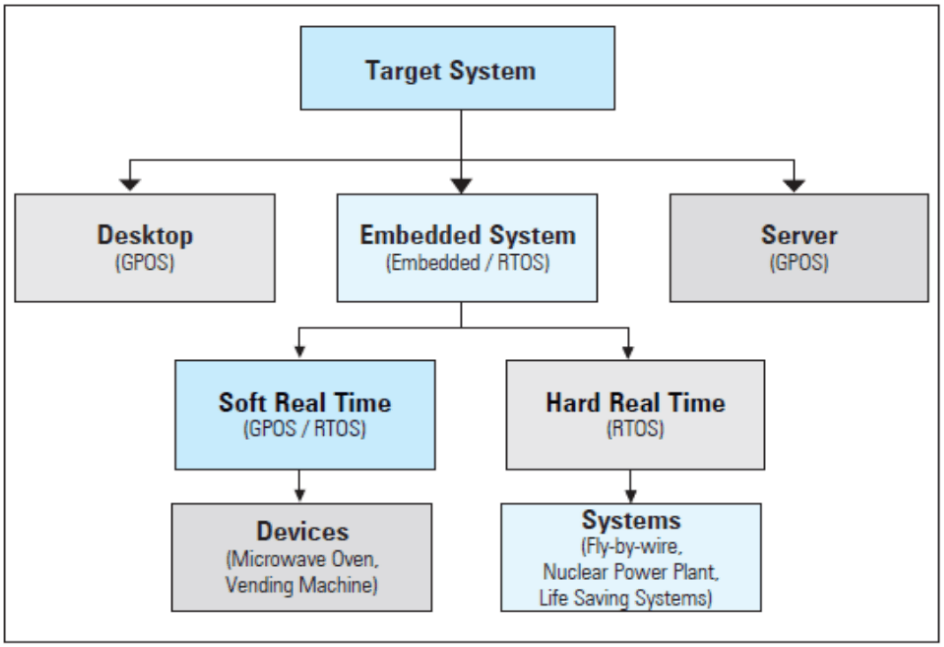
SS Notes written by senior on Notion (absolutely top-notch stuff)

31/08/21

Structure of Computing System

Memory
Processor
Input & Output Devices

Types of Systems



GPOS - General Purpose OS

RTOS - Real-Time OS - Deadlines are very important, critical, such as in medical devices, monitors used in nuclear power plants, etc. So any deadlines cannot be missed which is why a separate OS type exists for it called RTOS.

Embedded System - Dedicated system that performs specific tasks.
(Embedded Systems can be non-real-time systems as well like printers)

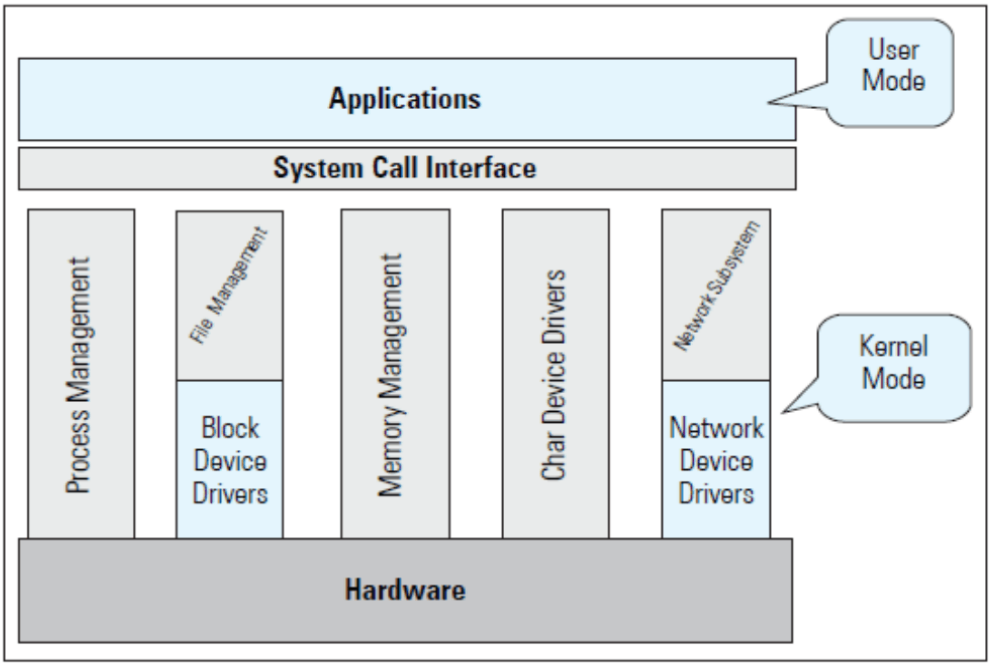
Soft Real Time Systems - Missing deadlines is not critical but it negatively affects the throughput of the system. Linux supports Soft RTOS by default, to support Hard RTOS we need to apply patches to Linux.

Throughput - Amount/number of tasks you can perform per second (or in a given unit of time).

You can execute real-time tasks along with non-real time tasks. If you're on Linux, go to terminal and type "top", it will show running processes. In the priority column, look for processes that have "rt" as the priority value. Those are real-time tasks and they have the highest priority, whereas all other tasks are non-real time tasks.
However, a system that executes only real-time tasks has to be separately configured to do just that. This configuration involves making the kernel as small as possible by removing redundant stuff that's not required for that RT system.

GPOS

[Remember all the diagrams and concepts in this section for the exam]



Note which all components come under the kernel!

Applications (which execute in User Mode) communicate with the Kernel Mode services and devices via the System Call Interface. And the Kernel Mode services are connected with the hardware. All system calls are executed in the kernel mode.

Network subsystem: 7 OSI Layers

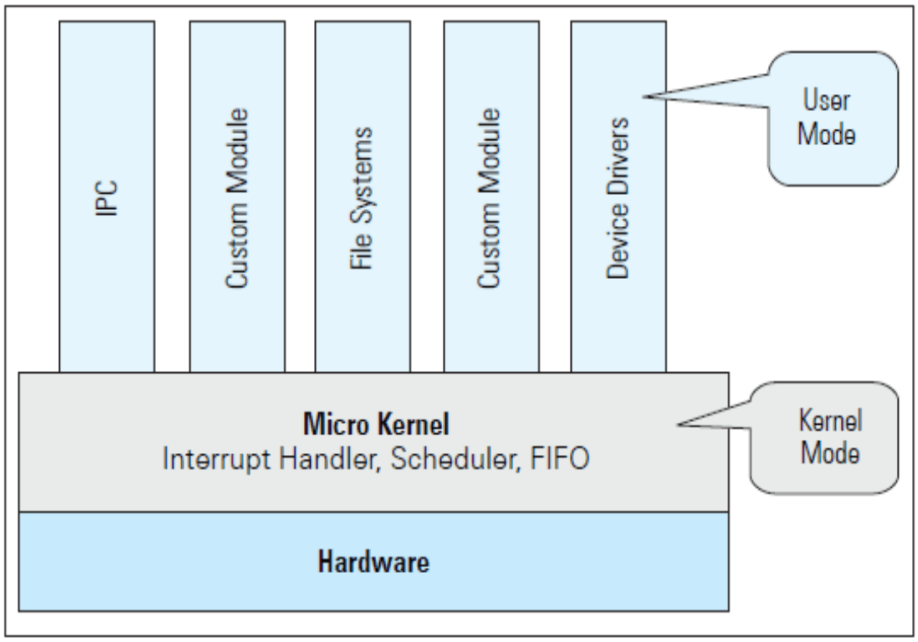
Device drivers: Software that's required to make a device be recognized and operated by the system. Depending on which type of device it is, we get three types of device drivers.

Block devices are **nonvolatile mass storage devices whose information can be accessed in any order**. Hard disks, floppy disks, and CD-ROMs are examples of block devices.

Character devices are **devices** that do not have physically addressable storage media, such as tape drives or serial ports, where I/O is normally performed in a byte stream. Random access is typically not found here and there's no buffer storage (cache) which is present for block devices.

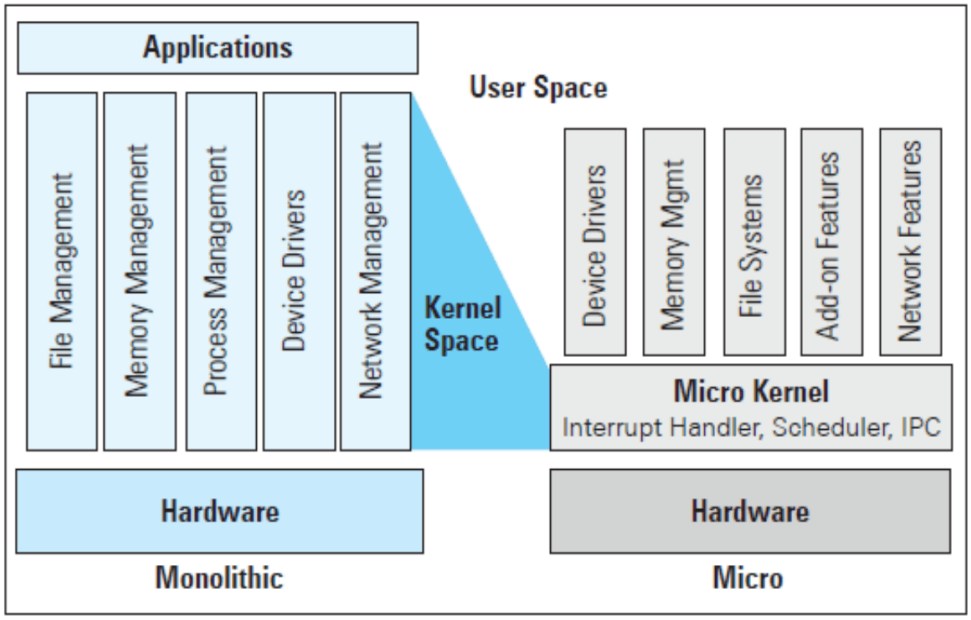
A network device driver is **a kernel-level driver that connects a communications device to the IRIX TCP/IP protocol stack using** the ifnet interface established by BSD UNIX.

Micro Kernel



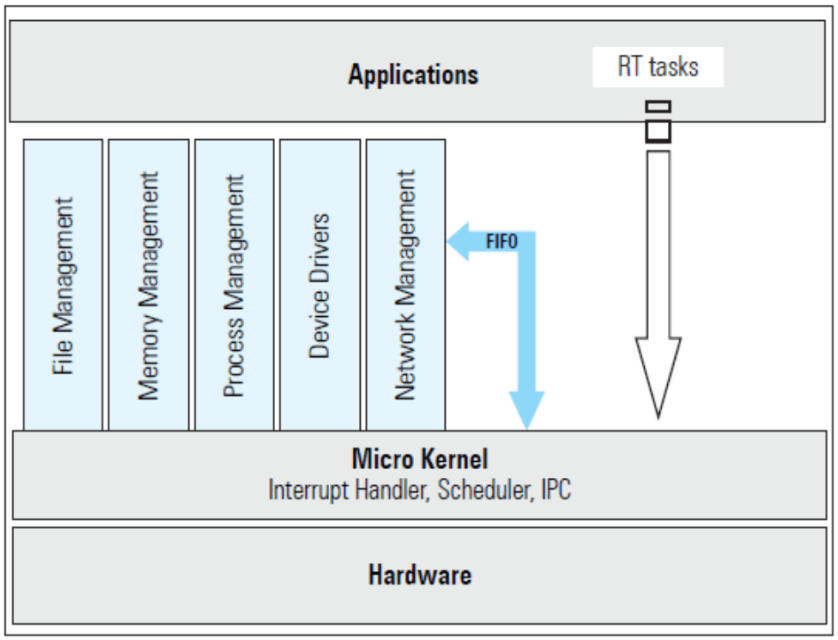
Kernel is very minimal.
Contains Interrupt Handler, Scheduler and FIFO which is named pipe for communication.
It is customized for dedicated tasks.
If you want to add extra services like file management, memory management, etc. then you can install it in the custom modules present in user mode/space and not in the kernel space.
Gives you deterministic behaviour from the system.

Monolithic vs Micro Kernel



This basically shows you how small the Micro Kernel is in comparison to the Monolithic Kernel because Monolithic places all of the services in the Kernel space.

Hybrid Kernel



This is a modification on the micro kernel to support hard real-time systems.

Here, the entire Linux kernel (the GPOS kernel parts like file management, memory management, etc.) is executed as a process in the micro kernel.
To actually make this system work, we have to apply patches to the kernel.
Now when a real-time task comes in, it will use the RT API calls to the kernel.
The kernel will be able to recognize that it's an RT task and it will get executed directly via the kernel without having to go through the GPOS. If an RT task needs any resources like devices or stuff that's in the GPOS "kernel" then it can communicate with it via the FIFO mechanism in the Micro Kernel.
Only if there's no RT task will the microkernel take tasks in via the GPOS "kernel" using FIFO.

Device controller is the electronic part of a device that controls the device electronically by setting status registers, flags, etc. whereas device driver is software that's required to run it on a system. The other part of a device is the actual mechanical part.

Operating System = Kernel + Utilities
And it is in these utilities where most Linux distributions differentiate themselves from each other. For example, RHEL (Red Hat) is built specifically for developing enterprise applications and similarly, other distributions provide tailor-built utilities for specific purposes.

The vanilla Linux kernel can also be modified. In which case, it has to be recompiled in order to get you the new build. So different distributions can also differ in what their kernel contains.

Kernel Source Code

The kernel code contains architecture dependent as well as architecture independent code

- **Machine-dependent code deals with**
  - **Low-level system startup functions**
  - **Trap and fault handling**
  - **Low-level manipulation of runtime context of a process**
  - **Configuration and initialization of hardware devices**
  - **Runtime support for I/O devices**

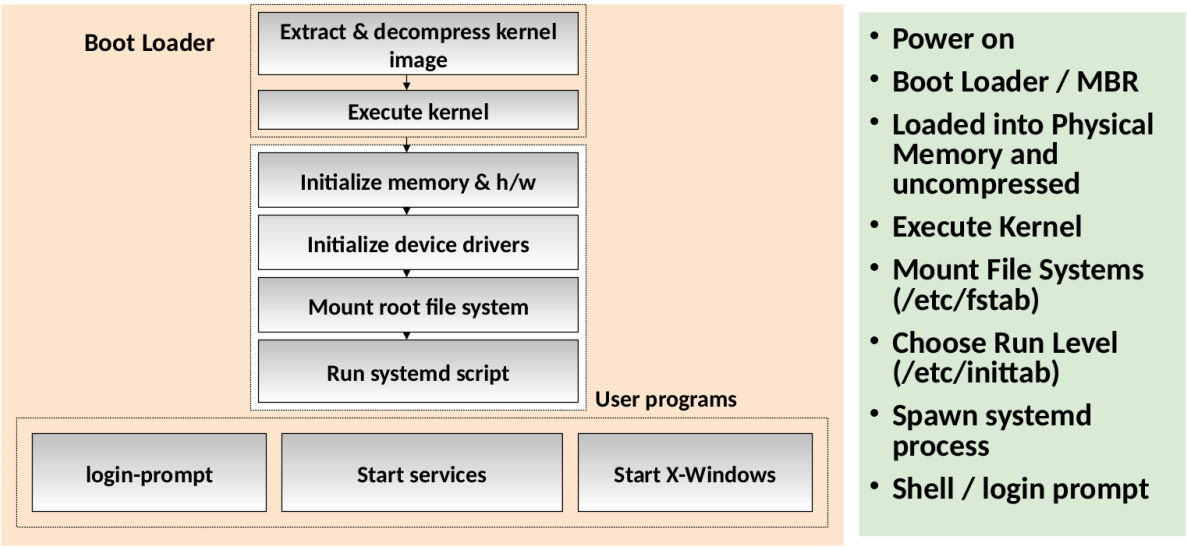
- **Machine-independent code deals with**
  - **System call handling**
  - **The file system: files, directories, pathname translation, file locking, and I/O buffer management**
  - **Terminal handling support: the terminal-interface driver and terminal line disciplines**
  - **IPC facilities**
  - **Network communication support**

Low-level manipulation of runtime context of a process is mainly referring to context switching which is machine-dependent.

95% of the Linux kernel is written in C language (machine-independent) and 5% is written in assembly language (machine-dependent).

When you install Linux kernel, it detects which architecture system you are installing it on depending on which the relevant system-dependent stuff is installed.

Booting Procedure



Kernel Image is a zip file, so that's why it has to be extracted and decompressed before execution.
Note: There is an extra step that's not shown here and that's loading the **ramfs,** it is a memory file system that will contain the keyboard driver, monitor driver, etc. which is why you're able to see the boot loader and are even able to choose options via keyboard input.
Once the kernel is executed, the ramfs is removed from memory.

Systemd / Init Process

Daemon process that supports parallelism, i.e. multi-core systems. It provides **a standard process for controlling what programs run when a Linux system boots up.**
Earlier there used to be the init process but it only supports single-core systems, so that's why systemd was introduced.
It is the first process that is run on the system and has pid=1. You can check this by running the command "pstree". You'll see that all processes are spawned from the systemd process call.

Do note that there'll be multiple systemd processes, each one handles a session which is a process group, so the associated systemd process will handle that process group. It's just that the first process executed is also a systemd process.

