-------------------------Linux kernel and Booting process------------------------------

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

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.

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.

An Introduction to Linux: When installing Linux, the source code is usually stored in /usr/src/linux.

Kernel:

A Unix kernel fulfills 4 main management tasks:

• Memory management
• Process management
• file system management
• I/O management

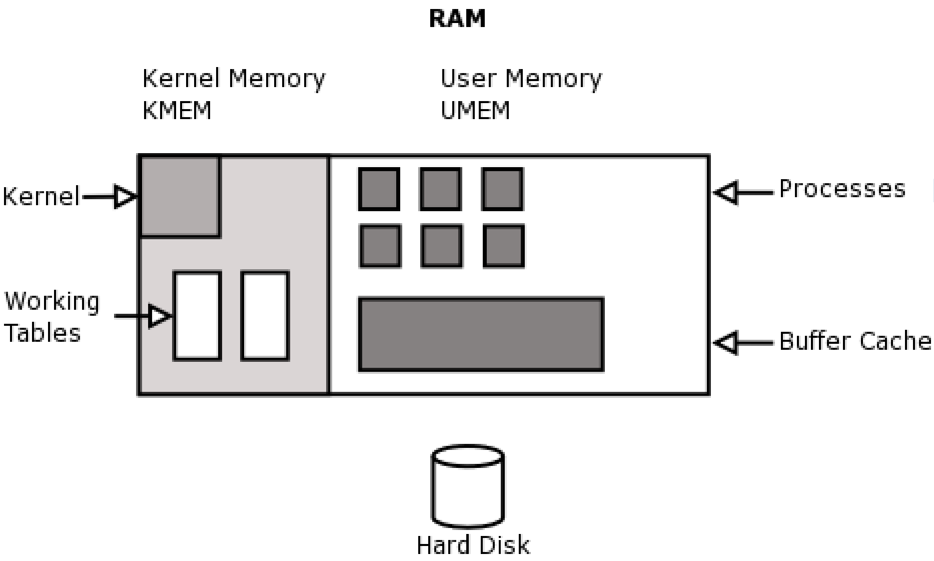
The kernel exists as a physical file on the file system in Linux it is /boot directory
and is usually called vmlinux.

/boot/vmlinuz-2.4.18-22

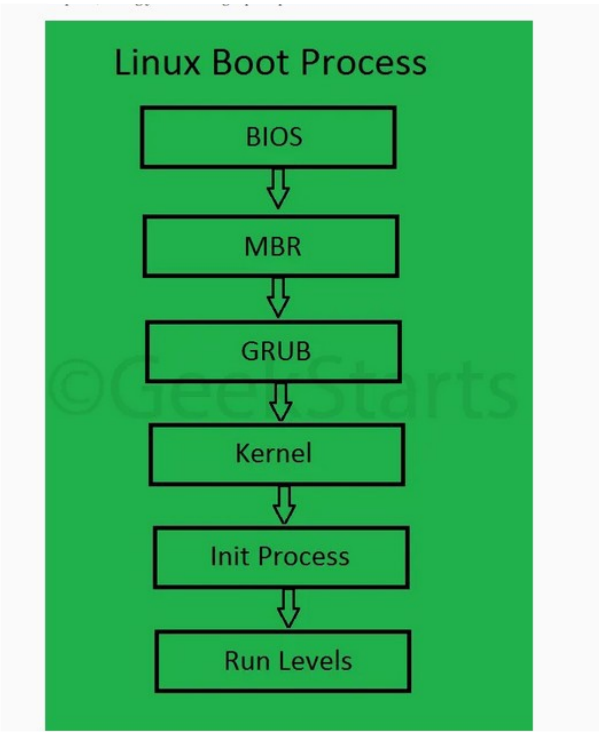
- At system boot time RAM only contains the boot loader, consuming a few kilobytes at the beginning of memory.
- The boot loader loads the kernel binary into memory from the hard disk, and places
- it at the beginning of memory.
- Once the kernel has been read in the boot loader tells the CPU to execute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

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.



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.

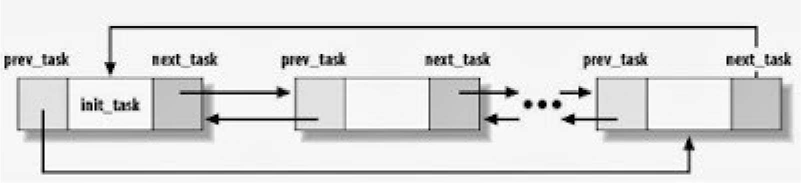
----------------The Kernel versus Process Management---------------------------------------

Process:

An instance of a running program is called a process.

