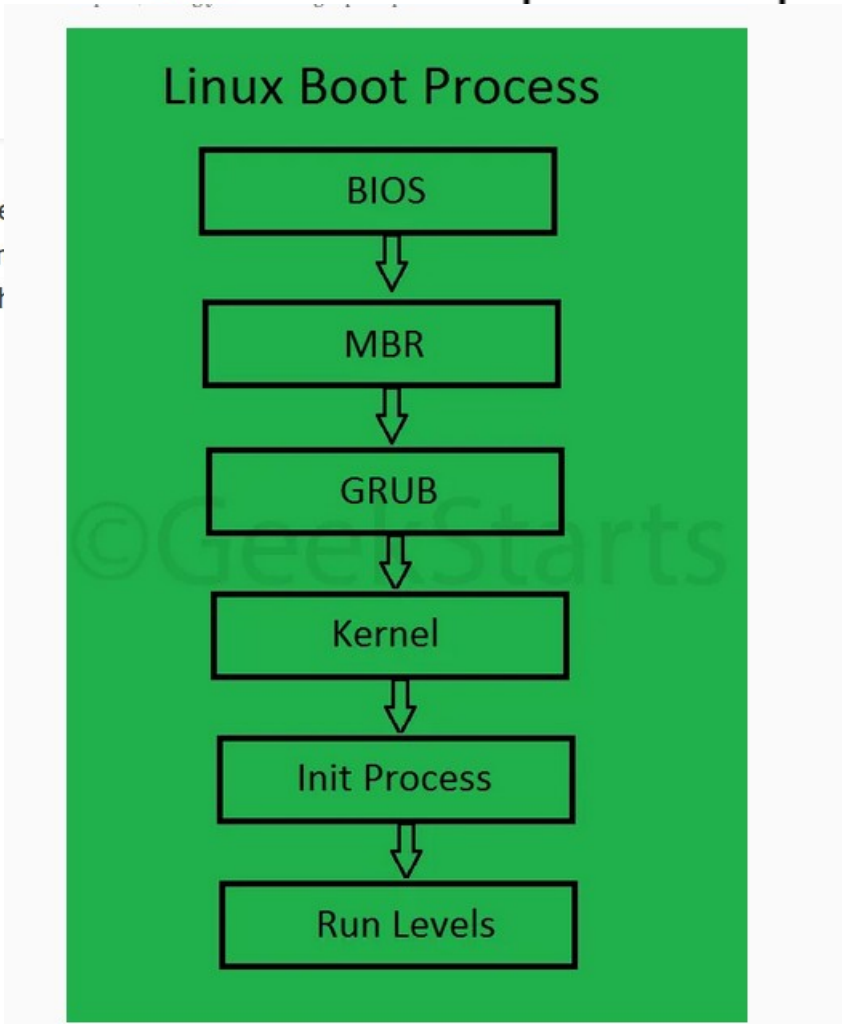
```
CPU------- MMU------Memory
```



1. CPU will gene
2. MMU will ger
3. In Memory pl

needed.

Linux Booting steps:



1. BIOS.

- BIOS performs a POST (Power On Self Test) to check and scan if all the hardware devices are properly connected.
- Then it scans the first sector of Hard-drive partition to find the boot loader (GRUB LILO for Linux).
- BIOS loads the MBR into memory.
- Gives the control to MBR

2. MBR.

- MBR or Master Boot Record holds information of your current Boot-loader of your Operating System.
- MBR is less than 512 bytes in size and holds various information like boot loader information, validation check and the partition table present in your hard-drive.
- It is always stored in the first sector of your hard-drive.
- MBR loads the first stage loader (stage1).
- The first stage boot loader (stage 1), loads the rest of boot loader prompting you an option to select multiple OS (if you have installed multiple OS) or loads the Operating System on hard-drive.
- MBR loads the GRUB (boot loader for mostly all Linux OS) and gives the control over to GRUB.

3. GRUB.

- GRUB, also known as Grand Unified Bootloader is the most common boot loader for various Linux distributions.
- GRUB gives you an option to select multiple OS, if you have them in your hard-drive partitions.

- The second stage loader (stage2) is loaded, giving you the GRUB screen where you can select multiple OS or change the default settings or edit start-up parameters.
- GRUB has the kernel and intrid images, which it loads and executes.

4. Kernel.

- Kernelis loaded in two steps:
    1. Kernel is loaded in Memory and decompressed and sets up the crucial functions.
    2. Kernel then runs the init process (in /sbin/init). It also sets up user space and essential processes needed for environment and for user login.
- To initialize the scheduler which has Process ID (PID 0) of 0, run the init process (PID 1) and then mount the system in rw mode are the responsibilities of kernel.
- The init process in the second step loads the critical daemons, checks the fstab file and loads the partitions accordingly.

5. Init Process.

- This process checks the '*/etc/inittab*' file to choose the run level.
- It reads the file to check default init level and executes it.
- Various init levels are:
    - 0 = Halt.
    - 1 = Single user mode.
    - 2 = Multiuser mode w/o NFS.
    - 3 = Full multiuser mode.
    - 4 = Reserved (for future use).
    - 5 = X11.
    - 6 = Reboot.