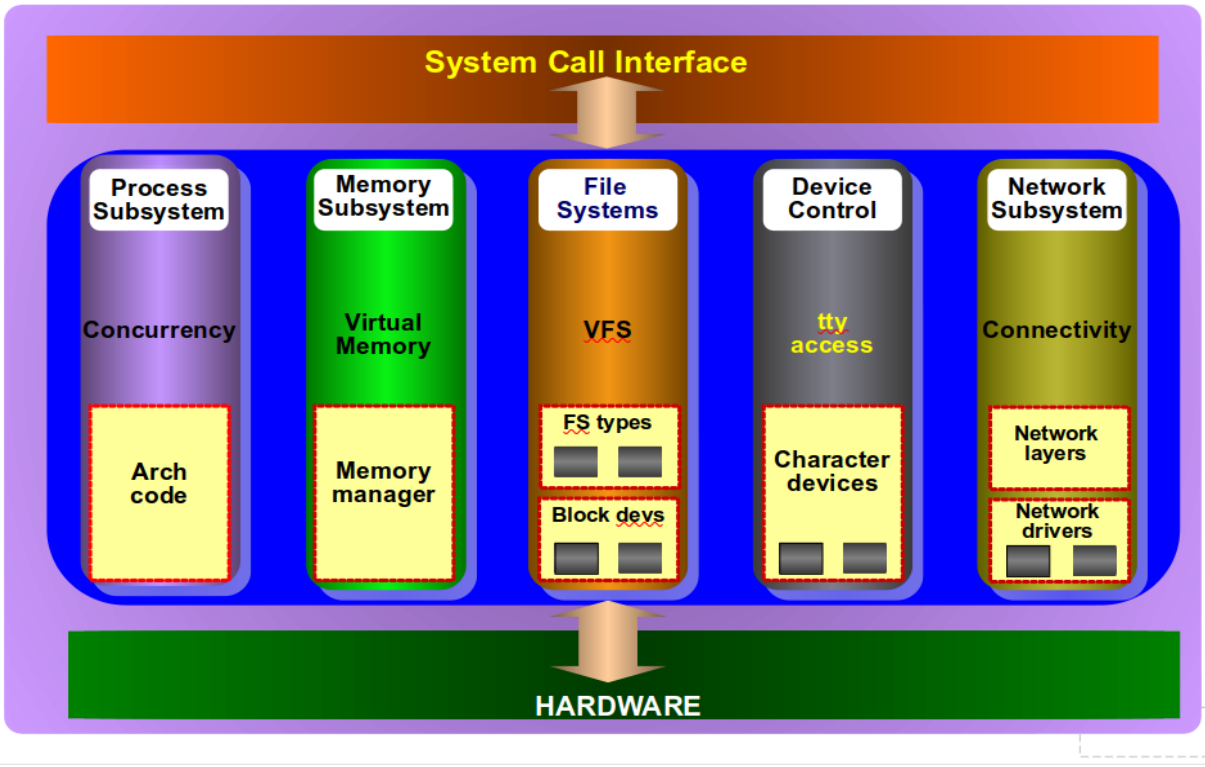
Root file system is the parent of all file systems in a Linux system. Root of the file tree structure, indicated by '/'.

X-Windows = GUI of the system basically

Monolithic kernel keeps the kernel space as one giant block. Doesn't this lead to a disadvantage over microkernel ? Like if 1 service fails, the entire OS crashes.
NO!
The concept used here is **kernel hardening**, in this, if a service fails, the OS identifies which kernel thread has failed and only restarts that thread, instead of rebooting the entire system.
The term kernel hardening refers **to a strategy of using specific kernel configuration options to limit or prevent certain types of cyber attacks**.

Note that kernel space and user space are not on the same address space. They are neatly partitioned into completely disjoint partitions. So you cannot end up on a kernel space block while accessing user space blocks.

Linux Kernel Architecture

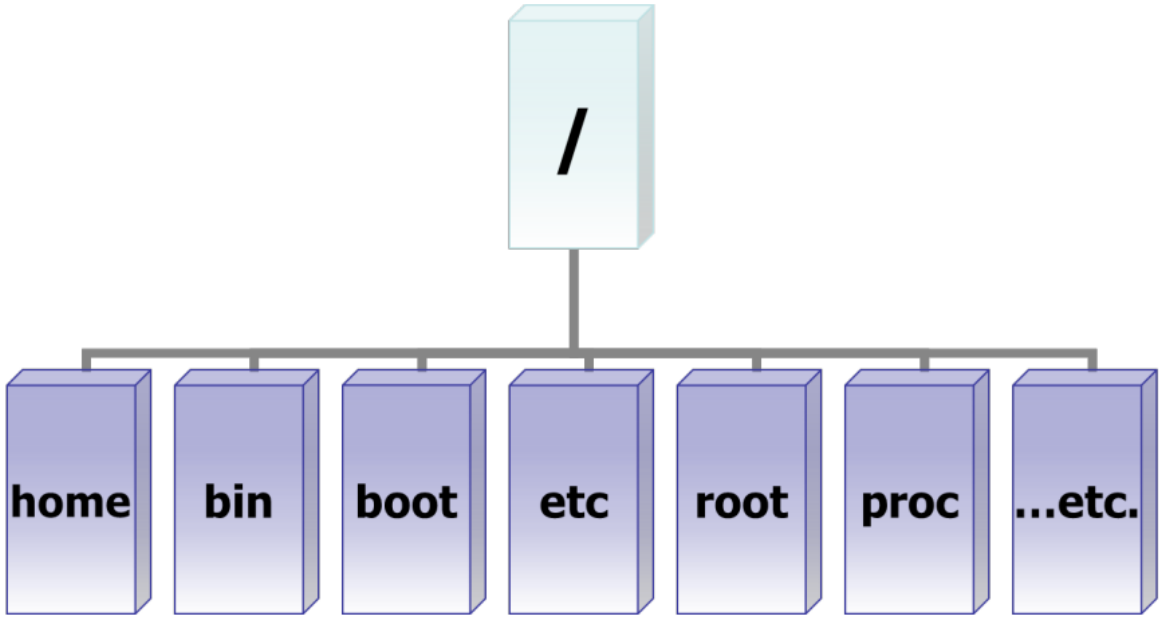


Process subsystem provides concurrency manager which is architecture dependent code.
Virtual memory is used in the memory subsystem.

File systems use the virtual file system and the block devices have buffers which are not present with character devices because they read byte streams.
The small black block is an indicator for device drivers => dynamically loadable modules.

File Management

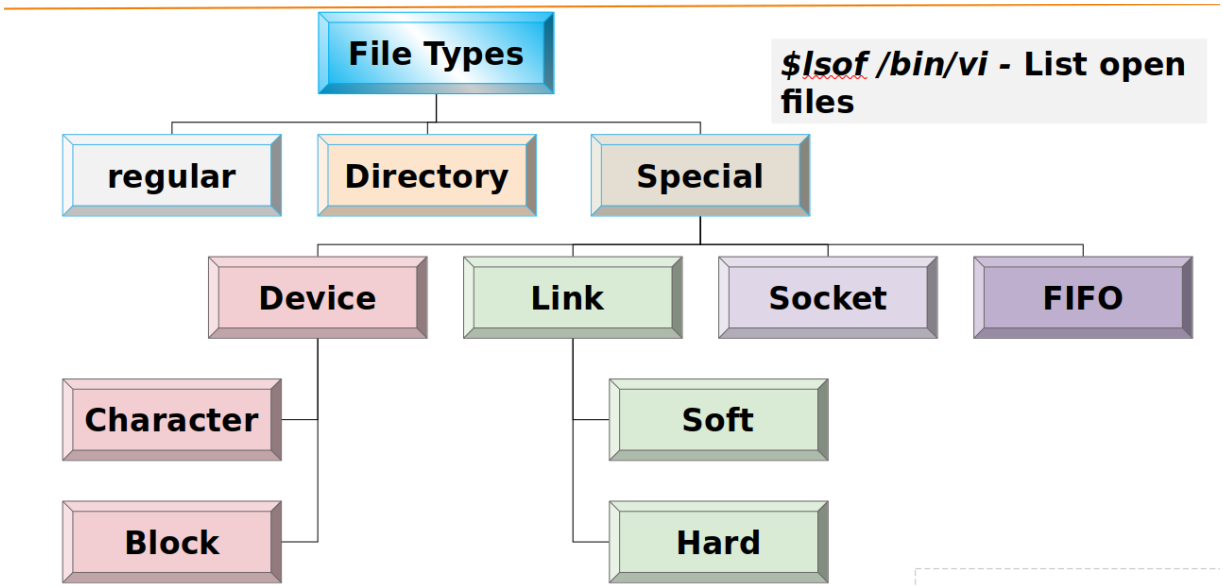
Inverted File Structure of Linux



home: Stores user files
bin: Stores executable commands in single-user mode.
boot: Stores boot loader files, e.g. kernel, initrd, etc.
etc: Host-specific system-wide configuration files.
root: Root user's home directory.
proc: Virtual filesystem (pseudo fs) providing process and kernel information as files.

The advantage of the inverted file structure is that from a system admin point of view it makes maintenance easy as each file system has its own purpose.
So if a particular module fails then only that module needs to be checked because of how the entire file system is partitioned.

File Types in Linux



Device special files: Stores two numbers, major and minor. Major identifies type of device (character or block) and minor identifies the instance of that device.
There is no size associated with these files. Because it is just an interface for the user/system to communicate with the actual device it is pointing to. So whatever data is sent or received is stored in the device or with the system and no storage happens with the device special files.

Links: Like shortcuts in Windows.
Soft Link is essentially a shortcut, it is just a reference to the file it was created as a shortcut to. So if you delete the file, then the link is a dangling pointer.
The problem with dangling pointers is that it is referring to a memory location that has been freed. So it is possible that this location can get reallocated to some other memory location and the dangling pointer is still referring to this location. So **using the dangling pointer** to do anything can result in unpredictable behaviour and most often results in **segmentation faults** and overall it causes **memory leaks.**

Hard Link is a pointer that points to the actual file in memory. So modifying the file via the link pointer also reflects in the pointer that was used for creating the hard link and vice versa.

Hard link and file have the same inode number whereas soft link and file have different inode numbers which indicates that two links connected by a hard link are pointing to the same file, so deleting one link doesn't really change anything.

ln source target - Creates hard link
ln -s source target - Creates soft or symbolic link

An inode number is **a uniquely existing number for all the files in Linux** and all Unix type systems. When a file is created on a system, a file name and Inode number is assigned to it.

Identifying type of a file,
Execute “ls -l” => The first character for each file in the output tells you its type.
- = Regular File, d = Directory
c = Character Device, b = Block Device
l = Soft Link
Hard Link = Whatever type of file it is linked to (because a hard link is just a file pointer)
S = Socket, p = FIFO (pipe)

Note: ls -l output contains the size parameter but for Device Special files it gives the major and minor numbers.

03/09/21

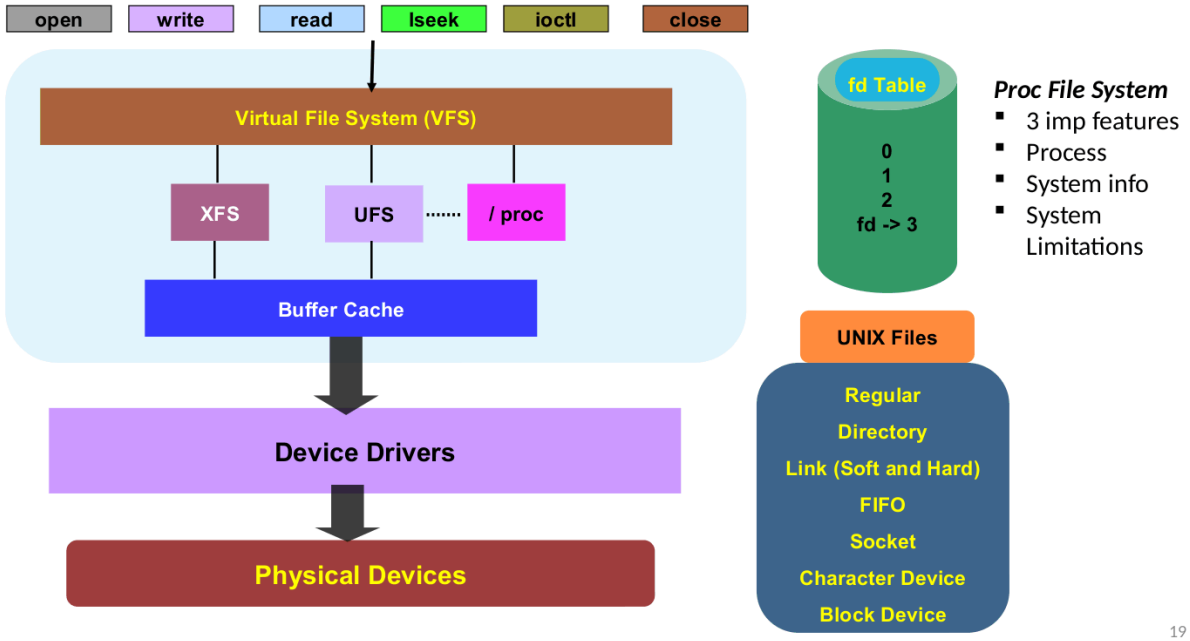
File Systems

- **Facilitates persistent storage and data management**
- **Facilitates file related system calls.**
- **Different types of file system for different needs –depends on implementation**
  - **A logical file system appears as a single entity to the user process, but it may be composed of a number of physical file systems.**

- **VFS**
- **XFS**
- **ext4**
- **UFS**
- **proc**
- **msdos**
- **iso9660**
- **Vfat**
- **Aufs**
- **…….**

The last sentence talks about how a virtual/logical file system can appear as one partition/entity despite being on different physical partitions in the disk.

File System Architecture

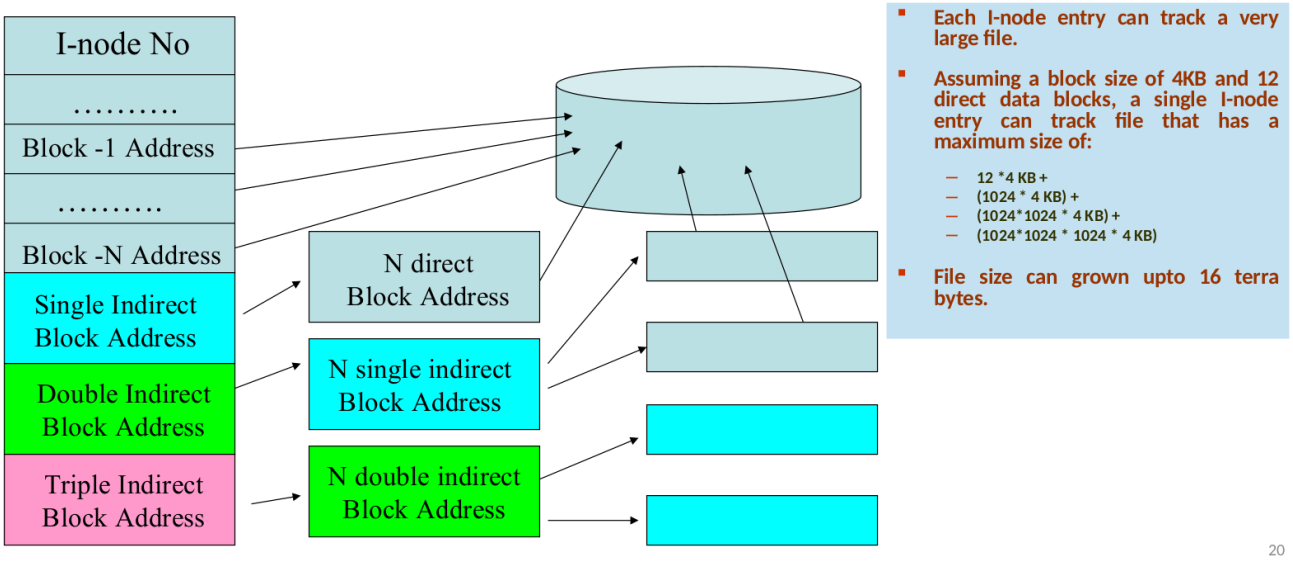


This is the advantage of Virtual File Systems. We can use the same set of system calls to interface with the file system no matter which file system is actually being implemented, for example, XFS, UFS, /proc, etc.

File Descriptor Table
Each process has a separate FD table.
There are 3 standard file descriptors, 0 = standard input, 1 = standard output, 2 = standard error
If you open any file, then its fd value will be set to 3 because that's the next available fd value, if you create another file in the same directory then its fd value will be set to 4 and so on.

More on buffer cache later.

ext File System



[Definite numerical on this in exam]

Each data block is 4 KiB by default but you can change this.
You can create a file with some text in it whose size is in bytes, but if you run the “du” command on it, you will see that the size of the file is 4 KiB.

If you create a file whose size is less than or equal to 12 * 4KiB (assuming direct data block consists of 12 entries), then directly 12 blocks will be allocated to that file without any kind of indirect addressing, so all addresses will directly point to actual data blocks.

If the file size is greater than 12 * 4KiB, then a single indirect block address is created which points to a direct block address block and each address in this block points to an actual data block.
Assume that addresses are of size 4 bytes, then how many data blocks can be addressed by a single Indirect Block Address?
Block size = 4 KiB [For Direct Block Address]
Address size = 4 bytes [Size of each address stored in the direct address block]
Therefore, total number of data blocks addressed = 4 KiB / 4 bytes = 1 KiB = 1024 blocks.
So a Single Indirect Block Address can address 1024 blocks, i.e. 1024 * 4 KiB of storage.

If the file size is greater than 1024 * 4 KiB, then double indirect block addressing is used. The Double Indirect Block Address points to 4 KiB Single Indirect Block Address which then points to a Direct Block Address block and then finally actual data blocks.
Each Single Block Address block can access 1024 data blocks and each Double Block Address block can access 1024 Single Block Address blocks.
So in total we can address, 1024 * 1024 blocks, i.e. 1024 * 1024 * 4 KiB of storage.

If file size is even greater than this, then there comes triple indirect block addressing which will be able to address, 1024 * 1024 * 1024 blocks, i.e. 1024 * 1024 * 1024 * 4 KiB of storage.

If file size is greater than 16 TiB, then it cannot be stored and would have to be broken into atleast two halves for it to be addressable by ext4.

Numericals can be like, number of direct, indirect blocks are given and find max size of file that can be stored in this. How many files can be stored in this if one file occupies one block? How many inode numbers will be present (which is just the same as asking how many files can be stored as each file has a unique inode number).

Kilo vs Kibi

Kilo = Powers of 10
Examples: KB, MB, GB, etc.

Kibi = Powers of 2
Examples: KiB, MiB, GiB, etc.

Copy-On-Write Technique (COW)

Copy a file only when a write operation is to be performed on it, otherwise just create a link to it while reading.

A more formal explanation:
Copy-on-write (sometimes referred to as "COW") is an optimization strategy used in computer programming. The fundamental idea is that if multiple callers ask for resources which are initially indistinguishable, you can give them pointers to the same resource. This function can be maintained until a caller tries to modify its "copy" of the resource, at which point a true private copy is created to prevent the changes becoming visible to everyone else. All of this happens transparently to the callers. The primary advantage is that if a caller never makes any modifications, no private copy needs to be created.

[Note: Callers here can refer to processes, users, etc.]

XFS

When you create a file system, Linux creates a number of blocks on that device.

- Boot Block
- Super-block
- I-node table
- Data Blocks

- Linux also creates an entry for the "/" (root) directory in the I-node table, and allocates data block to store the contents of the "/" directory.