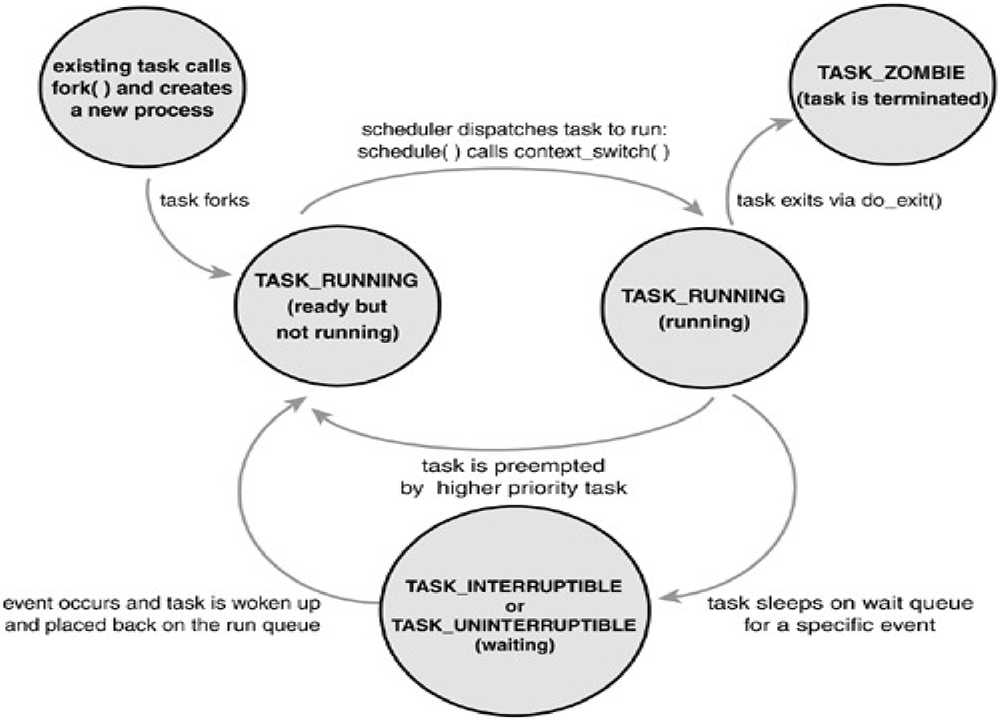
Process Descriptor and the Task Structure:

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.**

Process State:


existing task calls fork( ) and creates a new process
task forks
scheduler dispatches task to run: schedule( ) calls context_switch( )
TASK_ZOMBIE (task is terminated)
task exits via do_exit()
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
task is preempted by higher priority task
event occurs and task is woken up and placed back on the run queue
task sleeps on wait queue for a specific event
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)

The state field of the process descriptor describes the current condition of the process.

- **RUNNING and RUNNABLE:**When the CPU executes a process, it will be in a RUNNING state.

- Sleeping:
  The Sleeping state indicates the process is currently waiting on certain resources (like waiting on I/O, waiting on locks, application code making the process to sleep, etc.). There are two types of Sleeping processes:

- **INTERRUPTABLE_SLEEP:** When a process is in INTERRUPTABLE_SLEEP, it will wake up from the middle of sleep and process new signals sent to it.
- **UNINTERRUPTABLE_SLEEP:** When a process is in UNINTERRUPTABLE_SLEEP, it will not wake up from the middle of sleep even though new signals are sent to it.

- **STOPPED:**The STOPPED state indicates that the process has been suspended from proceeding further. In Linux, when you issue the 'Ctrl + Z' command it will issue a SIGSTOP signal to the process.
- **Zombie:** A process will terminate when it calls a 'system exit' API or when someone else kills the process

Process Context vs Interrupt Context:
The process and interrupt context is with reference to the kernel execution, when kernel is working on behalf of a process or it is running some kernel threads it is said to be executing in process context whereas when the kernel is handling some interrupt handler then it is said to be working in interrupt context.

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 registers
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

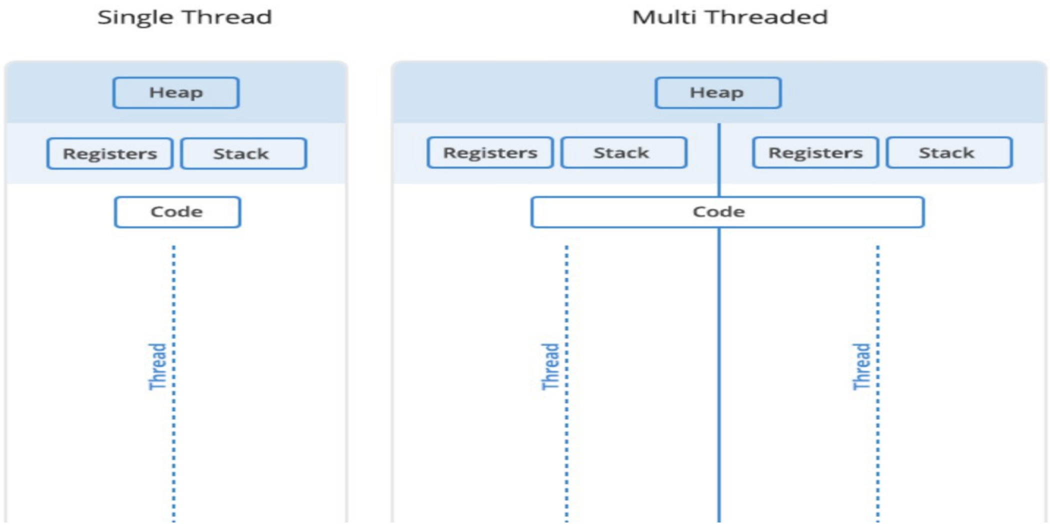
points to the instruction following the last instruction executed, the process resumes
execution from where it was stopped.