6. Run level programs.

- If you press any key when you see the GUI and system is loading up, you go into the text mode, where you can see the kernel starting and testing all the daemons. Eg: Starting DHCP server…. Ok.
- These are run-level programs, defined in your run level directory. You can change these run-levels in
  '*/etc/rc.d/rc*/d*', where * varies from 0-6.
- These run-levels also have symbolic links in '*/etc/*'. That is '*/etc/rc0.d*' has symbolic link with '*/etc/rc.d/rc0.d*'.

Microprocessor vs Microcontroller:
Microprocessor is an IC which has only the CPU inside them i.e. only the processing powers.
Microcontroller has a CPU, in addition with a fixed amount of RAM, ROM and other peripherals all embedded on a single chip.

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.

CPU switches from User mode to Kernel mode in following cases:

1) User process triggers system calls

2) Device sends interrupt

3) CPU raises exception.

Process Management: Each process is represented by a process descriptor that includes information about the current state of the process.
When program stops execution,it saves the current contents of several processor registers in the process descriptor:
The program counter (PC) and stack pointer (SP) registers
The general-purpose reg**system call**isters
The floating point registers
The processor control registers (Processor Status Word) containing information about the CPU state
The memory management registers used to keep track of the RAM accessed by the process

When the kernel decides to resume executing a process, it uses the proper process descriptor fields to load the CPU registers. Since the stored value of the program counter points to the instruction following the last instruction executed, the process resumes execution from where it was stopped.

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.

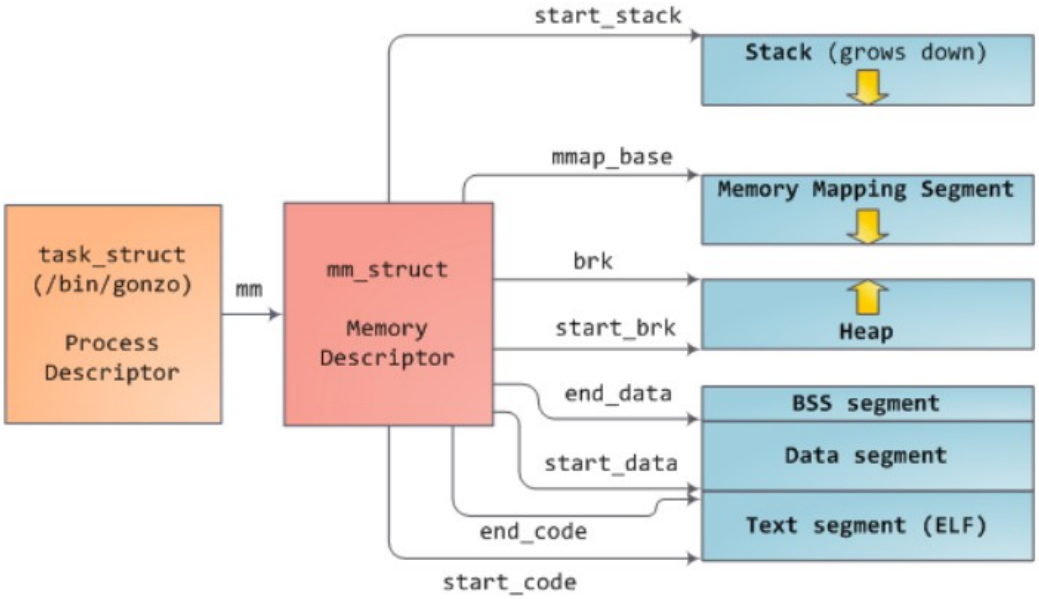
Virtual memory:
acts as a logical layer between the application memory requests and the hardware Memory. Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory
Management Unit (MMU).
Virtual memory advantages:
Several processes can be executed concurrently.
It is possible to run applications whose memory needs are larger than the available physical memory.
Processes can execute a program whose code is only partially loaded in memory.

The main ingredient of a virtual memory subsystem is the notion of virtual address space. The set of memory references that a process can use is different from physical memory addresses. When a process uses a virtual address, [9] the kernel and the MMU cooperate to locate the actual physical location of the requested memory item.

For example, a system with 4GB RAM would have a minimum of 1024x4x1.5=6,144MB [1GB RAM x Installed RAM x Minimum]. Whereas, the maximum is 1024x4x3=12,288MB [1GB RAM x Installed RAM x Maximum].

Random access memory usage:
All Unix operating systems clearly distinguish two portions of the random access memory (RAM). A few megabytes are dedicated to storing the kernel image (i.e., the kernel code and the kernel static data structures). The remaining portion of RAM is usually handled by the virtual memory system and is used in three possible ways:
To satisfy kernel requests for buffers, descriptors, and other dynamic kernel data structures
To satisfy process requests for generic memory areas and for memory mapping of files
To get better performance from disks and other buffered devices by means of cache

---



Each process is represented by process descriptor(instance of task_struct),Taks_struct contains mm_struct, mm_struct has primary table and primary table has secondary table and this is finally mapped to Page frames(physical memory).

Process's logical address space(256kb)-->1 page size(4kb)-> primary table will contain 256/4 = 64 entries.

Kernel Memory Allocator:
The Kernel Memory Allocator (KMA) is a subsystem that tries to satisfy the requests for memory areas from all parts of the system. Some of these requests will come from other kernel subsystems needing memory for kernel use, and some requests will come via system
calls from user programs to increase their processes' address spaces.