- **The super-block contains info. such as**
  - a bitmap of blocks on the device, each bit specifies whether a block is free or in use.
  - the size of a data block
  - the count of entries in the I-node table
  - the date and time when the file system was last checked

The boot block contains information on how to boot the system, bootloader, kernel's location

- Each device also contains more than one copy of the super-block.
- Linux maintains multiple copies of super-block, as the super-block contains information that must be available to use the device.
- If the original super-block is corrupted, an alternate super-block can be used to mount the file system.

- The I-node table contains an entry for each file stored in the file system. The total number of I-nodes in a file system determine the number of files that a file system can contain.
- When a file system is created, the I-node for the root directory of the file system is automatically created.
- Each I-node entry describes one file.

- **Each I-node contains following info**
  - file owner UID and GID
  - file type and access permissions
  - date/time the file was created, last modified, last accessed
  - the size of the file
  - the number of hard links to the file
  - Each I-node entry can track a very large file

- **Ordinary file creation**
  - The kernel allocates space in the hard disk. The text in the file is stored one character per byte of memory.
  - The file holds these characters and nothing more. It does not contain any information about its beginning or ending.
  - An inode entry is created on a section of the disk.

Device Special Files

- **A device special file describes following characteristics of a device**
  - **Device name**
  - **Device type (block device or character device)**
  - **Major device number (for example '2" for floppy, "3" for hard-disk)**
  - **Minor device number (such as "1" for "hda1")**

- **Switch Table - Unix kernel maintains a set of tables using the major device numbers.**
- **The switch table is used to find out which device driver should be invoked**
- **For example : fd –> file table –> inode table –> switch table –> device drivers**

When you create a device special file, it will have a file descriptor which will point to the file descriptor table which will point to the file table (this is true for regular files as well) which will point to the inode table which points to the switch table (this is only true for device special files) and using this switch table entry, it is determined which device driver should be invoked to access the device.
[More on all these tables later]

Note: Minor number identifies the instance of a device and this can go to a maximum of 255. So a device driver can handle a maximum of 256 instances of the same device type, e.g. 256 printers, 256 hard disk partitions, etc.

Note: The device special files are, character device, block device, FIFO and Socket. Among these FIFO is not actually linked to a device and so is actually a pseudo device special file!

$mount command

- Each file is located on a file system.
- Each file system is created on a device, and associated with a device special file.
- Therefore, when you use a file, Unix can find out which device special file is associated with that file and send your request to corresponding device driver.

- Each file system must be mounted before it can be used. Normally, all file systems are mounted during system startup depending on */etc/fstab* entry
- (*blkid*, *lsblk*, *lsblk* –help)
- root file system is mounted by default
- It is possible to mount a file system at any time using the "mount" command.

etc/fstab file

This configuration file stores all the partitions in the system and is read at boot time to mount all the partitions that are specified in it at the specified mount point, like mount root partition at '/', home partition at '/home', swapfile, etc.

- The "dev" directory contains names of each device special file.
- Each file system has a "/" (root) directory. However, once a file system is mounted, it's the root directory that is accessed through mount point.
- A file system is mounted typically under an empty directory. This directory is called the "mount point" for the file system.

- When a file system is mounted, the system reads the I-node table and the super-block into memory.
- The in-memory I-node table is used when a process tries to access a file.
- If kernel does not find an entry in this I-node table, it reads the I-node from the on-disk I-node table into in-memory I-node table.
- File system can be unmounted using umount cmd

- You can use mount command to find how many file systems are mounted, and what is the mount point for each file system :

Screen shot of mount command

```
$ mount
/dev/sda6 on / type ext4 (rw)
none on /proc type proc (rw)
```

Running mount command gives you all the file systems mounted.

procfs

The proc file's location is none (in mount command output) because it is a pseudo file system (completely virtual), so it has no physical partition.

It stores necessary information about currently running processes, system information and system limitations => Important for MCQs

It is created on the fly and not stored on disk (whereas inode table is stored on disk).

If you run a process, then it is stored as a file with filename as pid under the /proc/ folder.
This folder will contain various metadata related to the process.

To check the status of a process you can do,
“cat /proc/pid_of_process/status | head”
[Head for printing only first 10 lines]

Get cpu info using “cat /proc/cpuinfo”
Get memory info using “cat /proc/meminfo”

Get info about interrupts using “cat /proc/interrupts”

Watch command

You can see how quickly interrupts are raised and dealt with by creating a watch command on the interrupts file.
“watch -n 0.1 cat /proc/interrupts”
Watch the output of “cat /proc/interrupts” for changes every 0.1 seconds

The watch command is thus pretty useful for monitoring changes in various commands like interrupts, ps, du, etc.

Running commands in the background

Attach & at the end of any command to make it run as a background process
Example, “./a.out &”
You can only see this in action if a.out is for a C file that runs infinitely or something that takes a while. It’ll be run as a background process. So you can use that same terminal window to do anything else or even check info about that process via, “ps pid_of_background_process”.

ipcs command

ipcs shows information on the inter-process communication facilities for which the calling process has read access. By default it shows information about all three IPC resources: shared memory segments, message queues, and semaphore arrays

Buffer Cache

- The file system also maintains a buffer cache.
- The buffer cache is stored in physical memory (non-paged memory).
- The buffer cache is used to store any data that is read from or written to a block-device such as a hard-disk, floppy disk or CD-ROM.
- It reduces disk traffic and access time

- If data is not present in buffer cache
  - the system allocates a free buffer in buffer cache
  - reads the data from the disk
  - stores the data in the buffer cache.
- If there is no free buffer in the buffer cache
  - the system selects a used buffer
  - writes it to the disk
  - marks the buffer as free
  - allocates it for the requesting process.

- While all this is going on, the requesting process is put to wait state.
- Once a free buffer is allocated and data is read from disk into buffer cache, the process is resumed.
- A process can use the sync() system call to tell the system that any changes made by itself in the buffer cache must be written to the disk.

31

There is no fixed size for the buffer cache. Whatever space is free in the RAM (after the space consumed by the processes in execution) is allocated to the buffer cache, so it depends on the system load.

Why do we use buffer caches?

Because otherwise, for each read or write we will have to communicate with disk memory which means waiting for I/O, this means that CPU becomes idle and will switch to another task, run ISR (Interrupt Service Routine) and do something else. Once data becomes available, it will again have to switch back to this task, move the data into main memory and then finally perform operations. All of this is skipped if a sufficient amount of data (one data block) is fetched in one go by storing into the cache from where it can be directly accessed.
So the process can directly access data from the buffer cache because it is in the main memory already as well.

I/O Handling

- The basic model of I/O system is a sequence of bytes that can be accessed either randomly, or sequentially.
- There are no file formats (sequential, indexed etc.) and no control blocks (such as a file control block) in a typical user process.
- The I/O system is visible to a user process as a stream of bytes (I/O stream). A Unix process uses descriptors (small unsigned integers) to refer to I/O streams.

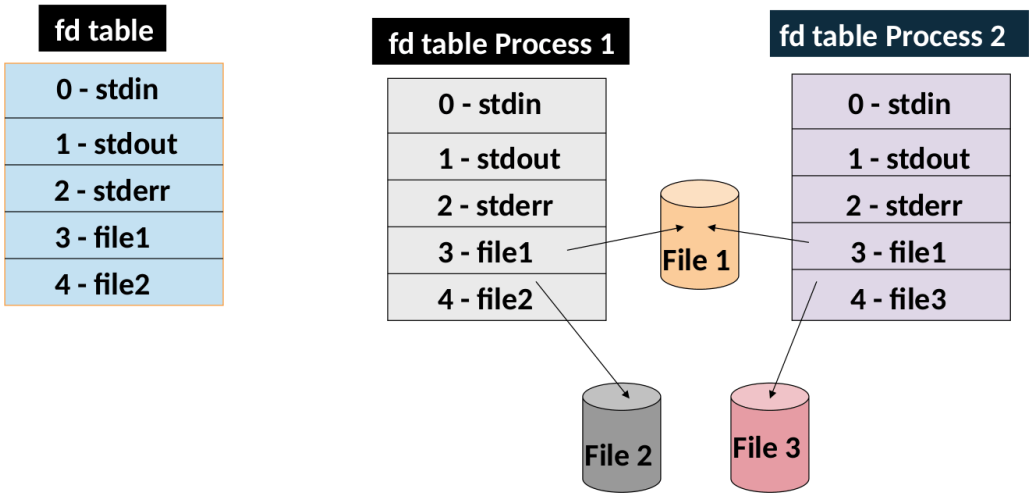
- The system calls related to the Input-Output operations take a descriptor as an argument.
- Each process has a separate File Descriptor (FD) Table.
- Valid file descriptor ranges from 0 to a maximum descriptor number that is configurable. (ulimit, /proc/sys/fs/file-max)
- Kernel assigns descriptors for standard input (0), standard output (1) and standard error (2) of the FD table as part of process creation.
- Kernel always assign minimum possible value from the fd table to any new file descriptors.

- The system calls related to the I/O system take a descriptor as an argument to handle a file.
- The descriptor is a positive integer number.
- If a file open is not successful, fd returns -1.

33

There are no control blocks other than the read() system call returning a zero when you have reached the end of a file. So all you can do is access the data that is present in the file.

fd Table



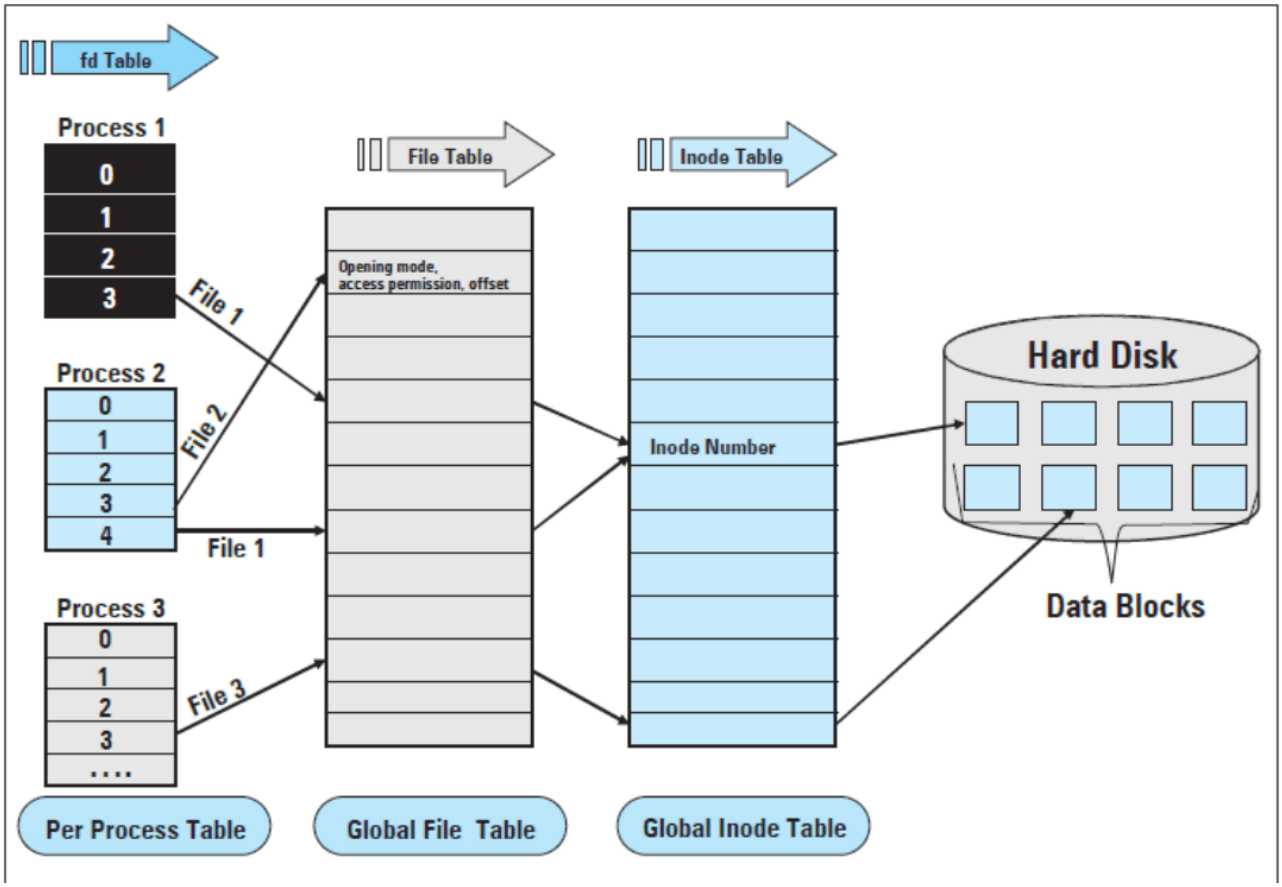
Each process has a separate fd table.

This slide shows how the fd table is built up for each process. In process 1, file 1 was opened first, followed by file 2 which is why they get fd as 3 and 4, respectively. However in process 2, file 1 was opened first followed by file 3 which is why they get fd as 3 and 4. This shows how two processes can have different fd tables even if they're accessing same/similar files or maybe both are the same processes just at different stages in their execution, etc.

File Table / Open File Table

This is a different thing from the fd table. It keeps track of all files that have been opened which is why it's called so. It contains certain metadata about the file opened as well as a pointer to its inode table entry.
An Open File Table entry is made per open() call made. So if a process called open() on the same file 10 times, then 10 new entries will be made in the Open File Table.
However the inode table will only have one entry per file.

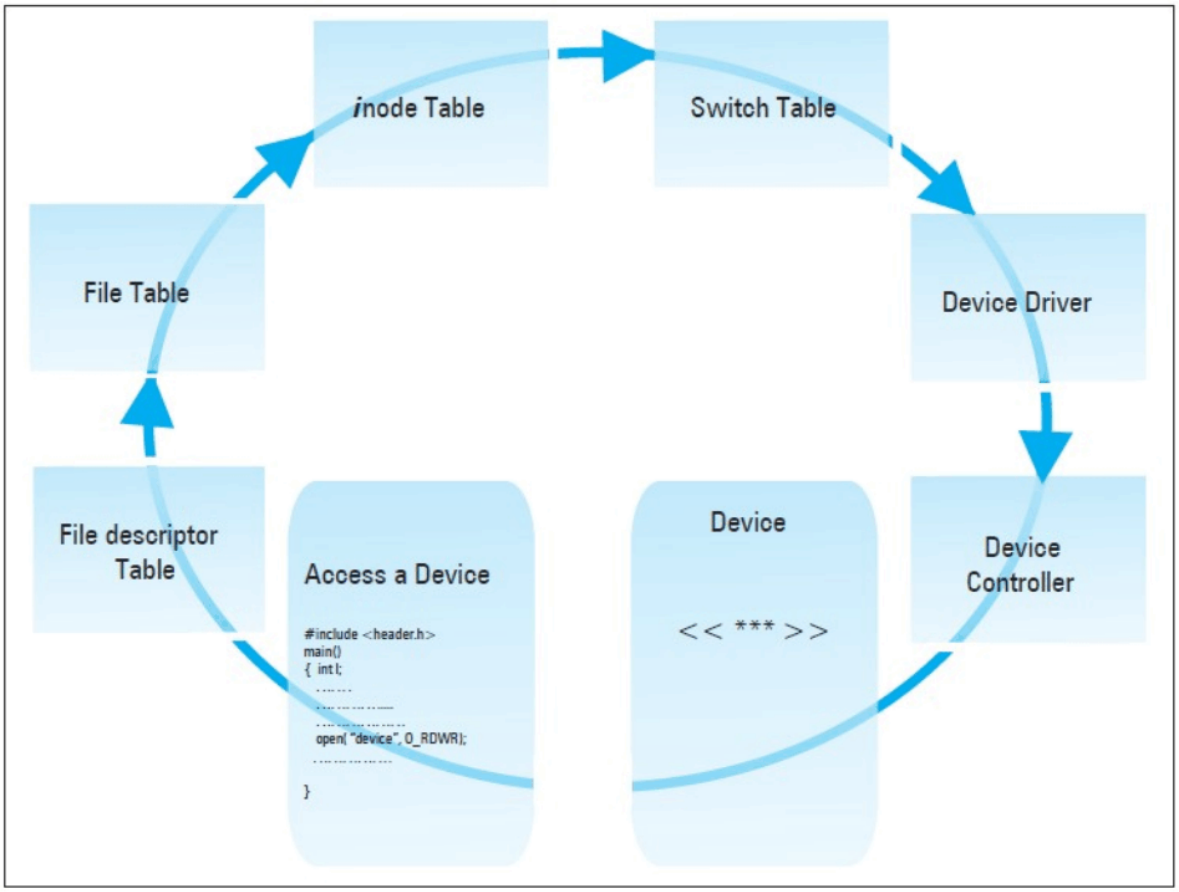
fd to Data Block



Switch Table

Used to identify device drivers for device special files. inode Table points to data block, but in case it is info about a device driver, then the inode entry points to the switch table entry which will in turn give you the major number for that device.

User Process Accessing Device Special File



System Calls

Use “man command_name 2” to get the man page for the system call associated with the command.

Read this to learn about file permissions (chmod command stuff) in Linux
https://www.guru99.com/file-permissions.html

Summary:
Three types of users, User, Group, Other Users (All)
Execute permission = 1, Write permission = 2, Read permission = 4
--rw-wx-wx = Regular File with given set of permissions for the given user types
d-rw-wx-wx = Directory with given set of permissions for the given user types

chmod 777 = All 3 permissions given to all types of users (Very dangerous)

07/09/21

System Calls

- File descriptor table (fd, process specific)
- File table (offset, mode,permission, pointer to inode table)
- Inode Table (inode number, pointer to Data Block).
- Switch table (only for device special files)
- Data Block (where a file is stored)

- Library Functions (Application Programs)
  - fopen, fwrite, fread, fclose
- System Calls (System Programs)
  - open, write, read, close
- Entry Points (Kernel Programs)
  - my_open, my_write, my_read, my_release

- The system call code is physically located in the kernel. The kernel itself is stored in a separate area of memory - which is normally not accessible to the process.
- Therefore, the first thing that is required to execute a system call is to change to Kernel Mode - so that the kernel memory can be accessed - this is what the "int 0x80" instruction in system call wrapper function (on Intel) does.

37

The library functions are a wrapper around the system calls, so in the end you are calling system calls.

open() System call

open(const char *pathname, int flags, mode_t mode)
Opens file specified by pathname
Returns fd of file created if succeeded else -1.