An orphan process is a computer process whose parent process has finished or terminated, though it (child process) remains running itself.

A zombie process: or defunct process is a process that has completed execution but still has an entry in the process table as its parent process didn't invoke an wait() system call.

Threads: the units of execution within a program.
Each thread within a process has a 1. unique program counter, 2. process stack, 3. and 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.

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. child process inherits mutex, condition variables, open file descriptor, message queue descriptor.
//page tables are copied and page frames are shared.

fork() vs vfork() vs exec() vs system() vs clone():

Fork : The fork call is used to duplicate the current process, the new process identical in almost every way except that it has its own PID. The return value of the function fork distinguishes the two processes, zero is returned in the child and PID of child in parent process.

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.

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.

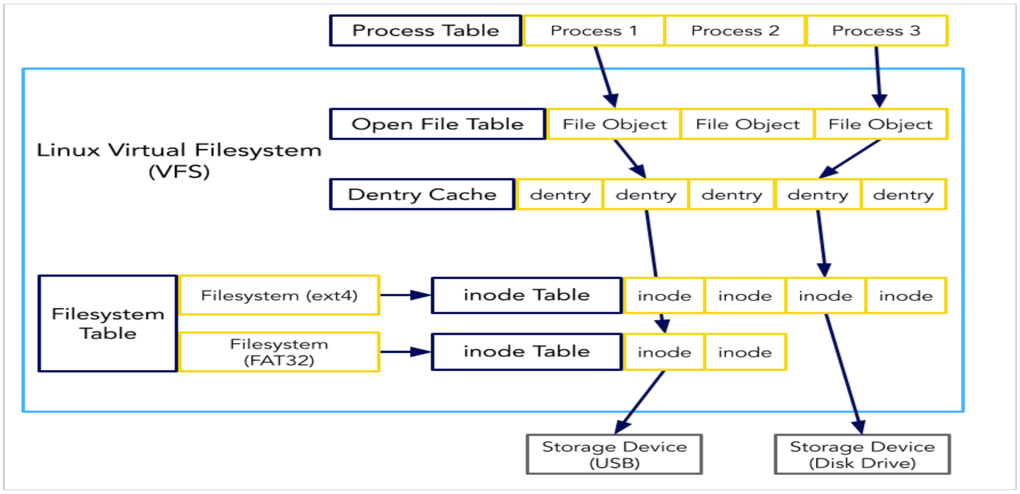
Clone : Clone, as fork, creates a new process. Unlike fork, these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.

Scheduler:

The scheduler is invoked: when there 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.

------------------------------------- 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.

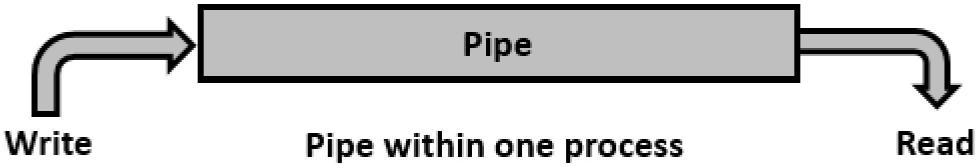


-----------------------------------IPCS --

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*);

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.

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.



#include<unistd.h>

```
int pipe(int pipedes[2]);
```

This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Named/fifo pipes: we can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication.

```
/* Filename: fifoserver.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "MYFIFO"
int main() {
   int fd;
   char readbuf[80];
   char end[10];
   int to_end;
   int read_bytes;

   /* Create the FIFO if it does not exist */
   mknod(FIFO_FILE, S_IFIFO|0640, 0);
   strcpy(end, "end");
   while(1) {
      fd = open(FIFO_FILE, O_RDONLY);
      read_bytes = read(fd, readbuf, sizeof(readbuf));
      readbuf[read_bytes] = '\0';
```

```
    printf("Received string: \"%s\" and length is %d\n", readbuf,
      (int)strlen(readbuf));
    to_end = strcmp(readbuf, end);
    if (to_end == 0) {
      close(fd);
      break;
    }
  }
  return 0;
}
```

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.