Process virtual address space handling:
The address space of a process contains all the virtual memory addresses that the process is allowed to reference. The kernel usually stores a process virtual address space as a list of memory area descriptors.

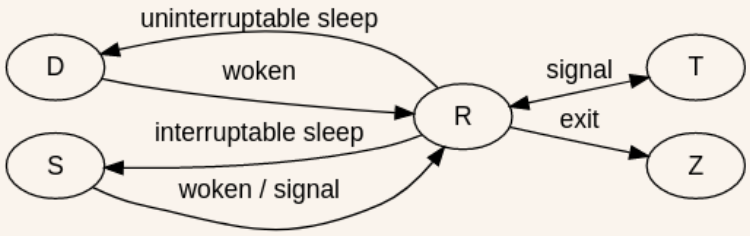
Swapping and caching
In order to extend the size of the virtual address space usable by the processes, the Unix operating system makes use of swap areas on disk. The virtual memory system regards the contents of a page frame as the basic unit for swapping.

Process: Every process under Linux is dynamically allocated a `struct task_struct` structure. The maximum number of processes which can be created on Linux is limited

only by the amount of physical memory present, and is equal to (see `kernel/fork.c:fork_init()`):

```
PROCESS STATE CODES
   R  running or runnable (on run queue)
   D  uninterruptible sleep (usually IO)
   S  interruptible sleep (waiting for an event to complete)
   Z  defunct/zombie, terminated but not reaped by its parent
   T  stopped, either by a job control signal or because
      it is being traced
   [...]
```

A process starts its life in an `R` "running" state and finishes after its parent reaps it from the `Z` "zombie" state.



Interrupts and Exceptions
Interrupts are often divided into synchronous and asynchronous interrupts:
Synchronous interrupts: are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction.
Asynchronous interrupts: are generated by other hardware devices at arbitrary times with respect to the CPU clock signals.

Maskable and non-maskable interrupt:
Non-maskable interrupt can not be ignored by CPU(chipset errors, memory corruption problems, parity errors and high-level errors needing immediate attention)
Maskable interrupt can be ignored by CPU

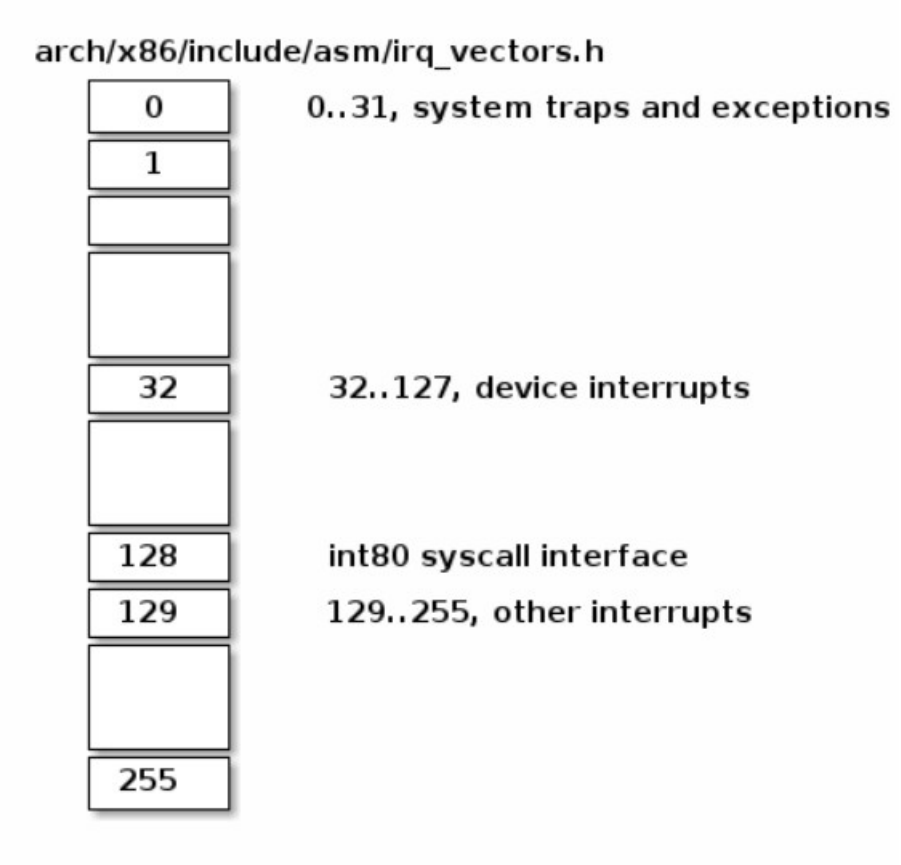
Hardware Interrupt handling
device raises an interrupt on the corresponding IRQn pin