Flags determines which mode file is opened, O_RDONLY, O_WRONLY, O_RDWR
You can also specify additional options such as,
O_CREAT (create the file if it doesn't exist)
O_EXCL (to be used in conjunction with O_CREAT, if the file already exists, then don't create it and return -1).
If you specify O_CREAT then you also have to specify file permission mode with which to create the file.
For example, open("file.txt", O_RDWR|O_CREAT, 0744);
0744 = First 0 is to handle octal representation.

umask(): Default Permissions

If you do not specify permissions while creating a file, then default permissions will be set on that file and this is obtained with the umask command.

So just run umask on your terminal to get the default permissions that will be set on any file.
Note: Actual permissions are determined as (0777 - umask_value). So if umask = 0, then that means all permissions will be given and so on.

creat() System call

creat(const char* pathname, mode_t mode)
Mode gives the file permissions for the file that is created.
Other than that it is the system as open() with O_CREAT option.

Adding a new user to the Linux environment

Switch to root user
sudo su root

adduser newUser
You will have to give some info about the new user (if required) and a new user will be created. But this user will not be a “sudoer”, i.e. no permissions to execute commands that require root permissions.
So if you login as newUser and execute sudo apt-get update, then it will fail.

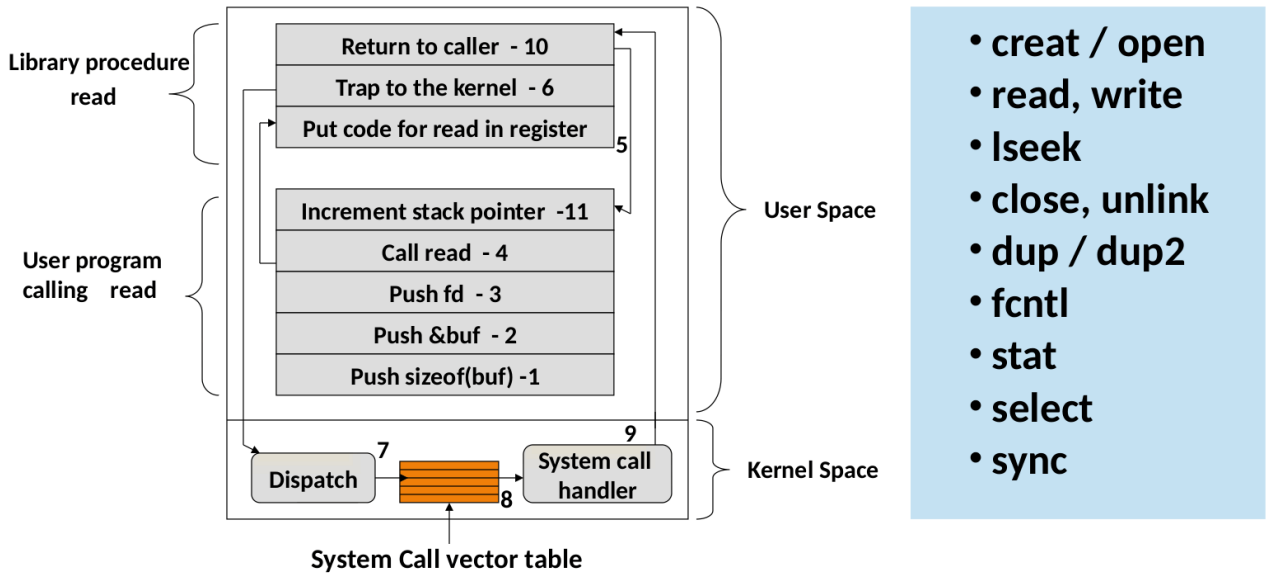
To add user as a “sudoer”,
usermod -aG sudo newUser

read() System Call

read(fd, &buff, sizeof(buff)))

buff = buffer into which the bytes that are read from the file given by fd is stored.
You can adjust the third parameter to determine how many bytes to read in at a time.

For example, you can read a file character by character as,
char ch;
int fd = open(“abc.txt”, O_RDONLY);
read(fd, &ch, 1);
This reads the first byte of file “fd” and stores it in the character variable “ch”.
read() returns -1 if failed otherwise it returns the number of bytes that have been read.



38

It's a bit difficult to follow, but this shows what happens in sequential order when the read() system call is made by a user program.

Trap to the kernel means that we are getting access to the kernel by sending the trap interrupt, so you can see how after sending the trap signal, we switch from user space to the kernel space for dispatching the read() system call and executing it.
System call vector table contains pointers to all the system calls.

To reiterate, System calls have to be executed in the kernel space.
The switching from user space to kernel space is handled by the kernel itself after the trap signal is sent for all standard system calls. In case we have our own system calls, we will have to write this logic.

lseek() System Call

Used for random access in a file.
Whenever we get/create a file pointer, it is pointing to the start of the file, so this helps us essentially move around in the file.
The library version of this function is fseek()

lseek(int fd, off_t offset, int whence)
It gives a pointer to the file given by fd where the first "offset" bytes have been skipped.

whence determines how the offset parameter is to be interpreted, the above statement is just the default interpretation of offset. Better check the man page for this to get more understanding.

EXTRA: Neat Trick for reading data from STDIN in C

scanf(“%s”, str);
If input is given as, “Hello World”
Then str = “Hello”
But what if we want str = “Hello World”?
The default delimiting character in C is “ “, so in this case we want it to be the newline character \n.
We can set this for each scanf call as,
scanf(“%[^\n]”, str);

lseek() System call Continued

Example use of lseek():
lseek(fd, 5, SEEK_SET)
SEEK_SET: File offset is to set to offset specified, in this case, 5
So this lseek() call sets fd to point to the 6th byte (character) in the file after skipping the first 5 bytes.

close() System call

int close(int fd);
Closes a file descriptor so that it no longer refers to any file and may be reused.
If fd is the last file descriptor referring to the underlying open file description, the resources associated with the open file description are freed, i.e. if fd is obtained after making an open() call and it is the last fd that points to that file, then the resources used for opening that file will be freed.
Returns 0 on success and -1 on error.

unlink() System call

unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

int unlink(const char *pathname);

dup() and dup2() System calls

They are used for duplicating a file descriptor. More on this later in the IPC section.

int dup(int oldfd);
The dup() system call creates a copy of the file descriptor oldfd, using the lowest-numbered unused file descriptor for the new descriptor.

After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags; for example, if the file offset is modified by using lseek(2) on one of the file descriptors, the offset is also changed for the other.

int dup2(int oldfd, int newfd);
The dup2() system call also does the same thing except it assigns the duplicate fd to be equal to the newfd parameter instead of doing it automatically like dup(). Note that if specified newfd is already being used by some other file, then that file will be closed; freeing up newfd to be used in the dup2() call.

To confirm that the duplicated fds (each having different fd value) are indeed pointing to the same file, you can check the fd table by doing “ls -l /proc/pid/fd”

fcntl() System call

int fcntl(int fd, int cmd, ... /* arg */);

fcntl() performs operations on the file described by fd and the operation performed is determined by cmd.

For example,
cmd = F_DUPFD duplicates the fd
F_GETFD returns the fd flags, F_SETFD sets the fd flags
F_GETFL returns the file status flags, this includes things like file access mode, file creation flags, etc.

And loads more which you can check in the man page.

stat() System call

int stat(const char *pathname, struct stat *statbuf);
Gives you statistics/metadata about the file specified by pathname.
This metadata is stored in the stat structure pointed to by “statbuf”. It contains various members such as the inode number (st_ino), file type and mode (st_mode), etc.
Check the man page for a full description of this.
Returns 0 on success else -1.

fstat() is the same as stat() except it uses the fd for identifying a file instead of the pathname.

perror(): FUNCTION FOR PRINTING ERRORS

There is a global integer variable called **errno** which is set by system calls and some library functions in the event of an error to indicate what went wrong.
perror() function is used to print the error that is described by what is currently present in errno
So, you can do something like,

```
if (open("file.txt", O_RDWR) == -1)
        perror("Error in opening file");
else
        // Do something with file
```

Suppose file.txt doesn't exist, then open() returns -1 and errno is set accordingly and STDOUT print will be,
"Error in opening file: File doesn't exist"
The stuff after the colon is added by the perror function.
So this is really handy in debugging errors.

select() System call

This is the man page description.
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *except_fds, struct timeval *timeout);
select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation (e.g., read(2) or a sufficiently small write(2)) without blocking.

Basically, this allows us to monitor a set of fds. fd sets can be of 3 types, read fds, write fds and exception fds. The timeout parameter (in seconds) specifies how long would this select() call monitor the fd sets for. If any event happens before the timer expires, then the select() call would return the number of file descriptors contained in the three returned descriptor sets (i.e. total number of bits that are set in read, write, except fd sets). If no event happens before the timer expires then select() would return 0 because 0 bits were set in the fd sets and if error happens then -1 would be returned.

Formal description of arguments from the man pages.
The arguments of select() are as follows:

readfds
The file descriptors in this set are watched to see if they are ready for reading. A file descriptor is ready for reading if a read operation will not block; in particular, a file descriptor is also ready on end-of-file.

After select() has returned, readfds will be cleared of all file descriptors except for those that are ready for reading.

Note on Non-Blocking Operations

Non-blocking operation means that it does not block the flow of execution for it to complete. In this context, a non-blocking read operation immediately returns what is available and does not wait for data to become available, so it is not guaranteed whether the read operation is actually able to read something. In the case of non-blocking writes, they achieve this behaviour by buffering the written data elsewhere in memory and unblocking the writing process immediately.

writefds
The file descriptors in this set are watched to see if they are ready for writing. A file descriptor is ready for writing if a write operation will not block. However, even if a file descriptor indicates as writable, a large write may still block.

After select() has returned, writefds will be cleared of all file descriptors except for those that are ready for writing.

exceptfds
The file descriptors in this set are watched for "exceptional conditions".

nfds
This argument should be set to the highest-numbered file descriptor in any of the three sets, plus 1. The indicated file descriptors in each set are checked, up to this limit.

Still do check the man page for this because this is quite complicated.

Couple of functions associated with fd sets.
FD_ZERO(*fd_set) -> Set all fds in this set to zero
FD_SET(fd, *fd_set) -> Set fd specified in this set to one.
FD_CLR(fd, *fd_set) -> Set fd specified in this set to zero.
FD_ISSET(fd, *fd_set) -> Check whether fd specified is set to 1 in this set.

Demo code to demonstrate select() monitoring STDIN for 5 seconds

```c
int main() {
      fd_set rfds; // Read fds set
      struct timeval tv;
      int retval;

      // Set all fds in read fd set to zero
      FD_ZERO(&rfds);
      // Set the fd determined by fd = 0 to 1, so we are monitoring STDIN
      FD_SET(0, &rfds);
```

```
        // Initializing timeval structure, if we initialize tv_sec,
        // then tv_usec can be ignored essentially
        // which is why it's set to zero here
        tv.tv_sec = 5;
        tv.tv_usec = 0;

        // nfds has to be set to 1 + highest fd value,
        // in our case highest fd value = 0, so nfds = 1
        retval = select(1, &rfds, NULL, NULL, &tv);

        if (retval == -1)
                perror("Error occurred");
        else if (retval)
                printf("Data is available for reading within 5 seconds");
        else
                printf("Data is not available for reading within 5 seconds");
}
```

To see this program in action, compile this code and run it. It will wait for 5 seconds, don't give anything on STDIN. And it should print the else statement printf() and if you do type something, it should print the else if printf() statement.

Similarly we can monitor other actual files for such changes as well.

Summary

int creat (char *file_name, mode_t mode)
int open(char *file_name, int flags);
int open (char *file_name, int flags,mode_t mode);
Ex: fd = open("temp", O_RDWR|O_CREAT, 0744);

dup or dup2 copies the oldfd into the newfd.
int new_fd = dup (old_fd);
int dup2 (int new_fd, int old_fd);
new_fd and old_fd shares: locks, file position and flags.

int write (int fd, const void *buf, int count); it writes count bytes to the file from the buf.
On success return with: Number of bytes written
0 - indicates nothing was written or -1 on error.

int read (int fd, void *buf, int count);

It reads count bytes from the file and store the data into the buf.

On success return with: Number of bytes read

0 - indicates end of the file or -1 on error.

int lseek (int fd, long int offset, int whence);

whence: SEEK_SET, SEEK_CUR or SEEK_END – from the end of file

- On success the system call returns with any one of the following value:
  - Offset value or 0 or -1

int stat (“file_name”, struct stat *);

int fstat (fd, struct stat *);

int lstat (“file_name”, struct stat *);

Select is a system call used to handle more than one file descriptor in an efficient manner.

- int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

- fd_set is the file descriptor set, which is an arrays of file descriptors.
- FD_CLR (int fd, fd_set *myset); FD_ISSET (int fd, fd_set *myset);
- FD_SET (int fd, fd_set *myset); FD_ZERO (fd_set *myset);

I/O subroutine

write modifications to the disk file - void sync (void);
Close a fd - int close (fd);
int unlink("file_name") - equivalent to $rm file_name;

Linux uses internal routines for accessing a file. For example

- **namei() (convert a "file_name" into an inode)**
- **iget() (reads an I-node)**
- **iput() (writes an I-node)**
- **bread() (read a block from buffer cache/disk)**
- **bwrite() (write a block from buffer cache to disk)**
- **getblk() (get a free block in the buffer cache)**

These are not that relevant for us because it's not that important for system programmers but you should still know about this at a higher level.
Use man pages whenever stuck.

File Locking

Why do we need file locking?

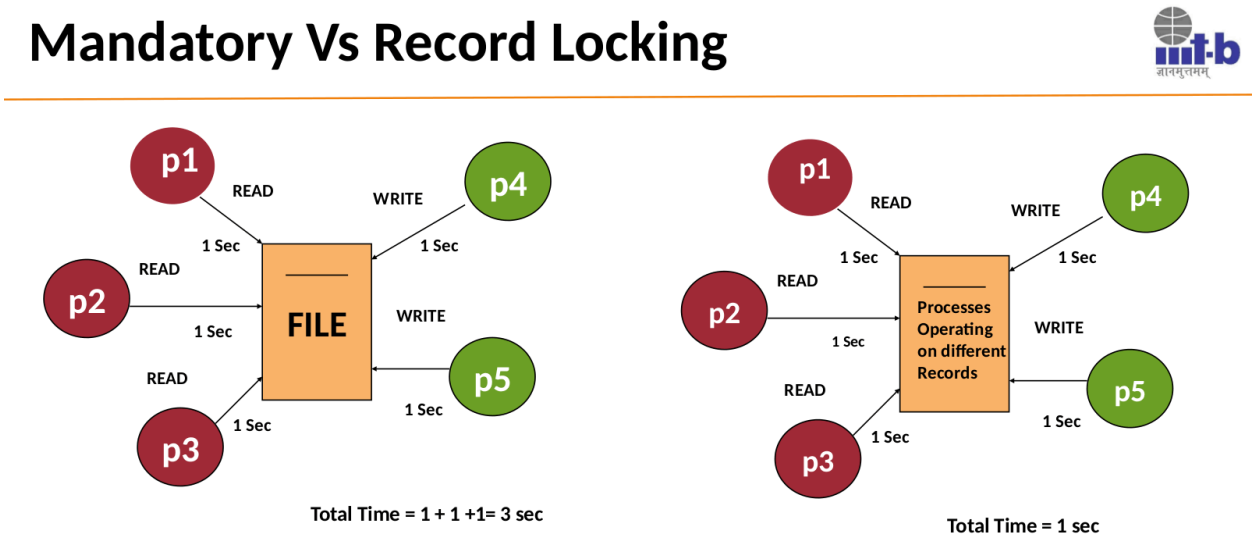
It often happens that we want to share data in a file to multiple processes.
So we have to provide the concept of **concurrent access** because it can happen that multiple processes will want to access the same file at the same time.
But this results in **Race Condition: Two or more processes trying to access the same file.** This can easily result in a deadlock. So we want to avoid this but at the same time **maintain synchronization**, i.e. all processes must have the updated version of the shared resource that they are accessing.
And this finally brings us to File Locking.

Types of Locking

- **Two Types**
  - **Mandatory locking**
    - **Lock an entire file**
  - **Advisory (or) Record locking**
    - **Lock to specific byte range**
    - **Granularity**
    - **Improve Performance**

Mandatory locking is enforced at the kernel whereas Advisory Locking is a convention that is followed by processes to ensure that proper locking protocol is achieved.
This basically means that in advisory locking, a process can lock a record but another process that doesn't use the locking concept will still be able to access the record.
However, in mandatory locking, once a file is locked by a process it is kept track of by the kernel and if any new process tries to access it then it will be blocked by the kernel until the lock is released.

Advisory locking is essentially like blocking only the critical sections of a file. This definitely improves performance over Mandatory locking. This can be illustrated as follows,



File Lock Structure

```
struct flock {
   short   l_type;
   /* lock type: read, write or unlock */
   short   l_whence;
   off_t        l_start;
   off_t        l_len;
   pid_t   l_pid;
};
```

Read lock: allows many readers but not a single writer

Write lock: allows only one writer but not a single reader

- **Lock or unlock is performed by fcntl function.**
- **int fcntl (int fd, int cmd, struct flock &);**
- **Command may be:**
  - **F_SETLKW**
  - **F_SETLK**
  - **F_GETLK**

F_SETLK acquires/releases a lock on the given file, if lock cannot be acquired/released it returns -1.
F_SETLKW is also for acquiring/releasing a lock on the given file, but if lock cannot be acquired/released, it waits till it can acquire/release lock, which is why it has that “W” for “Waiting”.
Because of this behaviour we typically use F_SETLK for unlocking and F_SETLKW for acquiring locks.
F_GETLK checks whether there is already a lock on the given file.
l_whence is the same as that with lseek(). The value of l_whence determines how l_start is interpreted. l_whence can take three values, beginning of file, from current position or from end of the file.

To unlock a file we just set flock.l_type = F_UNLCK and make fcntl() call with this new flock.