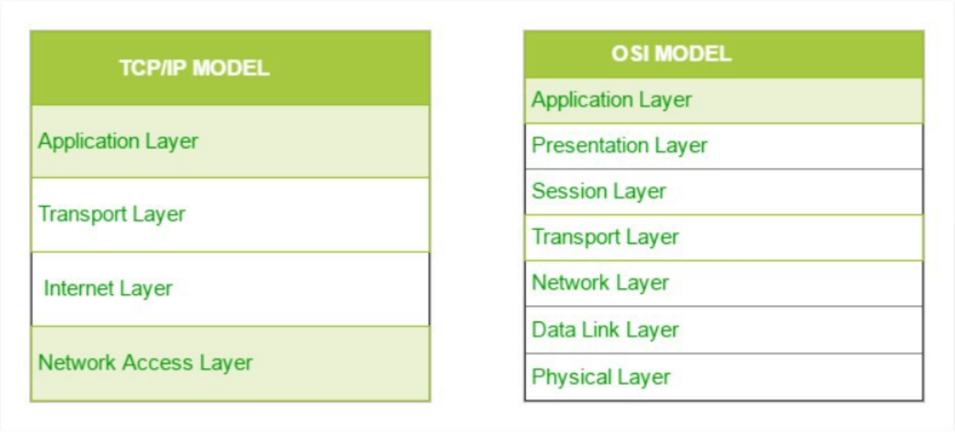
//Connect the socket to the address of the server using the connect() system call.

//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.

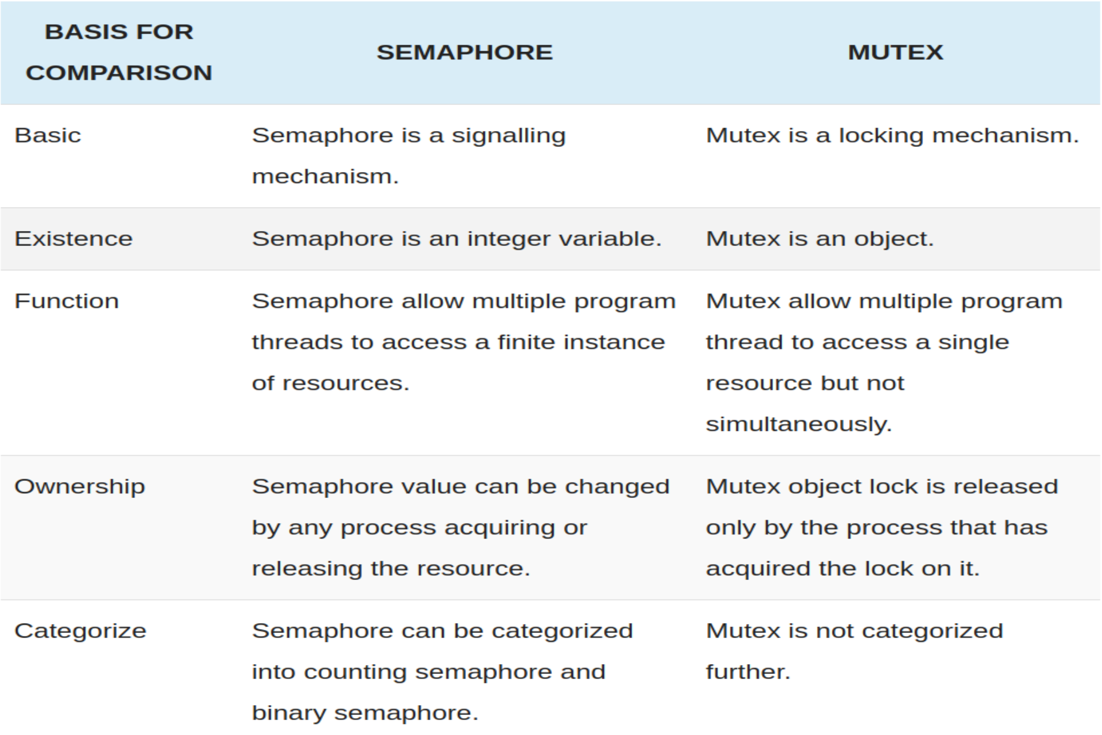
//we should make non blocking calls and can be done by calling socket.setblocking(0)

//TCP/IP nad OSI model:



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.

Condition variable:

pthread_cond_init()
pthread_cond_wait()
pthread_cond_signal()
**********************************scenario for cond_variable**********************

Two threads reading and writing to buffer. Writer thread locks mutex object and perform write operation to buffer and reader thread locks mutex object once it is released by writer thread and read
it. We can implement it through condition 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.

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

----------------Interrupt, exception and system calls------------------

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
needed.