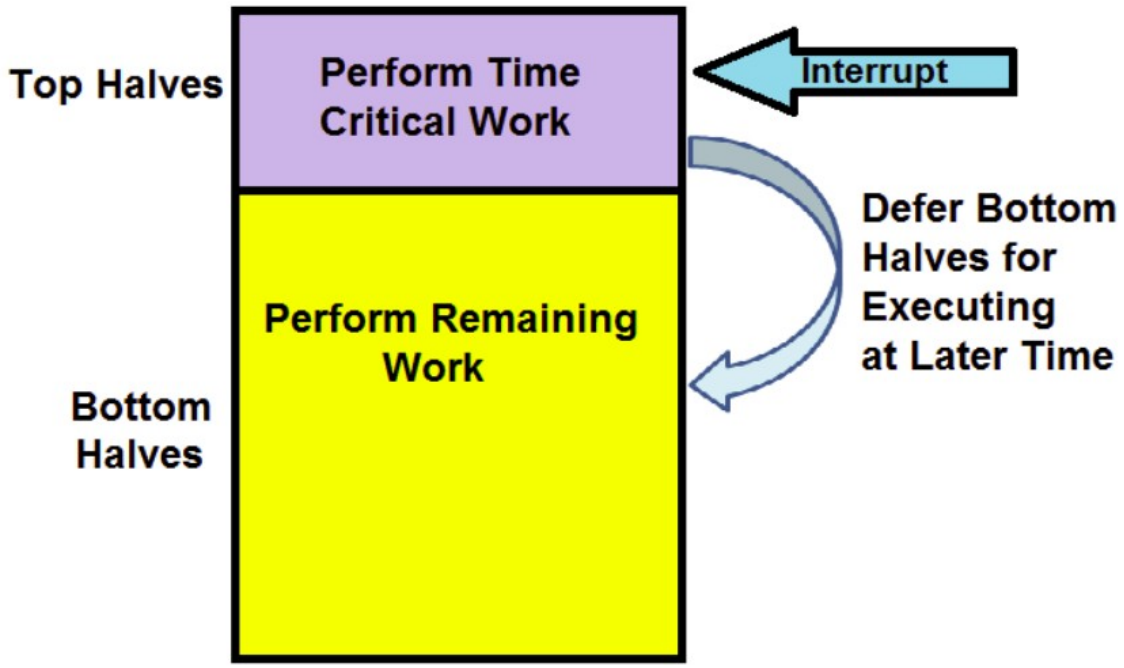
PIC(Programmable Interrupt Controller) converts the IRQ into a vector number and writes it to a port for CPU to read

PIC raises an interrupt on CPU INTR pin

PIC waits for CPU to acknowledge an interrupt

CPU handles the interrupt





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 some time 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) Acknowledgement 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.

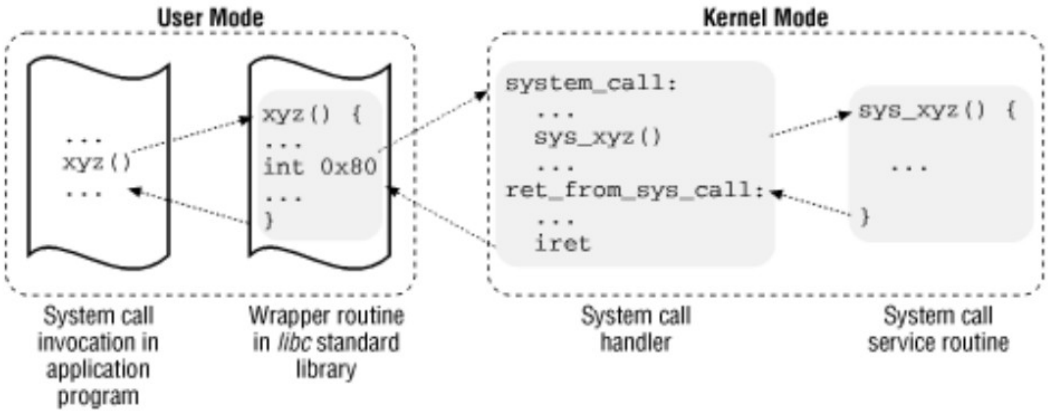
5)If the work needs to be ensure that another interrupt does not interrupt it , should be perform in interrupt handler.

Softirqs vs Tasklet:

Sofirqs are re-entrant , that is the different CPU can take the same softirq and execute it while the Tasklets are serialized that is the same CPU which is running the tasklet has the right to complete it