It can happen sometimes that a process **acquires a lock** on a file but forgets to unlock it (because of a bug) or the process **gets killed before it could release the lock**. In such a case, **the file would remain locked forever which is a big no no**. **To avoid this, we also specify the pid of the process, to ensure that whenever a process is killed/exits, all the locks held by that process are automatically released by the kernel.**

Read the man page of fcntl() for detailed info.

flock() Library Function

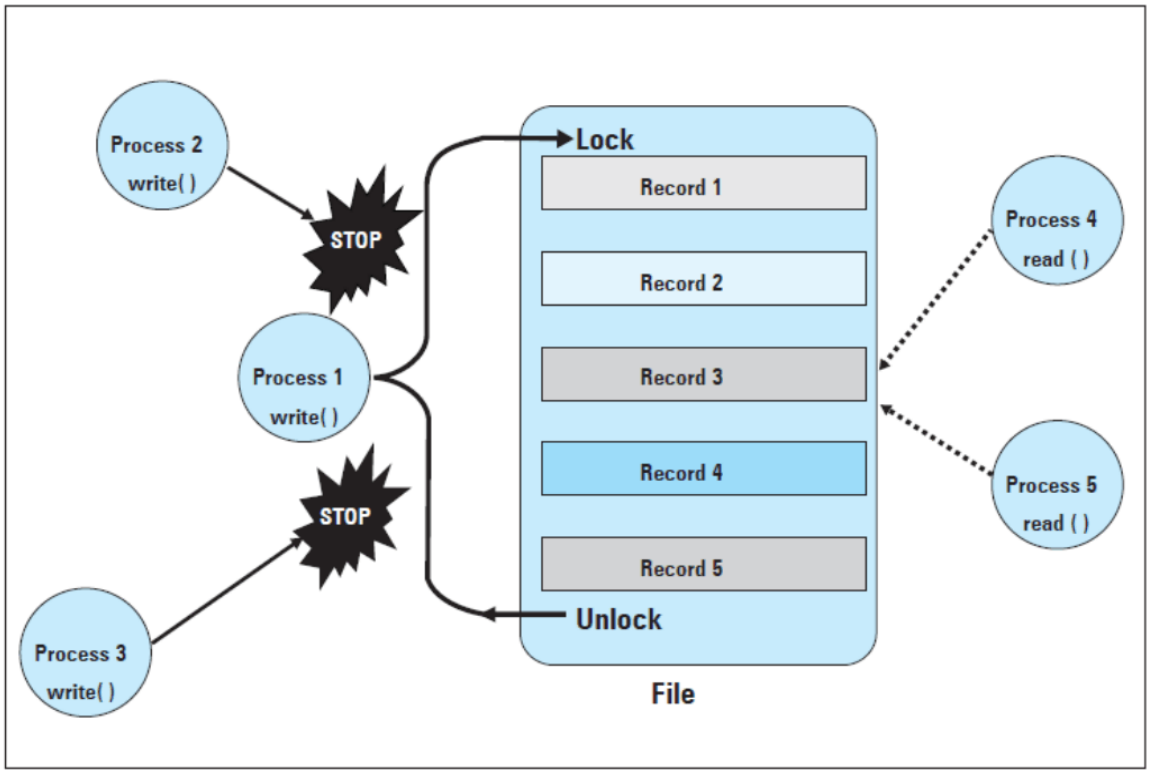
Note: We are going to be using fcntl for file locking because it is the actual system call and this is just a library function.

Apply or remove an advisory lock on an open file.

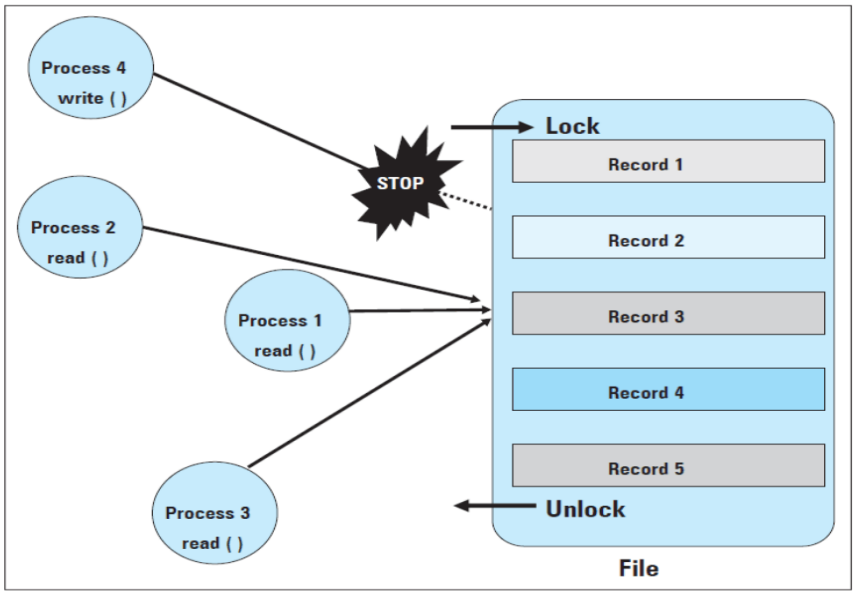
int flock(int fd, int operation);
Operation can take following values,
LOCK_SH = Shared lock
LOCK_EX = Exclusive lock
LOCK_UN = Remove an existing lock held by this process.
Returns 0 on success, else -1.

Summary

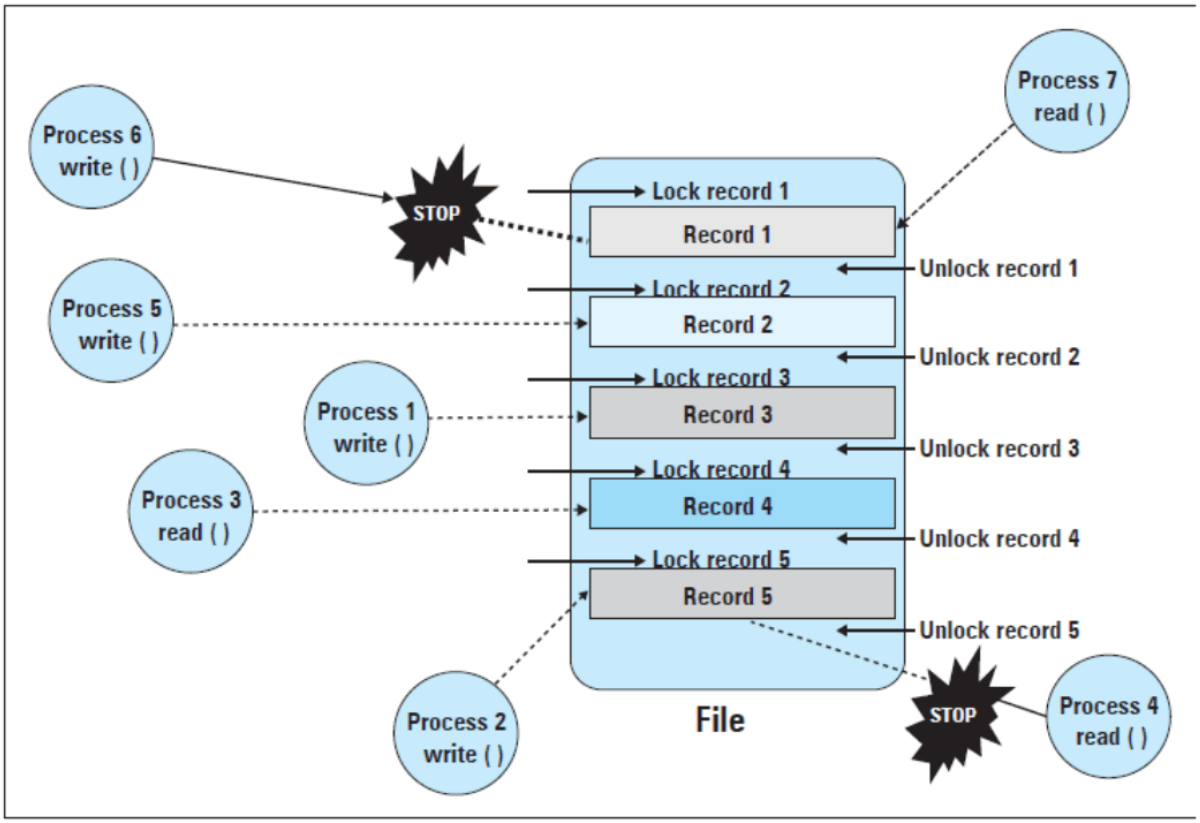
Non-Shared Lock - Mandatory File Locking



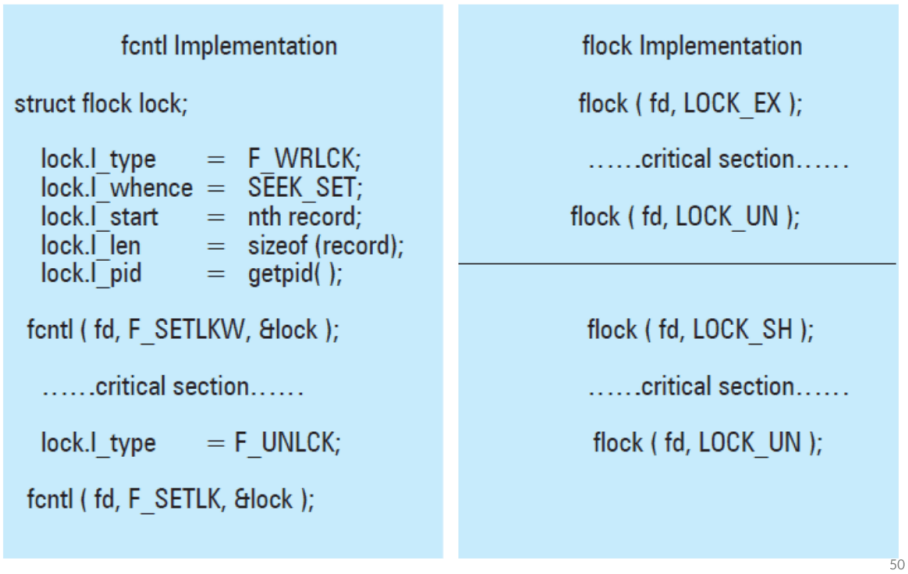
Shared Lock - Mandatory File Locking



Shared Lock - Advisory/Record File Locking



File Locking Implementation



50

How to implement Mandatory Locking using fcntl?
Set lock.l_start = 0 and lock.l_len = 0;
This tells the system that the entire file is to be locked, i.e. mandatory locking.

Demo program to add write lock on a file

```
int main() {
	struct flock lock;
	int fd;
	fd = open("file", O_RDWR);
	lock.l_type = F_WRLCK;
	lock.l_whenec = SEEK_SET;
	lock.l_start = 0;
	lock.l_len = 0;
	lock.l_pid = getpid();
	printf("Before entering into critical section\n");
	fcntl(fd, F_SETLKW, &lock);
	printf("Inside the critical section\n");
	printf("Press enter to unlock");
	// This makes the program wait till an input is received on STDIN and
enter is pressed, so file remains locked, till that happens
	getchar();
	printf("File unlocked\n");
	lock.l_type = F_UNLCK;
	fcntl(fd, F_SETLK, &lock);
	printf("End\n");
```

```
}
```

Now to see this locking in action, you can compile this program and execute it. Since no process has a lock on the file, the fcntl() call succeeds and the process enters into the critical section, so the program will now be stuck at the getchar() call. At the same time open a new terminal and execute the program in it. Now because the file is already locked by the first process, this fcntl() call fails, but because it is F_SETLK**W**, it will wait till we allow the first process to unlock the file, i.e. press any key to get out of getchar() call. So you can see how at one point of time, there can be only one write lock on a given file.

The same can be repeated for read lock as well with literally the same code, the only difference will be that now multiple processes can have read locks on the same file. So no process is dependent on the other process unlocking the file.

Hands-On List 1: Q17

Write a program to simulate online ticket reservation.
Implement write lock. Write a program to open a file, store a ticket number and exit. Write a separate program, to open the file, implement write lock, read the ticket number, increment the number and print the new ticket number then close the file.

Step 1: In this we have to make a database which consists of just a ticket number and which will be incremented by another file.

To create the database we can use a struct,

```
int main() {
  int fd;
  struct {
    int ticket_no;
  } db;

  db.ticket_no = 10;
  fd = open("db", O_CREAT|O_RDWR, 0744);
  write(fd, &db, sizeof(db);
  close(fd);
  fd = open("db", O_RDONLY);
  read(fd, &db, sizeof(db));
  printf("Ticket no: %d\n", db.ticket_no);
  close(fd);
}
```

This program creates a file called db in RDWR mode with 744 permissions and writes to it the

db structure which contains ticket_no = 10. It then opens the “db” file in RDONLY mode and reads the db structure from it. It then prints out the ticket_no associated with that db structure which has to be 10 in this case.
Remember to close each fd after its use.

Step 2: Incrementing Ticket Number
This is our critical section for which we will have to implement write locks on the created “db” file. And this is how to implement it,

```
int main() {
      // Add definition of db struct
      struct flock lock;
      int fd;
      fd = open("db", O_RDWR);
      // Initialize lock data members, we want mandatory write lock on db
file with SEEK_SET

      printf("Before entering into critical section");
      fcntl(fd, F_SETLKW, &lock);
      printf("Inside the critical section\n");

      // Read after acquiring lock, so that no inconsistencies happen
      read(fd, &db, sizeof(db));
      printf("Current ticket number: %d\n", db.ticket_no);
      db.ticket_no++; // Actually incrementing the ticket number

      // Because of initial read, fd is now pointing to the end of the
file, so we have to reposition to the beginning of the file
      lseek(fd, 0, SEEK_SET);
      write(fd, &db, sizeof(db));

      // This is just the demonstrate that locking actually works
      printf("Press enter to unlock\n");
      getchar();

      // Unlock the file
      lock.l_type = F_UNLCK;
      fcntl(fd, F_SETLK, &lock);
      printf("Exited critical section\n");
}
```

If we don’t want to redefine the db struct everytime then we can write it externally and add it as a header file in each file we want to use it for.

Hands-On List 1: Q18: Implement Record Locking

Write a program to perform Record locking.
a. Implement write lock
b. Implement read lock
Create three records in a file. Whenever you access a particular record, first lock it then modify/access to avoid race conditions.

In this case consider the example of storing train numbers and the ticket count for each train in a database. So now the structure will have two members, train_num and ticket_count.

One record refers to one instance of this structure, so to create three records, we create an array of this structure of size 3. This can be implemented as,

```
int main() {
	int i, fd;
	struct {
		int train_num;
		int ticket_count;
	} db[3];

	for (i=0; i<3; i+) {
		db[i].train_num = i+1;
		db[i].ticket_count = 0;
	}

	// Writing all 3 records to record.txt file
	fd = open("record.txt", O_RDWR);
	write(fd, db, sizeof(db);
}
```

Now we have to implement record level locking in another file,

```
// Definition of db struct but don't create array of db variables now

int main() {
	int fd, input;
	fd = open("record.txt", O_RDWR);
	printf("Select train number: 1, 2, 3\n");
	scanf("%d", &input);

	struct flock lock;
	lock.l_type = F_WRLCK;
```

```
      lock.l_whence = SEEK_SET

      // Record for train number 1 is stored at the beginning of the file
so at offset = 0
      // For train number 2, it is stored at offset = sizeof(db)
      // For train number 3, it is stored at offset = 2 * sizeof(db)
      // And that is essentially what this next line does
      lock.l_start = (input - 1) * sizeof(db);
      // This actually implements record-level locking
      // We start locking from the beginning of selected train's record
upto the end of that record which will be at l_start + sizeof(db)
      lock.l_len = sizeof(db);
      lock.l_pid = getpid();

      // Reading value of ticket number
      lseek(fd, (input - 1) * sizeof(db), SEEK_SET);
      read(fd, &db, sizeof(db));
      printf("Before entering critical section\n");

      fcntl(fd, F_SETLKW, &lock);
      printf("Current ticket count: %d", db.ticket_count);
      db.ticket_count++;

      // Currently fd is pointing to end of current record and we have to
move it to the start of current record, i.e. SEEK_CUR - sizeof(db)
// Or you can just do SEEK_SET + (input - 1) * sizeof(db)
      lseek(fd, -1 * sizeof(db), SEEK_CUR);
      write(fd, &db, sizeof(db));
      printf("To book ticket, press enter\n");
      getchar();

      // First getchar() call will be bypassed by train number input on
STDIN, so that's why we add another getchar() call here
      getchar();
      lock.l_type = F_UNLCK;
      fcntl(fd, F_SETLK, &lock);

      print("Ticket booked with number %d\n", db.ticket_count);
}
```

To see record locking in action, run this program and choose a train and wait at the critical section prompt. Run the program in another terminal and choose the same train and you can see that you cannot enter the critical section.

Now run the program again in another terminal but this time choose a different train and you can see that you can enter the critical section.

You can see how this concept of record locking is important and the key design principle in all of this is that **the critical section must be as small as possible.**

If two or more processes are waiting to access a lock on a file that is already locked, then once that process releases the lock the process that entered first would get the lock on the file. So **locking happens in FIFO order** because all lock requests are sent to a queue and they are processed accordingly.

14/09/21

Answering Some Questions

Extra reading: There are some special permissions attached to files in addition to read, write and execute like SUID, GUID and sticky bit. For more on these, read this article.
To summarize, SUID and GUID both force a file to be executed such that it looks like the owner of the file is executing the file instead of the user (in case of SUID) or group (in case of GUID) that is actually executing the file. Sticky bit restricts rename and deletion operations to be done by file owner and root only.
You can give such permissions as,
chmod o+t directory/
Give sticky bit permission “t” to other users “o”.

Handy command to use at times:
!<SearchPattern> - Executes the last command in history that matches this pattern
For example
!cc - Executes last executed cc command
Also remember that the **history command** will give you a list of previously executed commands.

How to check cpu info?
cat /proc/cpuinfo
Or use the utility command, lscpu

How to check CPU utilization?
mpstat command.
It gives you a lot of info about the processors which can be checked by going to it’s man page.
You can query the mpstat command as,
mpstat <time-interval> <number-of-times-to-get-readings-for>
Example, mpstat 1 5
This command gets mpstat output in an interval of 1 second 5 times, so the output table will

contain 5 entries describing the CPU utilization each one 1 second apart.

If you don't specify the number of times to get readings for then it will keep executing infinitely.
Cool way to see all of this in action,
Run mpstat 1, then in another terminal run an infinite loop C program and you can see the %usr usage jump quite significantly. It represents how much time is spent by the processor in the user space and that infinite loop code doesn't use any system call so no kernel space required -> fully user space execution.
If you add a system call in the infinite loop, then you can also see how the kernel and user space both get utilized (kernel space is represented as %sys).

mpstat gives data for all processors averaged. You can specify specific processors as,
mpstat -P 0,1 1
This gives results for processors 0 and 1 in 1 interval windows.

All of these commands and methods are important when it comes to identifying bottlenecks and analyzing the performance of your system (maybe for an application that you've developed).
You can check all of this in GUI format in the System Monitor application.

How to check which processes are taking up most resources?
top command

What does fg command do?
It brings the most recent background process to the foreground (fg).

How to check which process will be brought to the foreground?
Use **jobs** command to see the jobs in the stack, the last job will have a (+) mark indicating it is the first one to bring to fg and second last job will have a (-) mark, indicating it is the second one to bring to fg. So it all happens in LIFO order.