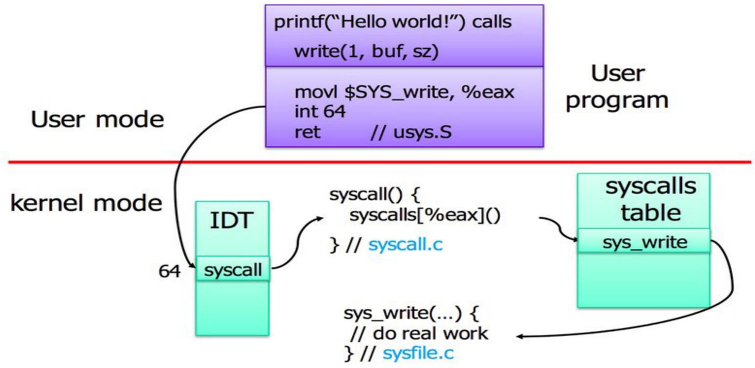
Interrupts and Exceptions:

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.

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.



Maskable and non-maskable interrupt:

Non-maskable interrupt cannot 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

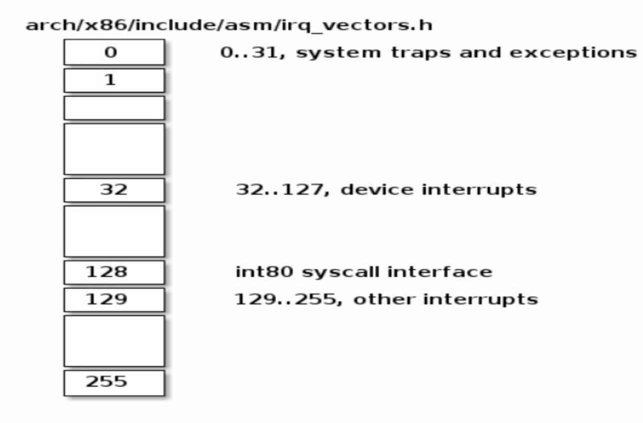
device raises an interrupt on the corresponding IRQn pin

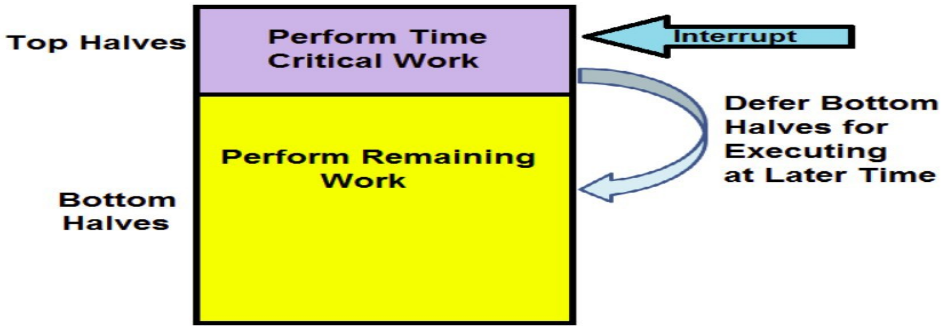
PIC(Programmable Interrupt Controller) converts the IRQ into a vector number and writes it to a port for CPU to read

PIC raises an interrupt on CPU INTR pin

PIC waits for CPU to acknowledge an interrupt

CPU handles the interrupt


arch/x86/include/asm/irq_vectors.h
0
0..31, system traps and exceptions
1
32
32..127, device interrupts
128
int80 syscall interface
129
129..255, other interrupts
255


Top Halves
Perform Time Critical Work
Interrupt
Defer Bottom Halves for Executing at Later Time
Perform Remaining Work
Bottom Halves

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.

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.

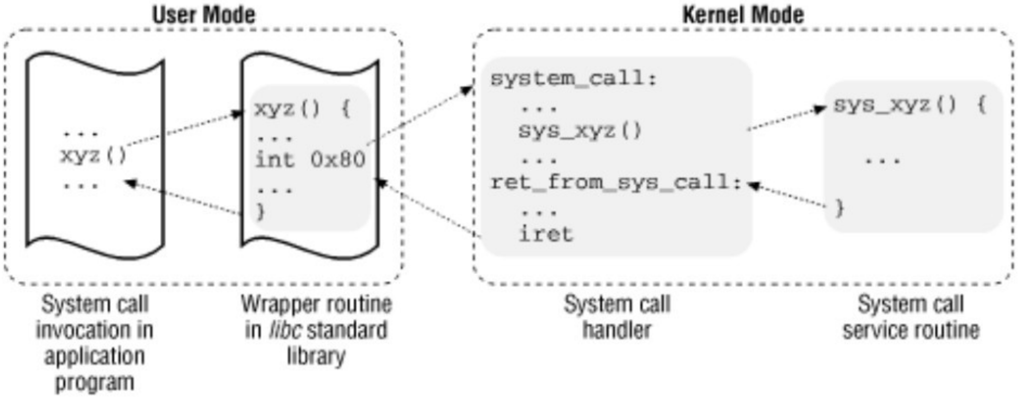
If the earlier selected file is deleted, the hard link to the file will still contain the data of that
file.