---

Figure 8-1. Invoking a system call

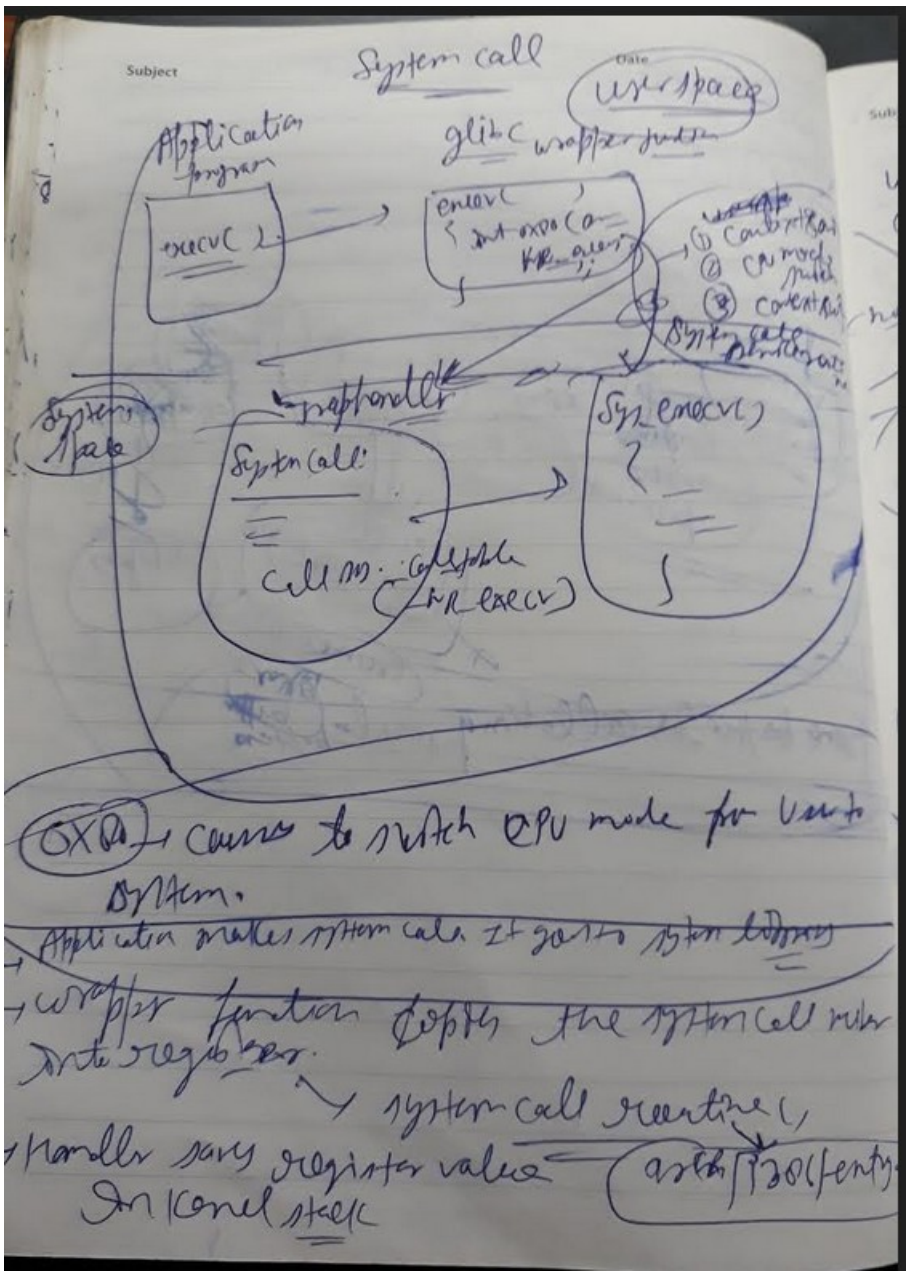


When user space process calls system calls, corresponding wrapper function will be available in libc library. This wrapper function will set vector no. 0X80. CPU will save the current process context in sytem stack and also will change processor state. In Kernel space, this will be checked in Interrupt vector table and system call table will be called and coresponding system call routine will be called.

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.

The main advantage of a preemptive kernel is that sys-calls do not block the entire system. If a sys-call takes a long time to finish then it doesn't mean the kernel can't do anything else in this time.



A 32-bit system can access 2^{32} memory addresses, i.e 4 GB of RAM or physical memory. A 64-bit system can access 2^{64} memory addresses, i.e actually 18-Quintillion bytes of RAM.

Device Drivers

The kernel interacts with I/O devices by means of device drivers. Device drivers are included in the kernel and consist of data structures and functions that control one or more devices

--Installing kernel:

1) Download and extract kernel code.
2) copy current kernel's configuration.
3) Make menuconfig

4) make --> compiles and links the kernel image. This is a single file named `vmlinuz`.
5) make modules_install --> **installs your kernel modules to `/lib/modules` or `/lib/modules/<version>`**
6) make install--> installs your built kernel to `/vmlinuz`

Writing my kernel module and loading dynamically:
Modprobe Vs insmod:
`modprobe` is the intelligent version of `insmod`. `insmod` simply adds a module where `modprobe` looks for any dependency (if that particular module is dependent on any other module) and loads them.

Module will be loaded into *proc*/modules/

Rmmod: remove modules from module list

Type of Devices:
character devices(serial ports, parallel ports, sounds cards)
Block devices(hard disk, floppy disk)
Network devices(switches, bridge,router)

Majors and minors

Char devices are accessed through names in the filesystem.