How to keep a process's /proc table entry without an infinite loop?
Now I don't know for sure how this works but…
Write a program with getchar() call. Then execute this program in background "./a.out &"
Then you'll have to press enter to get out of the getchar() call.
Now you will have a free terminal but the process is still alive (check "ps" output) and its /proc entry as well. So now you check the fd table and all the other data associated with the process.

Search man pages by keyword
man -k <your_keyword>
For example, man -k thread => Will return all man pages that contain "thread" in the title or description.
Very handy to find the right man page irrespective of which section it's in.

Process Management

- **Process is a program in execution.**
- **Processes carry out tasks in a system**
- **A process includes program counter (PC), CPU registers and process stacks, which contains temporary data.**
- **Linux is a multiprocessing system**
- **The Linux kernel is reentrant**

- **A process uses many resources like memory space, CPU, files, etc., during its lifetime.**
- **Kernel should keep track of the processes and the usage of system resources.**
- **Kernel should distributes resources among processes fairly.**
- **Most important resource is CPU. In a multiprocessing environment, to attain an ideal performance of a system, the CPU utilization should be maximum.**

Reentrant function means that that function can be called by different processes many times. So when we say that the Linux kernel is reentrant, it means that different processes can access the Linux kernel at the same time. This is possible because Linux is a multiprocessing system (multiple cores/CPUs in the same PC) and so multiple processes can be in execution at the same time accessing the Linux kernel.

CPU Utilization must be maximized means that CPU Waiting Time must be minimized.

Context Switch

- **In order to run Unix, the computer hardware must provide two modes of execution**
  - **kernel mode**
  - **user mode**
- **Some computers have more than two execution modes**
  - **eg: Intel processor. It has four modes of execution.**
- **Each process has virtual address space , references to virtual memory are translated to physical memory locations using set of address translation maps.**

- **Execution control is changing from one process to another.**
- **When a current process either completes its execution or is waiting for a certain event to occur, the kernel saves the process context and removes the process from the running state.**
- **Kernel loads next runnable process's registers with pointers for execution.**
- **Kernel space: a fixed part of virtual address space of each process . It maps the kernel text and data structures.**

- Context switching refers to switching the process in execution from one process to another.
- For this the state of the process in execution must be saved to memory which is the **Process Control Block (PCB).**
- This structure is a doubly linked list which is defined as a struct called task_struct in the usr/include/sched.h header file (You must download the Linux source code first to see it). The entire thing takes up around 800 lines in code, so you can imagine just how big this struct is and the number of data members it contains.
- Every process has a virtual address space and a fixed starting part of this is called the kernel space. This virtual address space points to the physical kernel data.

Execution Context

- **Kernel functions may execute either in process context or in system context**
- **User code runs in user mode and process context, and can access only the process space**
- **System calls and signals are handled in kernel mode but in process context, and may access process and system space**
- **Interrupts and system wide tasks are handled in kernel mode and system context, and must only access system space**

- **Every process is represented by a task_struct data structure.**
- **This structure is quite large and complex.**
- **When ever a new process is created a new task_struct structure is created by the kernel and the complete process information is maintained by the structure.**
- **When a process is terminated, the corresponding structure is removed.**
- **Uses doubly linked list data structure.**

54

Process Context = Related to the process in execution
System Context = Related to the system in general

- System calls and signals are initiated by processes/programs which is why they are executed in process context. But these calls and signals are stored in the kernel space, so that's why they are executed in kernel mode and access system space.
- Interrupts and system-wide tasks are not related to any particular process which is why they are system context and can only access system space.
- An example of a system-wide task is the periodic adjustment of process priority in scheduling algorithms (will know in more detail in later sections).

Task_struct Structure - /usr/src/linux-4.12/include/linux/sched.h

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;

        int on_rq;

        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;

        unsigned int policy;
        int nr_cpus_allowed;
        cpumask_t cpus_allowed;

        struct sched_info sched_info;
        struct list_head tasks;

        struct mm_struct *mm, *active_mm;
        ----------------
        ----------------
        ----------------
```

55

Process States

The different process states along with their symbols are,
[Check man top and find zombie keyword in it to get more details]

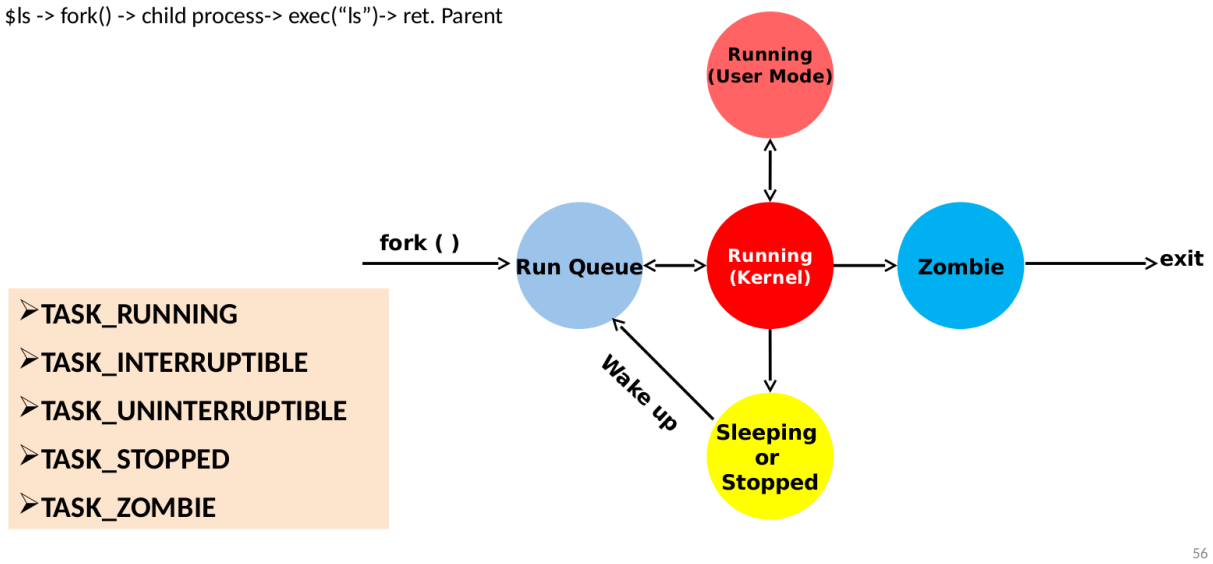
- Running = R
- Sleeping
    - Uninterruptible Sleep = D
    - Interruptible Sleep = S
- Stopped
    - Stopped by Job Control Signal (SIGSTOP signal => Ctrl + Z) = T
    - Stopped by Debugger during Trace = t
- Zombie = Z

NOTE: Stopped means Suspended and is not the same as Terminate!
So a stopped process can resume its execution whereas a Terminated process cannot, which is why all its entries are removed from /proc and other related locations.

Demo to see this in action,
Run a C program (infinite loop), then do “cat /proc/pid/status | head” to check the process state. While it's running, the state will be R and then press Ctrl + Z (sends the SIGSTOP signal) and the process state will be T.

Call the “sleep(time_in_seconds)” function in the C program and you can see the Interruptible Sleep state (S).

You cannot put a process into uninterruptible sleep at the user level. You can only do that at the kernel level.



Depending on the instructions to be executed, the process can be in kernel or user mode.

$ls -> fork() -> child process -> exec("ls") -> ret Parent
What does this mean?
When you execute the "ls" command from a terminal/shell. The shell executes fork() to create a child process that will execute the "ls" command with pwd (present working directory) as the argument to "ls" which is located at "/bin/ls" and then it returns the result to its parent which is the shell from which the command was executed.

Zombie State (IMPORTANT INTERVIEW QUESTION)

It is not the state where the process' parent gets killed, that is called an orphan process and not a zombie process.

A process that has finished execution but it's entry is still present in the process table.

Why does this happen?
When a process terminates it returns exit code to its parent and then the parent reaps off the child process entry from the process table. So in the time duration where the process has terminated and the parent hasn't picked up the exit code. Because of this, the child process still appears to be alive for the parent and which is why its entry is still present in the process table even though the actual process has finished execution. So it is alive despite being dead, i.e. zombie.

A process can remain in zombie state for a longer duration of time because its parent was busy doing some other operation while the child returned exit code and so the child terminated but parent hasn't read it yet and so it doesn't clean up the child's entry in the process table.

Can you kill a Zombie process?
NO! Because it is already dead
You can only try to kill the parent process or you have to wait until the parent picks up the child's exit signal which allows it to remove the child's process table entry.
If the parent process is killed, then the child becomes an orphan and is inherited by the systemd/init process (the first process in Linux system with pid 1) which will wait on the child process and eliminate it from the process table.

If you have an application that is creating zombie processes then you will have to reboot the system which is why you should be really careful to avoid such situations.

fork() System call

Used for creating a new process by duplicating the calling process. The new process is referred to as the child process and the calling process is referred to as the parent process.

Fork system call is used for creating a new process, which is called the child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same PC (program counter), same CPU registers, same open files which are used in the parent process.

It takes no parameters and returns an integer value.
Below are different values returned by fork():

- Negative Value: creation of a child process was unsuccessful.
- Zero: Returned to the newly created child process.
- Positive value: Returned to parent or caller. The value contains the pid of the newly created child process.

Check "man 2 fork" to get more details on how parent and child processes differ from each other.

```
int main() {
      printf("Fork returns: %d\n", fork());
      wait(0);
}
```

Output:
Fork returns: 9245
Fork returns: 0
[Explanation below]

wait() System call

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent ***continues*** its execution after the wait system call instruction.
Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.

It returns the pid of the child process that terminated. If multiple processes terminate simultaneously then any one of their pid's will be returned. **If a process has no child processes then wait() immediately returns -1.**

What does the above program print?
Fork returns: 9245
Fork returns: 0

Why two prints?
Because fork() duplicates the parent process.
So first it executes fork() to create a child process and returns the pid of the child process which gets printed as 9245 (or any other integer > 0).
Now both parent and child processes will concurrently execute the next instruction, in this case, it's the printf() call.
So firstly, printf() for parent executes which will print pid of child process and then printf() for child process executes which will print 0 because return value of fork() for newly created child process is 0.

How to identify whether a process is child or parent?

```
if (!fork())
      // Child Process
else
      // Process Process
```

fork() will return 0 for a child process, so it will satisfy the if condition and go into the if block.
If it's a parent, then it will fail if condition and go into the else block.

Creating an Orphan Process

getppid() = Returns parent's process id

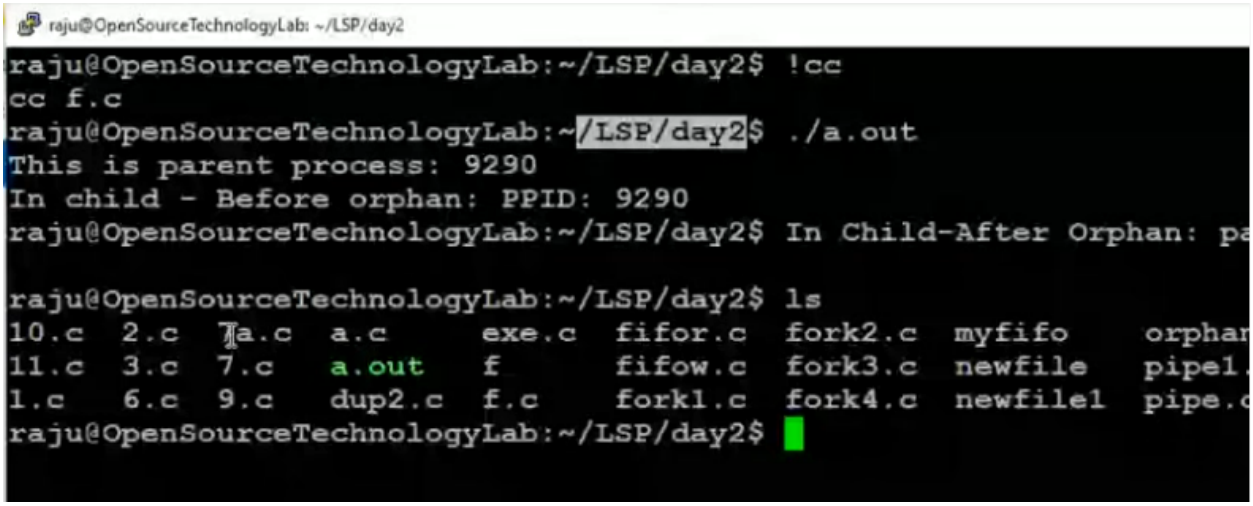
```
int main() {
      if (!fork()) {
            printf("In child - Before orphan PPID = %d\n", getppid());
```

```
            sleep(1);
            printf("In child - After orphan PPID = %d\n", getppid());
      }
      else {
            printf("This is parent process %d\n", getpid());
            exit(0);
      }
}
```

What's happening here?
Parent process calls fork() and then both parent and child process execute the if condition, fails for the parent process and succeeds for the child process.
Then both parent and child processes make their first printf() statements.
Then the child process sleeps for 1 second, during this, the parent process will execute exit(0) meaning that it has finished execution and terminated and now finally the child process has become an orphan and so **it will be inherited by the systemd process**. This will be reflected in the second printf() statement for the child process after it wakes up from its sleep(), getppid() will return 1 because pid of systemd is 1.

Interesting behaviour to be observed from this program execution,
When the parent terminates because of exit(0), it returns the result to its parent process which is the shell from where ./a.out was run. This means that we return to the shell prompt when the parent process exits and it is after this that the child process printf() is run and it too terminates. So after the child process terminates we do not get the shell prompt.

This is basically what I'm trying to explain,



We get the shell prompt and then "In Child-After Orphan: …" printf() occurs and after that we do not get a shell prompt until we press enter.

17/09/21

Creating a Zombie Process

```
int main() {
	if (!fork()) {
		printf("In child - PID = %d\n", getpid());
	}
	else {
		printf("This is parent process %d\n", getpid());
		sleep(30);
		wait(0);
	}
}
```

The only changes here are that the parent process sleeps for 30 seconds after which it waits for the child to terminate and the child process just prints pid and exits immediately.
So what is going to happen is, while the parent is sleeping, the child exits but its exit hasn't been received by the parent, so its entry remains in the process table and so the child is now a zombie process.

To check this, run this program and you'll see child process pid, then check "cat /proc/child_pid/status | head" to get process state which will be Z or you can do "top -child_pid"

You can also try killing the zombie process and you'll see that it doesn't work.

Do note that orphan processes are just a group of processes like how we have daemon processes, foreground processes, etc. It is not exactly a bad thing, it's just a natural outcome that happens with execution of the system. But zombies are a process state and it is indeed a bad thing if a process remains in a zombie state for a long time, so this must be avoided. A process becoming a zombie is not a bad thing, it too is just a natural outcome of execution but remaining a zombie for a long time is a bad thing.

fork() Practice Questions

How many "Hello World"s does this program print?

```
int main() {
	fork();
	fork();
	fork();
```

```
	printf("Hello World\n");
	wait(0);
}
```

For this you need to count how many processes are created in total.
1 is the parent process, then how many children are created?
You can draw a process tree structure which is like “pstree” output to get a better understanding.
For this program it will look like,
p
| -> c1 | -> c11 | ->c111
| | -> c12 |
|
| -> c2 | -> c21 |
|
| -> c3
So in total 7 children are created + 1 parent = 8 “Hello World”s

A handy formula for this is, for n consecutive fork() calls,
2^n - 1 children are created
So here, 2^3 - 1 = 7 children are created.

How many “Hello World”s does this program print?

```
int main() {
	printf("Hello World\n");
	fork();
	fork();
	fork();
}
```

Only 1!
7 children will be created like before, but the child that has been forked will continue execution from next instruction onwards. So none of the child processes will run the printf() statement.

Interview Question: How many “Hello World”s does this program print?