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 (software interrupts)
2) Device sends interrupt (hardware interrupt)
3) CPU raises exception.(Exception)

Figure 8-1. Invoking a system call



```
...
xyz()
...
```

```
xyz() {
...
int 0x80
...
}
```

```
system_call:
  ...
  sys_xyz()
  ...
ret_from_sys_call:
  ...
  iret
```

```
sys_xyz() {
  ...
}
```

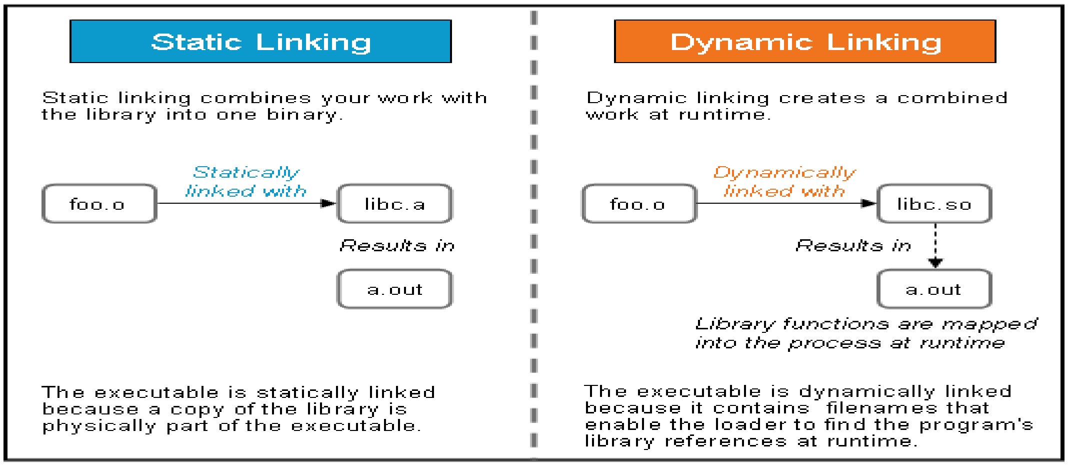
User Mode
Kernel Mode
System call invocation in application program
Wrapper routine in libc standard library
System call handler
System call service routine

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.

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

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.



Static linking combines your work with the library into one binary.

The executable is statically linked because a copy of the library is physically part of the executable.

Dynamic linking creates a combined work at runtime.

Library functions are mapped into the process at runtime

The executable is dynamically linked because it contains filenames that enable the loader to find the program's library references at runtime.

Whenever there is change in static lib, code must be compiled 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.