Major no is of 12bits and Minor no. Is of 20bits. Major no. Is for driver and minor no. is for device. Device file can be created using mknod: **mknod /dev/mycdev c 42 0**

Steps performed for device driver:
load driver->open device->read device/write to device->close device->unload driver

Reading from device-> uses copy_to_user
Write to device-> copy_from_user

Driver code sample:

```
static int dev_open(struct inode *inod, struct file *fil);

static ssize_t dev_read(struct file *filep,char *buf,size_t len,loff_t *off);

static ssize_t dev_write(struct file *flip,const char *buff,size_t len,loff_t *off);

static int dev_release(struct inode *inod,struct file *fil);

//structure containing device operation

static struct file_operations fops=

{

.read=dev_read, //pointer to device read funtion

.write=dev_write, //pointer to device write function
```

```
.open=dev_open,   //pointer to device open function
.release=dev_release, //pointer to device realese function
};
static int hello_init(void)   //init function to be called at the time of insmod
{
int t=register_chrdev(90,”mydev”,&fops);
if(t<0)
printk(KERN_ALERT “device registration failed.”);
else
printk(KERN_ALERT “device registred\n”);
return 0;
}
static void hello_exit(void) //exit function to be called at the time of rmmod
{
unregister_chrdev(90,”mydev”);
printk(KERN_ALERT “exit”);
}
 static int dev_open(struct inode *inod, struct file *fil)
{
printk(“KERN_ALERT device opened”);
return 0;
}
 static ssize_t dev_read(struct file *filep,char *buf,size_t len,loff_t *off)
{
copy_to_user(buf,ker_buf,len);
return len;
}
 static ssize_t dev_write(struct file *flip,const char *buf,size_t len,loff_t *off)
{
copy_from_user(ker_buf,buf,len);
ker_buf[len]=0;
return len;
```

```
}

static int dev_release(struct inode *inod,struct file *fil)

{

printk("KERN_ALERT device closed\n");

return 0;

}

 module_init(hello_init);

module_exit(hello_exit);
```

****

ioctl: The system call ioctl() is provided for device-specific custom commands (such as format, reset and shutdown) that are notprovided by standard system calls such as read(), write and mmap(). To invoke ioctl commands of a device, the user-space program would open the device first, then send the appropriate ioctl() andany necessary arguments

---

**********GDB**********

compile with -g to load debug symbols.
Gdb executable name
b/break fun_name/line no.
Run(r)
s(step into)
n(next line)
c(continue)

****

Priority inversion is a operating system scenario in which a higher **priority** process is preempted by a lower **priority** process.

Scenario: Three process p1(High), p2(medium), p3(low). P1 & p3 have to execute in critical section. P3 has locked mutex and excuting in cs, in betwen, p2 preempts p3 and p1 preempts p2.. so p1 can not excute cs since it has been locked by p3. This becomes priority inversion.

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.

---

mknod is a command which used create the device file (or) node in Linux file system. In unix or linux we will represent everything as a file .

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-> **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 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)`

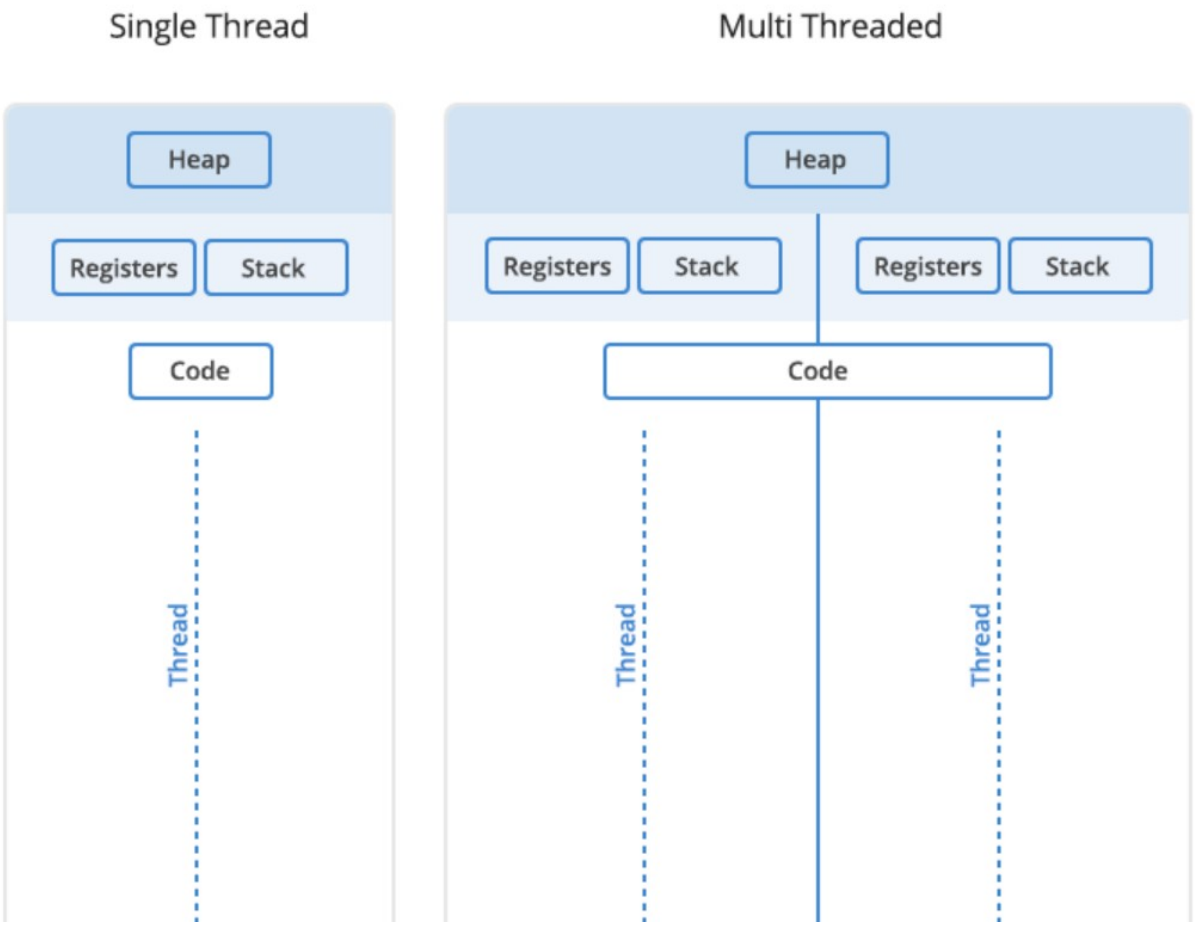
18. What is the difference between fork( ) and vfork( )?