```
int main() {
	printf("Hello World");
	fork();
	fork();
	fork();
	wait(0);
}
```

And ofcourse, the answer is not 1. And it's not zero either if you want to make wild guesses. The real answer is 8.
But why?
First thing to note is that it's not because of wait(0), you can remove it and you'll still get 8 "Hello World"s. The reason for this is the \n that was present in the previous program and the fact that there's no \n in this program.

Significance of \n in printf statements

When printf() encounters \n, the string accumulated in the **keyboard buffer for that process** will be sent to the monitor. If no \n was encountered, then the string accumulated in the keyboard buffer for the process will be sent to the monitor only when **the program exits.**

Now what happens in the above question is that the parent executes printf(), now "Hello World" is in the keyboard buffer for the parent process. Then the fork()'s are called, creating 7 children in total. Note that each child is a duplicate of the parent process, they use different memory spaces but the initial value of all the data in the child will be the same as that for the parent. When any child exits, the stuff in the buffer (same as that of parent) is sent to the monitor, this results in each process printing "Hello World" to the screen. There are 8 processes in total including the parent and so, there are 8 "Hello World"s.

You can try out different combinations of this question and see how the output changes to clear the concepts.

Process Scheduling

Every process in the system has a process identifier.

- **The process identifier is not an index into the task vector, it is simply a number.**
- **Each process also has User and Group identifiers, these are used to control the process access to the files and devices in the system**

- **The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime.**
- **This clock is the combination of software and hardware setup.**
- **It is independent of CPU frequency.**
- **A clock tick unit is Jiffy. System's interactive response depends on the clock frequency.**
- **For example: the jiffy value may be 10ms (100Hz) or 1ms (1000Hz) depending on implementation**

Note: Clock tick unit / jiffy is different from the CPU frequency.
An example of the jiffy value implementation is that, if it's set to 10ms and you give a sleep(11 ms) call, then the process will actually sleep for 20ms. However if jiffy value was 1 ms, then it

would have slept for exactly 11ms. So depending on how time-critical the system needs to be, the kernel can be adjusted to keep track of time more accurately.

- **Each clock tick, the kernel updates the amount of time that the current process has spent in system and in user mode.**
- **Linux also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire. These timers can be single-shot or periodic timers.**

- **The job of a scheduler is to select the most deserving process to run out of all of the runnable processes in the run queue.**
- **Implement fair scheduling to avoid starvation**
- **Implement suitable scheduling policy**
- **Updates state of the processes in every clock tick (jiffy)**

Throughput of the system is the most important criteria to be satisfied for any scheduling mechanism. Things change a bit for real-time systems because meeting deadlines is more important there and deterministic behaviour is required.

Linux uses priority-based scheduling.
Note: Lower Priority Value = More Priority given

Priority Ranges

0 to 99: For Real-Time processes -> Static Priority
100 to 139: For Non-Real Time processes, i.e. General processes -> Dynamic Priority
This is mapped to -20 to 19.
Note: +19 for general processes is different from +19 for real-time processes and so on.

For Real-Time processes,

If the **process will take more time** to execute **then Round Robin scheduling (SCHED_RR)** is used for it.
If the **process doesn't take much time** to execute **then FIFO scheduling (SCHED_FIFO)** is used for it because Round Robin will introduce unnecessary context switches on a task that could have been quickly completed.

How do we determine how long a process will take to execute?
We use heuristics based on the process to estimate how long it will take. This estimated execution time has to be independent of the load that the system is under at that point.

So in the case of real-time processes we will have to handle these aspects, what scheduling to use, what priority to assign, etc. because the system doesn't know about the criticality of your process.

Also, **you can only create real-time processes if you have sudo or root privileges.**

For General processes,

The scheduling algorithm and priority, etc. does not have to be handled by us, it will be automatically done by the system, so its scheduling algorithm is denoted as **SCHED_OTHER.** But under the hood this is a **priority-based round robin scheduling.**

All Real-Time processes are at higher priority than general processes.

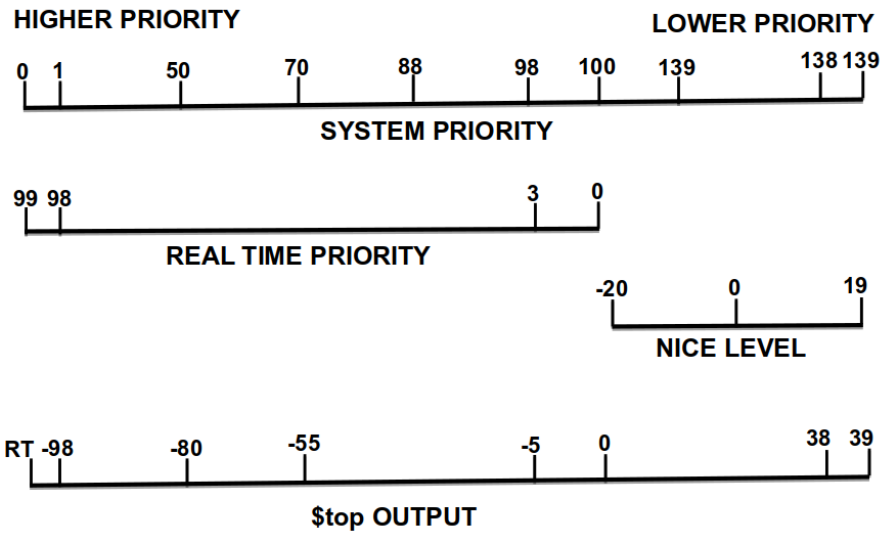
Implementation details on general processes priority,

It's range is from -20 to +19. -20 is the highest priority and +19 is the lowest priority.

General processes start with priority 0 but with **each clock tick spent in execution their priority is decremented,** this is called **rescheduling process priority** periodically.
Every clock tick a process waits for CPU time, its priority is incremented.
This is done to avoid starvation because lower priority processes can end up waiting forever.

The higher a process's priority, the larger will be the time slice (time quantum) allotted to that process. So round robin is being implemented but priority is also factored in.
The minimum time slice is 10ms (priority = 19), default is 150ms (priority = 0) and maximum is 300ms (priority = -20).

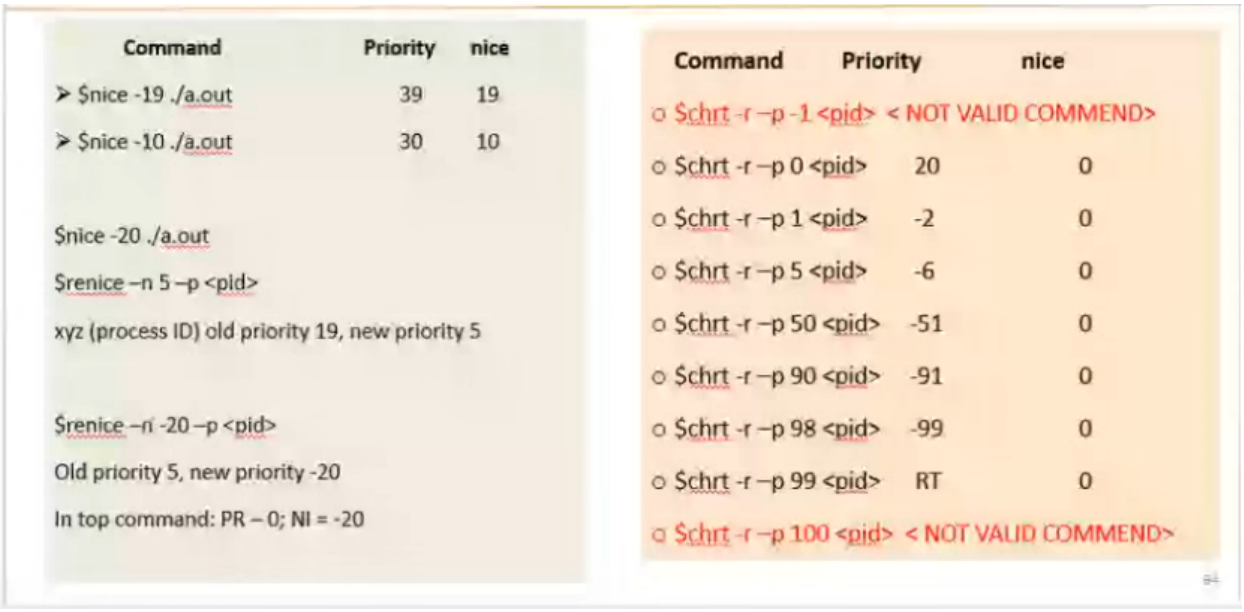
Understanding process priority in Linux



This priority concept is absolutely retarded and confusing to understand but note that priority levels -20 to 19 were already being used for general processes and then the real-time process support was added later to the Linux kernel, so the additional priorities had to be supported now without disturbing the original -20 to 19 priority levels.

That's why "top" command displays priority so weirdly, all the negative priorities are for real-time processes and priority = rt means real-time priority = 99, priority = -98 means real-time priority = 98 and so on and all the positive probabilities are for general processes which will be translated to the nice level of -20 to 19 in the column that's next to the priority in the "top" command output. Execute "top" and see this in action.

nice and chrt - Manipulating Scheduling Priorities



nice can be used to execute a process with modified scheduling priority instead of the default (0).
nice -<increment_value> process_to_execute
Increment value is added to the default priority, i.e. 0.
Example: nice -19 ./a.out -> Execute a.out with priority (in nice-level) 19 (highest).

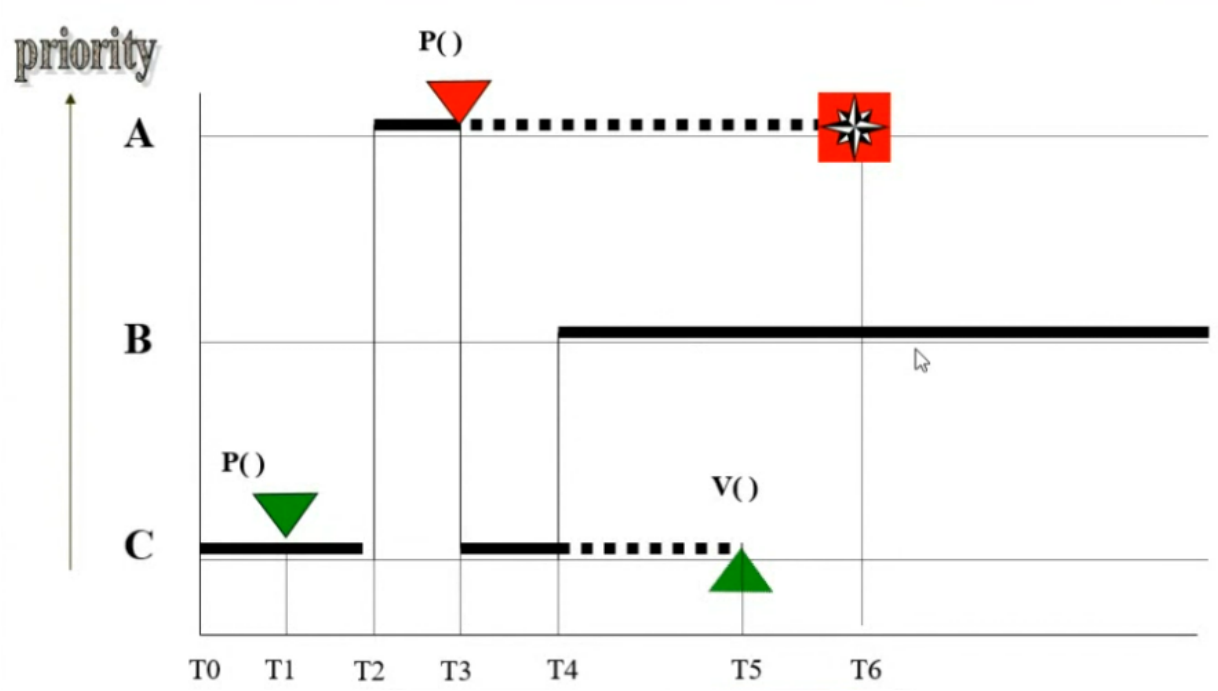
renice
It can be used to change the priority of a running process.
Syntax: renice -new_priority -p pid
renice -n <new_priority> -p <pid>, new_priority is in terms of nice level.
Example: renice -n 5 -p 69 -> Change priority of process with pid 69 to 5.

chrt
To modify the attributes of a real-time process including its priority.
Example: chrt -r -p 99 6969
This changes priority of process with pid 6969 to 99, i.e. RT in top output.
The -r option is used to set scheduling policy to Round Robin, you can also use -f to set it to FIFO.

21/09/21

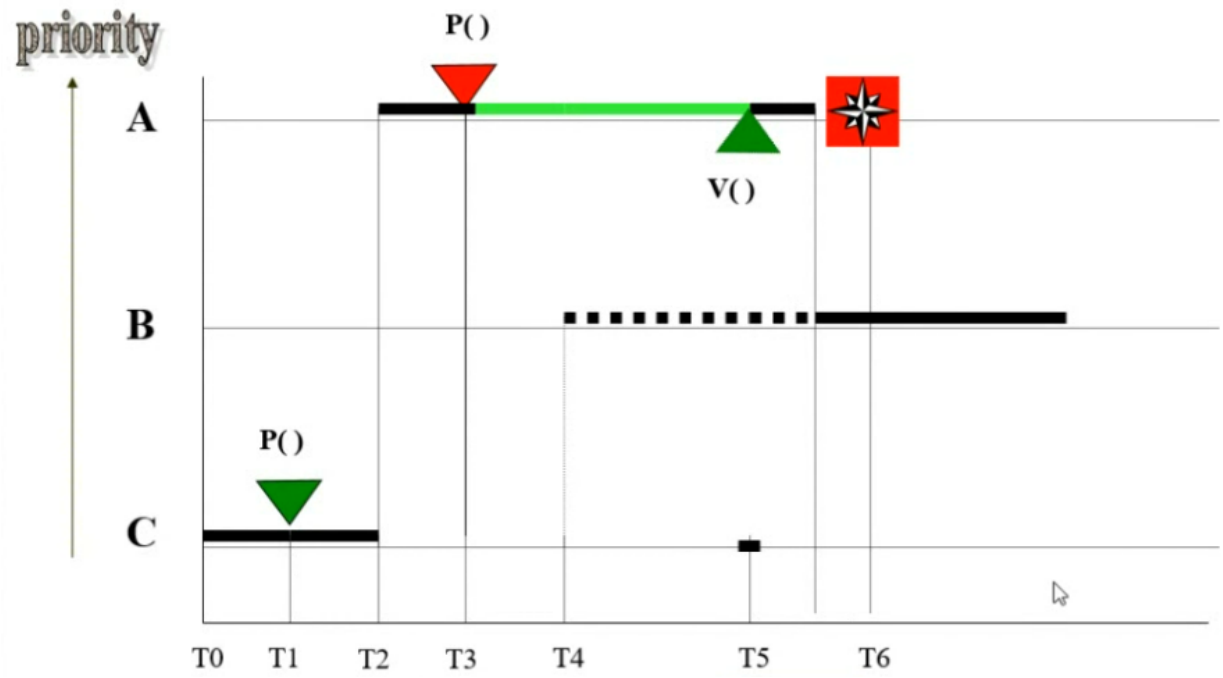
Unbounded Priority Inversion

This is a problem that happens in preemptive scheduling algorithms and is very important for Real-time scheduling.
Consider the following example,



Process C with low priority is currently executing in the critical section and a higher priority process A comes requesting for CS, but it will not be able to get into it and enter sleep state. Now a process with medium priority B comes in but it doesn't require access to C.S. and so it will preempt C and start execution. So the C.S. is still held by C which is currently sleeping because B is executing and A is also sleeping waiting to get access to C.S. So if A had a deadline (indicated by star mark in A's graph), then that will be missed.
This is essentially the priority inversion problem.

Priority Inheritance -> Solution to Priority Inversion

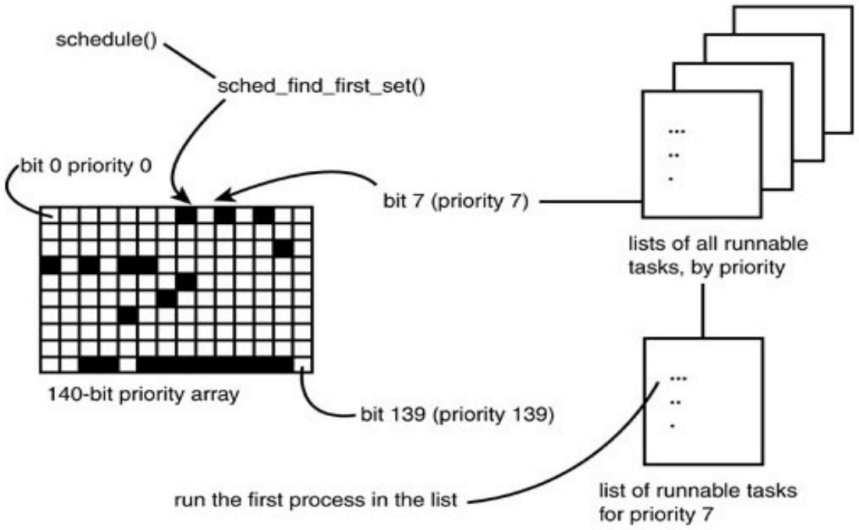


Same scenario as before.
When A arrives with higher priority, the priority of C is temporarily increased to the same as that of A and A enters sleep because it still wants C.S. that is held by C. Now when process B comes in, it will not be able to preempt C because it is at lower priority and so B enters sleep. Once C gets out of C.S., its priority is lowered to its original level and it enters sleep and now A can acquire a lock on C.S. and start execution in time for the deadline. Once A has finished, B can start execution and only when B has finished execution does C wake up from sleep.

To summarize, the original problem is that a process goes to sleep while holding C.S (C) because of which, a higher priority process that wants access to C.S. (A) cannot preempt it but a process that doesn't want access to C.S. but has higher priority (B) can preempt it leading to missed deadlines. The solution is that whenever a process like A comes in, we increment priority of process like C to be equal to that of A and decrement it back to original level once C has released the C.S.

O(1) Scheduler

Normally we assume that the waiting time for a process to get CPU time is proportional to the number of processes in the wait queue and so on. This is true in the case that the scheduler is O(n), but an O(1) scheduler guarantees that a process will get CPU time according to its priority without any other overhead like number of processes in the wait queue and so on.



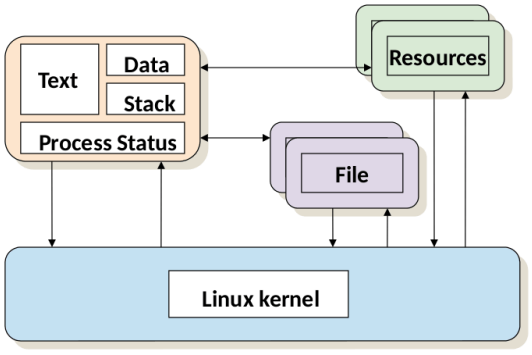
Ref: Linux Kernel Development by Robert Love

There are a total of 140 priority levels (RT + Non-RT), so we have a 140 bit array to indicate each priority level and each bit points to that priority level's wait queue.
The wait queue for each priority level indicates the processes that are waiting at that priority level.

So when the kernel wants to pick the next process for allocating CPU time, it will look at the priority array and find the highest bit that is set (i.e. there exists tasks at that priority level waiting for CPU time) and dequeue the first process from that wait queue for execution.
This is how the scheduler works in O(1) time.

Process Creation

- **A Command is normally executed in a shell**
- **When you enter a command, the shell searches the corresponding command's executable image (use PATH environment variable) and loads the image then executes.**
- **For execution, the shell uses fork and creates a new child process and the child process's image is replaced with the command's executable image.**
- **After completion of execution of the child process it gives the exit status to the parent process, i.e, shell.**

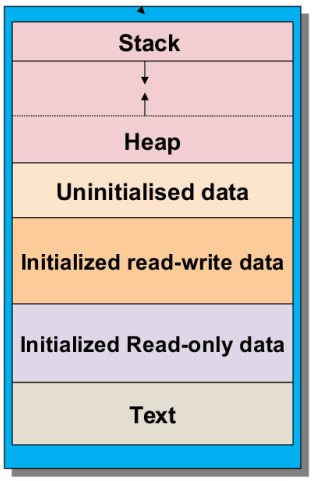


This is the same stuff that was explained for the "ls" command in the Process States section.

The diagram on the right shows the image of a process and how it consists of the text (which also includes process code), data, stack and process status. The diagram also shows how a process interacts with other components like resources, files and the Linux kernel.

When exec() functions are used, the old process image is replaced with the new one, so this is what's happening, the new text, data, stack and process status are loaded after removing the old ones. And this is why any code in the original process after exec() call doesn't execute because the original process itself has been fully replaced by the process called in exec() call.

Process Image



- The *text* portion of a process contains the actual machine instructions that are executed by the hardware.
- When a program is executed by the OS, the text portion is read into memory from its disk file, unless the OS supports shared text and a copy of program is already being executed.

- The data portion contains the program's data. It is possible for this to be divided into 3 pieces
- Initialized read only data contains elements that are initialized by the program and are read only while the process is executing.
- Initialized read write data contains data elements that are initialized by the program and may have their values modified during execution of the process.
- *Un-initialized data* contains data elements that are not initialized by the program but are set to zero before execution starts .