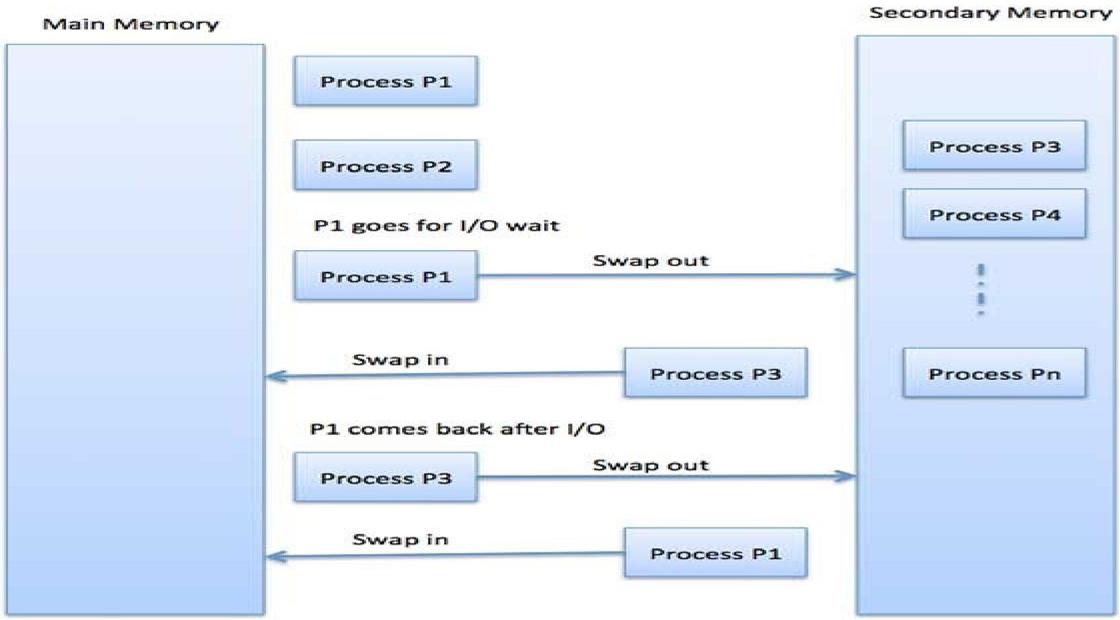
```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Static library creation:

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

Swapping:

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.



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.

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.

External fragmentation: Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

Internal fragmentation: Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

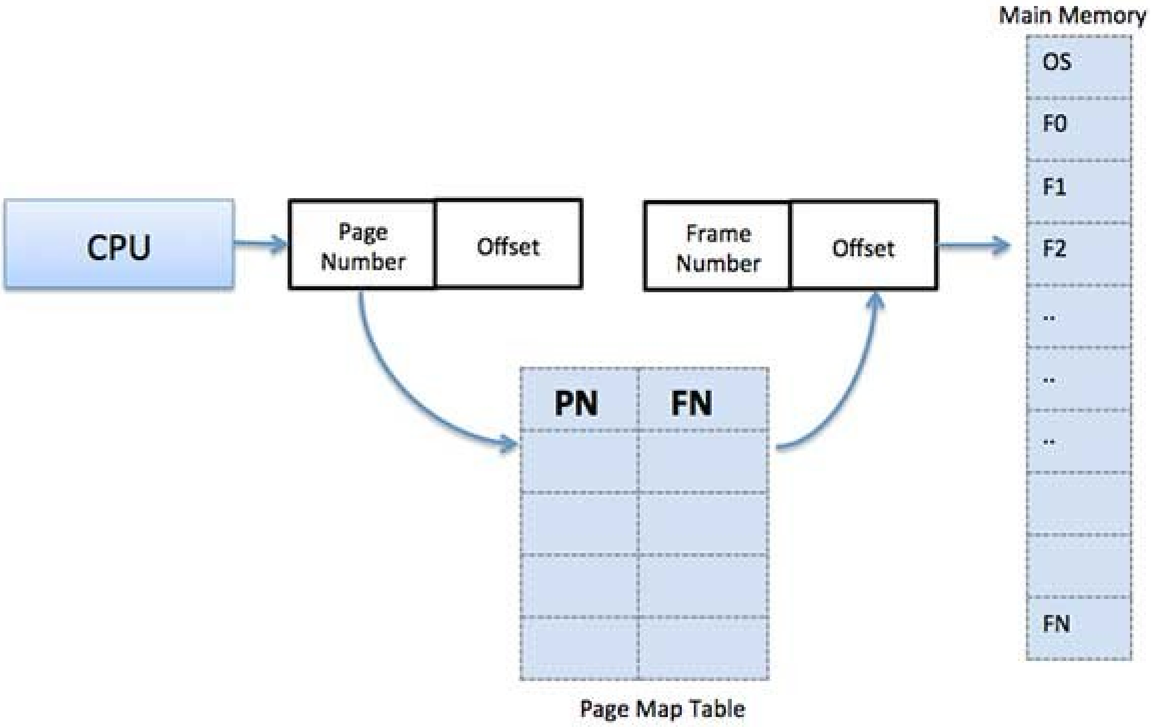
Address Translation

Page address is called **logical address** and represented by **page number** and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset



Advantages and Disadvantages of Paging

Here is a list of 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.

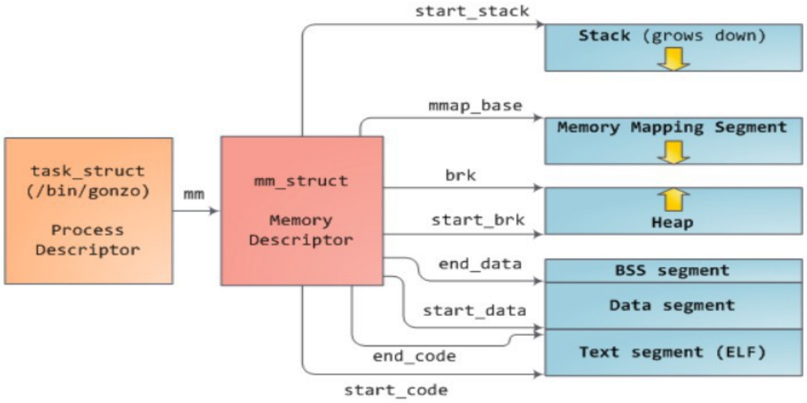
Segmentation: is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.
When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Virtual memory:
A computer can address more memory than the amount physically installed on the system. This extra memory is actually 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 quite similar to 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.

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 transferand interrupted only after entire block has been transferred.



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.