    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** has to 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?

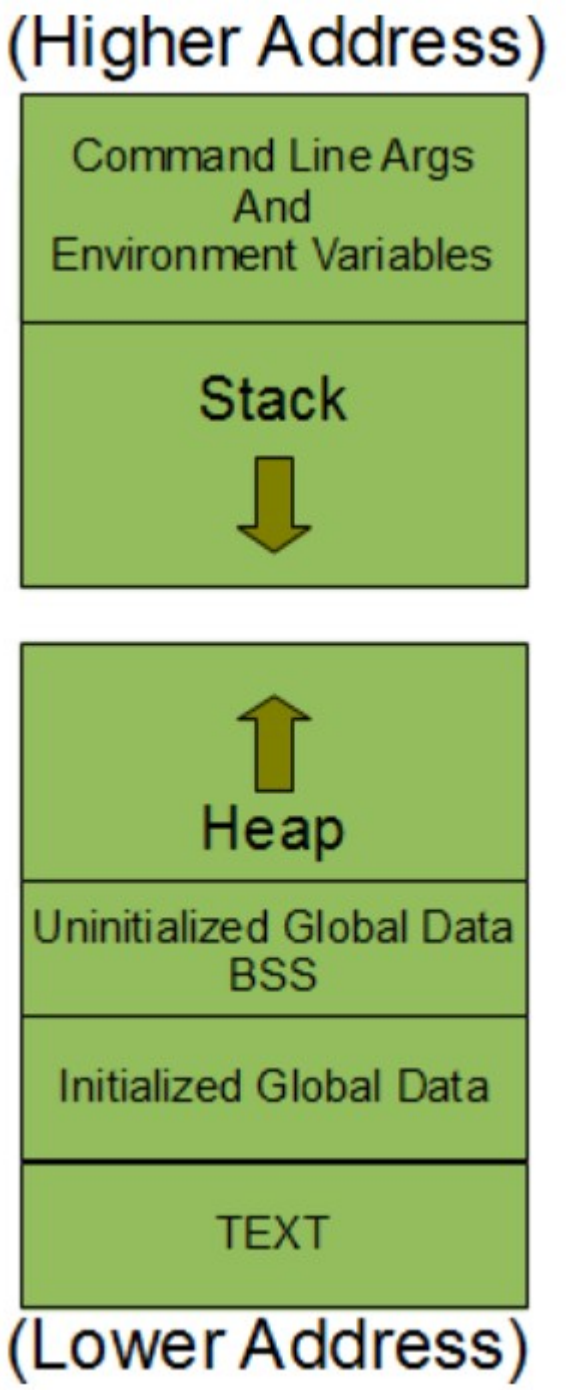
21. How processes and threads are created? (from user level till kernel level)



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

23. Priority inversion, priority inheritance, priority ceiling

24. Process Memory Layout



25. how much memory is occupied by process address spcae.

    4 GB

26. When a same executable is excuted in two terminals like terminal 1 execute ./a.out and terminal 2 executed ./a.out what will the programe 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?

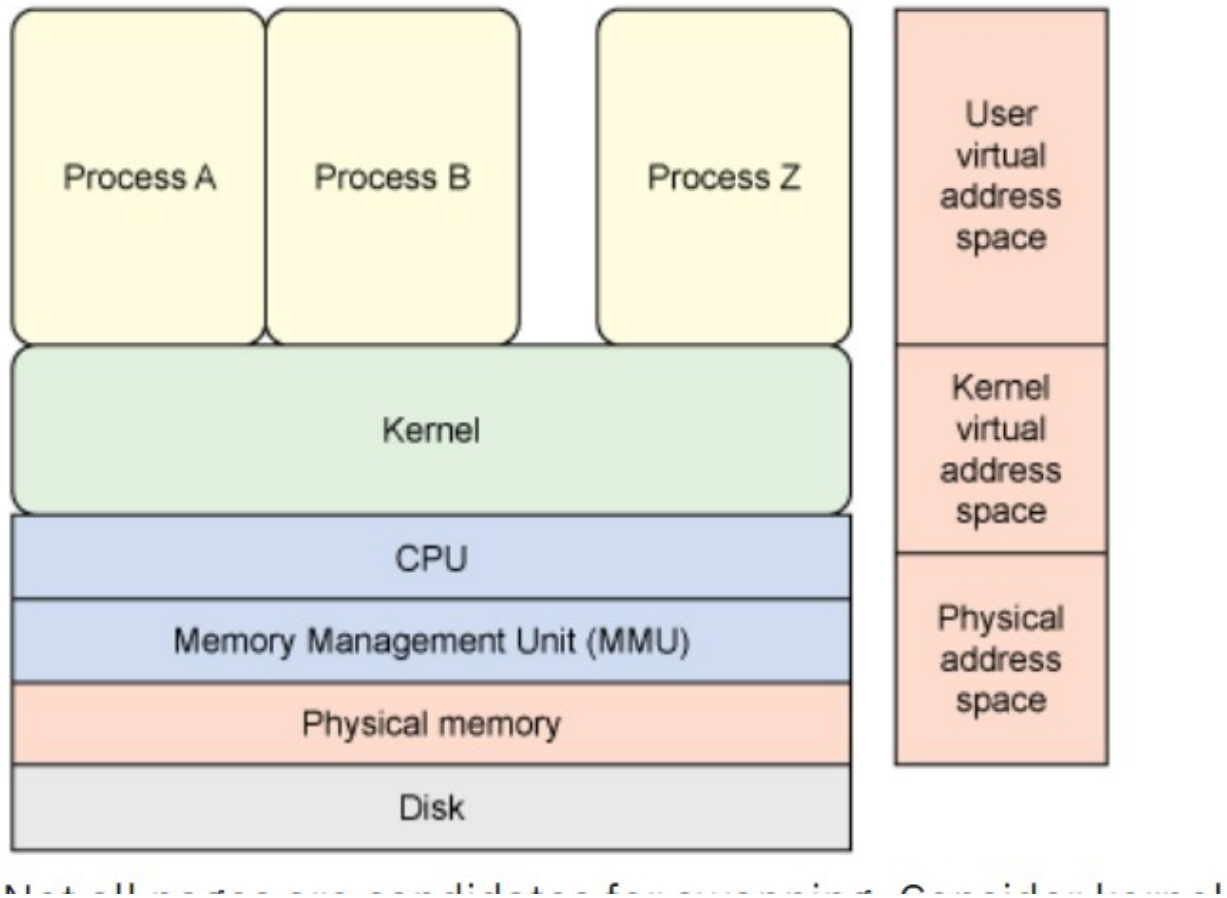
In general, each *thread* has its own registers (including its own program counter), its own stack pointer, and its own stack. Everything else is shared between the threads sharing a process.

In particular 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 and every 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.

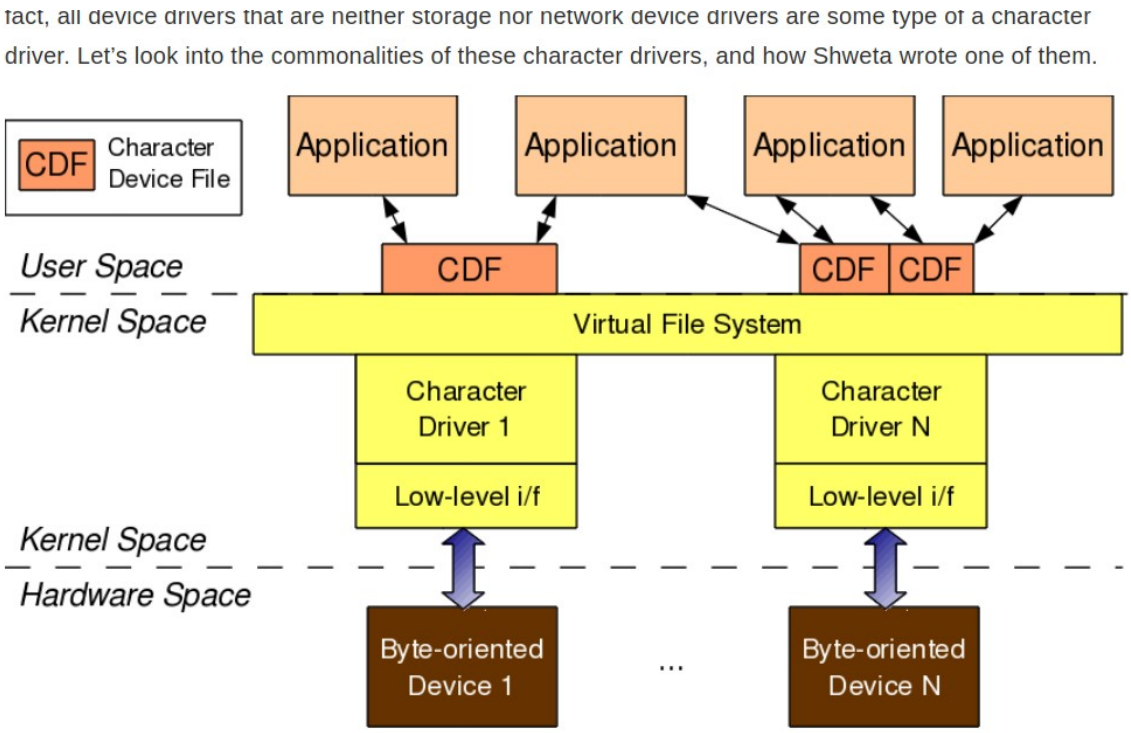
Spinlock: Use a spinlock when you really want to use a mutex, but your thread is not allowed to sleep. e.g.: An interrupt handler within OS kernel must never sleep.

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.