A 32-bit system can access 232 memory addresses, i.e 4 GB of RAM or physical memory. A 64-bit system can access 264 memory addresses, i.e actually 18-Quintillion bytes of RAM.

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.

---------------------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(**gdb b main/ gdb b file.c:233)**
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

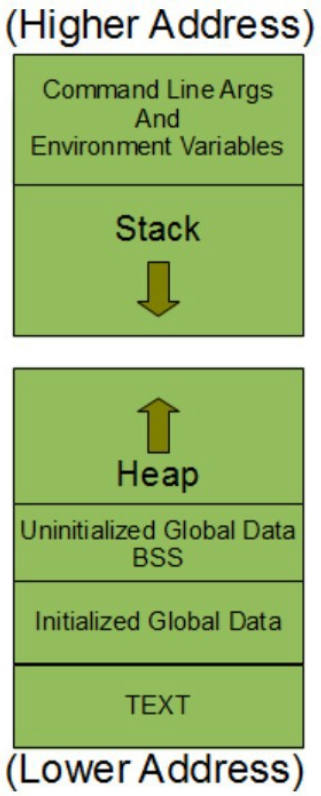
19. **What are the processes with PID 0 is Sched and PID 1 is init(**process primarily responsible
for starting and shutting down the system**)**

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?

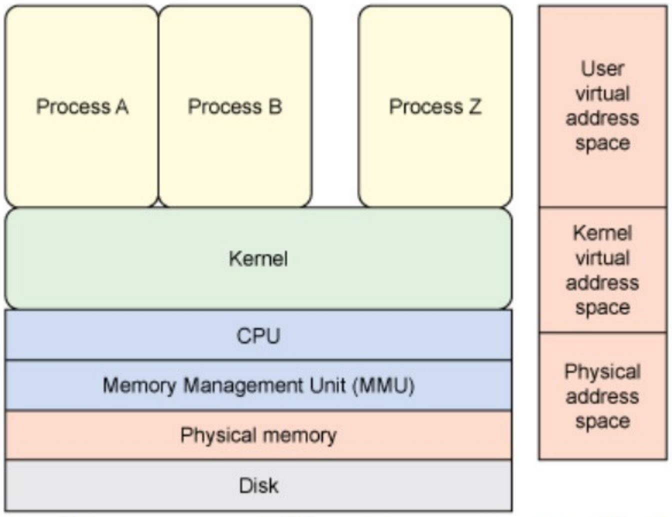
In general, each *thread* has its own registers (including its own program counter), its own stack pointer, and its own stack. Everything else is shared between the threads sharing a process.

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


Process A
Process B
Process Z
Kernel
CPU
Memory Management Unit (MMU)
Physical memory
Disk
User virtual address space
Kernel virtual address space
Physical address space

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

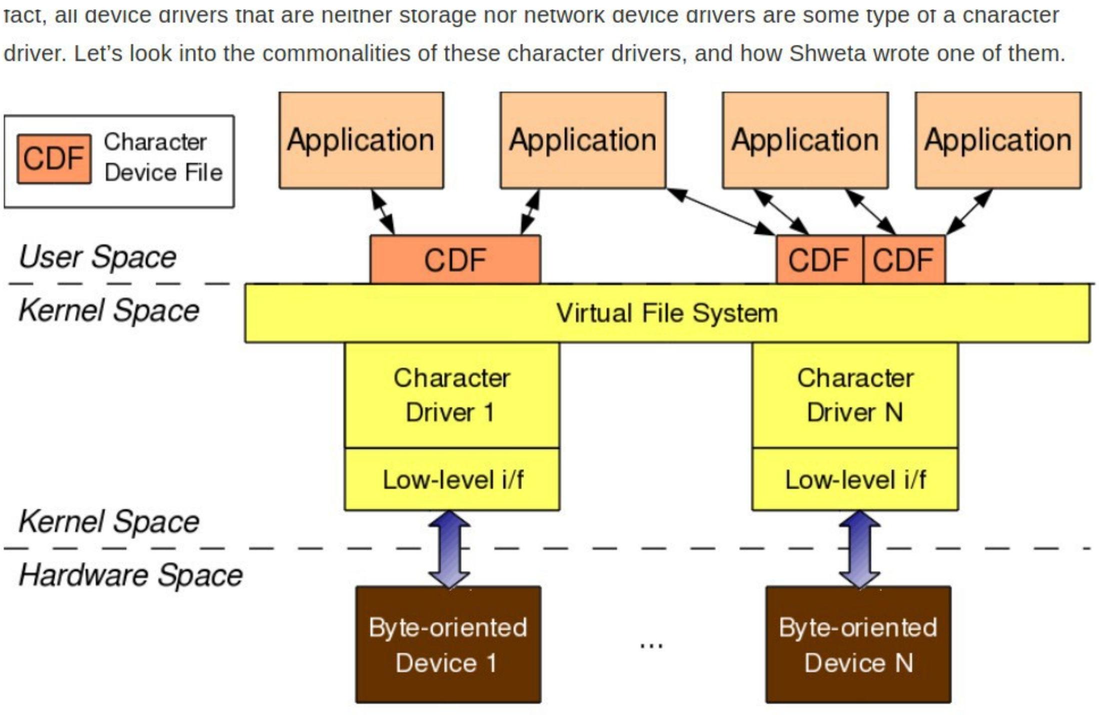
Steps to invoke device driver:

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.



fact, all device drivers that are neither storage nor network device drivers are some type of a character driver. Let's look into the commonalities of these character drivers, and how Shweta wrote one of them.

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.