```
$ size a.out  (man size )
  text   data    bss    dec    hex
filename
   920    268     24   1212   4bc
```

72

- The *heap* is used while a process is running to allocate more data space dynamically to the process.
- The *stack* is used dynamically while the process is running to contain the stack frames that are used by many programming languages.

- The stack frames contain the return address linkage for each function call and also the data elements required by a function.
- A gap is shown between heap and stack to indicate that many OS leave some room between these 2 portions, so that both can grow dynamically.
- The *kernel context* of a process is maintained and accessible only to the kernel. This area contains info that the kernel needs to keep track of the process and to stop and restart the process while other processes are allowed to execute.

73

Copy-on-Write wrt fork()

- **Instead of copying the address space of the parent, Linux uses the COW technique for economical use of the memory page.**
- **The parent space is not copied, it can be shared by both the parent and the child process but the memory pages are marked as write protected.**
- **if parent or child wants to modify the pages, then kernel copies the parent pages to the child process.**
- **Advantage: Kernel can defer or prevent copying of a parent process address space.**

Process Tree Structure

- **In a Linux system no process is independent of any other process.**
- **Every process in the system, except the initial process has a parent process.**
- **New processes are not created, they are copied, or *cloned* from previous processes.**
- **Every task_struct representing a process keeps pointers to its parent process and as well as to its own child processes**
- **$pstree – process tree structure shows the process dependency.**

- **Parent process creates children processes, which, in turn create other processes, forming a tree of processes.**
- **Resource sharing**
  - **Parent and children share all resources.**
  - **Children share subset of parent's resources.**
  - **Parent and child share no resources.**
- **Execution**
  - **Parent and children execute concurrently.**
  - **Parent waits until children terminate.**
- **Address space**
  - **Child duplicate of parent.**
  - **Child has a program loaded into it.**

68

Resource Sharing

[For the examples, consider that you use fork() to create a child process]

- Parents and Children share all resources.
  - Example: printf() statement.
- Children share a subset of parent's resources
  - Example: Using if(!fork()) to separate parent and child code, so they can share some resources but whatever is in the if-else block will be exclusive to the parent and child, respectively.
- Parents and Children share no resources.
  - Example: Making an execl() call in the child process.

Time Stamp Counter

System can provide very high resolution time measurements through the time-stamp counter which counts the number of instructions since boot.

To measure Time Stamp Counter (TSC)

```
# include <sys/time.h>
unsigned long long rdtsc ( )
{
    unsigned long long dst;
    _ _asm_ _    _ _volatile_ _ (“rdtsc”:”=A” (dst));
    return dst;
 }
```

```
main ( )
 {
   long long int start, end;

   start = rdtsc();

   /* Give your job;  */

   end = rdtsc();

   printf (" Difference is : %llu\n", end - start);

}
   /*  This is the most accurate way of time
   measurement */
```


rdtsc = Read Time Stamp Counter
dst = Destination
__asm__ = Macro that specifies its assembly language
__volatile__ = Indicates variable value can keep changing. Compiler will check the latest value and assign it to the variable.
“rdtsc”:”=A”(dst) => Assembly code that stores time stamp counter value in variable dst.

rdtsc() is not included in any header file which is why we need to define it here.

The time() command exists to do the same task but the time stamp counter is very accurate in comparison because it uses some assembly instructions.

Note: Before using this program, you need to know your CPU’s frequency. That can be determined with the command “lscpu”.
Mine is 1552 MHz = 1.552 GHz approx. This indicates how many clock ticks happen per second.
So one clock tick in this case happens in 1 / (1.552 * 10^9) = approx 0.66 nanoseconds.

When we do “end - start” in the code, it will give us the clock ticks that have elapsed, so if we want to know the time that has elapsed in seconds then we’ll have to divide (end - start) by the frequency to get time taken in seconds.

Daemon Process

- **Daemon process starts during system startup**
- **They frequently spawn other process to handle services requests**
  - **Mostly started by initialization script /etc/rc**
- **Waits for an event to occur**
- **perform some specified task on periodic basis (cron job)**
- **perform the requested service and wait**
  - **Example print server**

- **executed as the background process**
- **Orphan process**
- **No controlling terminal**
- **run with super user privileges**
- **process group leaders**
- **session leaders**

- Some script/task that has to be executed periodically can be executed as a daemon process.
- It must be executed in the background.
- It must execute as an orphan process, otherwise its parent will forever be waiting for it to return but a daemon process by its nature is supposed to run forever.
- It should have no controlling terminal, i.e. it should not require any shell or terminal to execute. Otherwise if the terminal is closed, then the daemon process would get closed along with it.
- It should not be associated with any controlling terminal which is why it should be process group leader as well as session leader.
- It should run with super user privileges.

Daemon Process Creation

```
int  init_daemon ( void ) {
  if ( ! fork ( ) ) {
       setsid ( ) ;
       chdir ( " / " ) ;
       umask ( 0 ) ;
      /* Specify Your Job  */
       return ( 0 );
  }
  else
       exit ( 0 );
}
```

Process Types

- Parent
- Child
- Orphan
- Daemon

Process States

- Running (R)
- Stopped (T)
- Sleep
  - Interruptible (S)
  - Uninterruptible (D)
- Zombie (Z)

75

setsid() -> Create a new session and make this process the Session Leader and also the process group leader thereby making it not associated with any controlling terminal.

chdir(path) -> Changes the current working directory of the calling process to the directory specified in path.

umask(0) -> umask specifies the default permissions that are associated with any created file. 0777 - umask() is done to determine what will be the file permission. So when we call umask(0), we are giving all permissions to any files that will be created by this process.

exit(0) -> Terminates the parent process making it an orphan process.

The TTY parameter of the daemon process will be “?” in “ps” command output. Because it has no controlling terminal. And its parent should be a systemd process because it’s an orphan process.

24/09/21

Inter Process Communication (IPC)

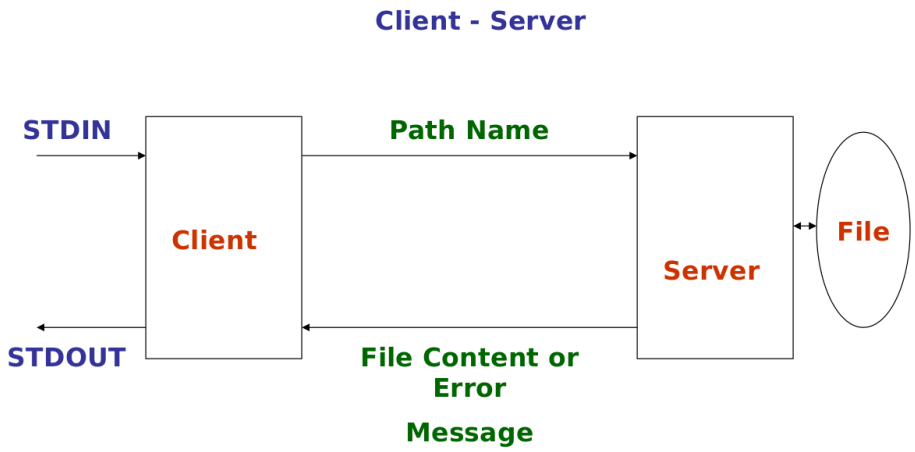
- In a multiprocessing environment, often many processes are in need to communicate with each other and share some of the resources.
- The shared resources must also be synchronized from the concurrent access by many processes.
- IPC mechanisms have many distinct purposes: for example

* Data transfer * Sharing data
* Event notification * Resource sharing
* Process control

- **Primitive**
  - Unnamed pipe
  - Named pipe (FIFO)
- **System V IPC**
  - Message queues
  - Shared memory
  - Semaphores
- **Socket Programming**

Socket Programming is like an unnamed pipe but its full duplex (two-way read-write allowed and both ends can communicate simultaneously), whereas unnamed pipe is only half duplex (two-way read-write allowed but only one end can communicate at a time).

Pipes



pipe (or unnamed pipe "|")

- On command line pipe is represented as "|"
- It can be used in the shell to link two or more commands
  - For example ls –Rl | wc
- Two ends of a pipe is represented as a set of two descriptors.
- A pipe is used to communicate between related processes.

- Half duplex
- Data is passed in order.
- Pipe uses circular buffer and it has zero buffering capacity
- The read and write system calls are blocking calls.

78

Related processes => There exists parent child relationships.
Example, ls -Rl | wc => The shell in which this command is being run is the parent of both "ls" and "wc" processes.

Imagine the pipe as a pipe that is open at both ends and carrying water. Once you send in water at one end, it will immediately reach the other end, so there is no buffering capacity.

"Read and write system calls are blocking calls" meaning that the read end of the pipe will not open until the write end is ready and the write end will not open until the read end is ready.

pipe() System Call

int pipe(int pipefd[2])
[Returns 0 on success, else -1]

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe and pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

pipe() returns fds which means it must exist in the fd table for the process. So execute this program that calls pipe() and then check its fd table. You will see two entries 'pipe:[some numbers]' which are the read and write ends of the pipe.

Example program to show how pipe() is used,

```c
int main() {
      int i, j, fd[2];
      char buf[80];
      i = pipe(fd);
      printf("i=%d\n", i);
      j = write(fd[1], "Hello\n", 7);
      read(fd[0], buf, j);
      printf("From pipe: %s\n", buf);
}
```

Here,

- We create a pipe.
- Write “Hello\n” into the write end of the pipe, fd[1].
- Read from the read end of the pipe, fd[0] and store it in buf
- Print contents of buf

A thing to note is that if you use fork(), then “fd” will be available to both parent and child processes and so one can use them to communicate between each other. But remember that the pipe is half duplex.

Data Transfer Using Pipe

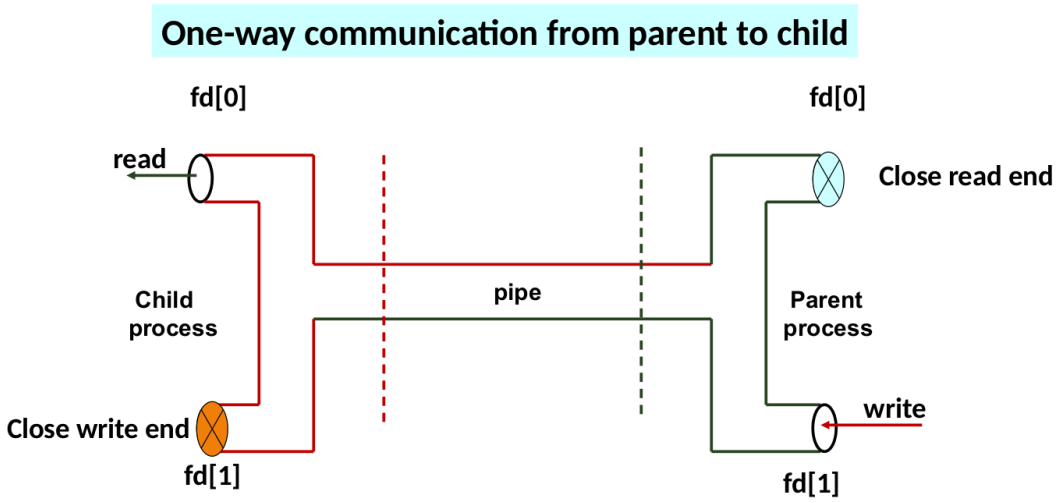
int fd[2];
pipe(fd);
-returns with fd[0], fd[1];

write(fd[1], ……);
read(fd[0], ……);

- **Create a pipe.**
- **Call fork.**
- **Parent can send data and child can read the data or vice versa.**
- **Unused ends (descriptors) should be closed.**

79

What does close unused ends mean?



80

So if we are using parent as the write-end and child as the read-end, then we must close the read-end for the parent and the write-end for the child (using close(fd[0]), close(fd[1]) calls). **Because otherwise if we write data into the pipe from the child process, then where should it go? Read-end of parent process or Read-end of child process? That's why we close the ends appropriately.**

Program for One-Way Data Transfer

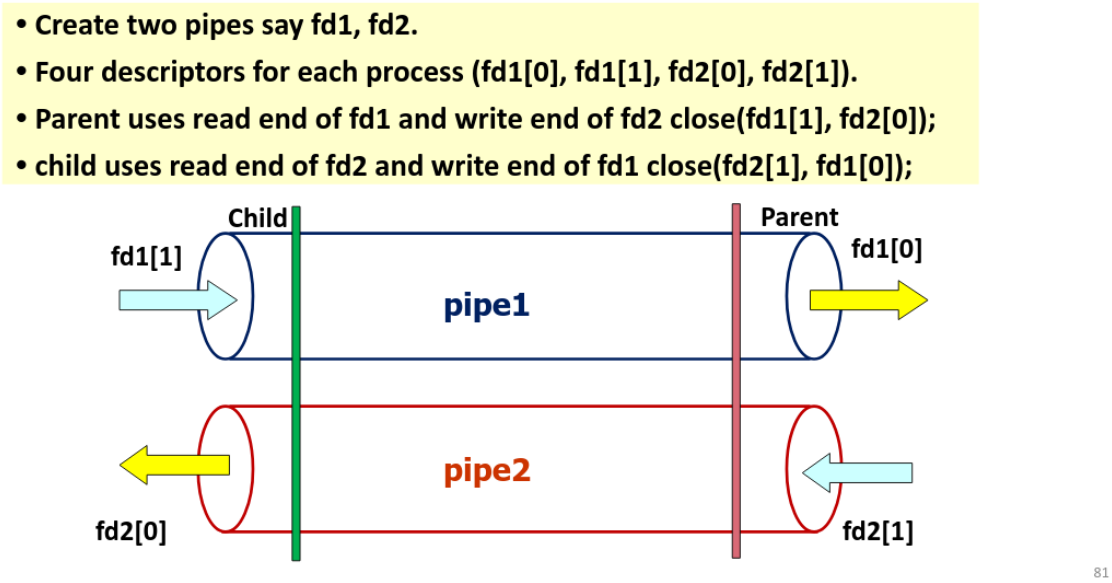
```
int main() {
	char buff[80];
	int fd[2];
	pipe(fd);

	if (fork()) {
		close(fd[0]); // Close read-end of pipe for parent
		printf("Enter message to the child: ");
		scanf(" %[^\n]", buff); // Read till newline encountered
		write(fd[1], buff, sizeof(buff));
	}
	else {
		close(fd[1]); // Close write-end of pipe for child
		read(fd[0], buff, sizeof(buff));
		printf("Message from parent: %s\n", buff);
	}
	wait(0); // Wait till all children have terminated
}
```

An important reason as to why this even works is to remember that read() and write() are

blocking calls. So in this case, the child process waits till data is available for reading on the read-end of the pipe. Also, if read-end is not available for reading, then write-end will wait till that happens to write the data.

Two Way Data Transfer



[We cannot keep both ends open for a single pipe because pipe is half duplex and simultaneous writes or reads can happen in that case]
fd1 -> For parent to read and for child to write
fd2 -> For child to read and for parent to write

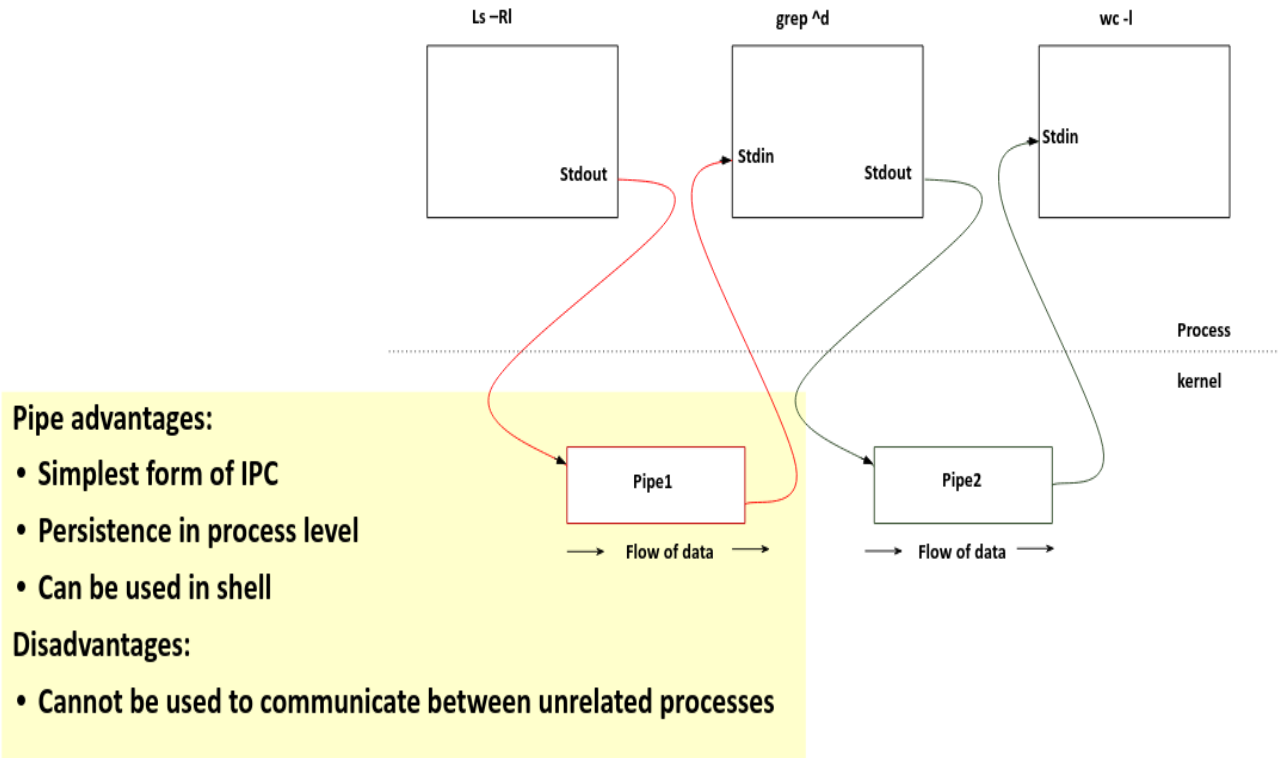
This gets really confusing and hectic so it will be very helpful to draw a diagram like this to visualize which process is using which end.

```
if (!fork()) {
      close(fd1[0]); // Close read-end of fd1 for child
      close(fd2[1]); // Close write-end of fd2 for child
      printf("Enter message to parent: ");
      scanf(" %[^\n]", buff1);
      write(fd1[1], buff1, sizeof(buff1));
      read(fd2[0], buff2, sizeof(buff2));
      printf("Message from parent: %s\n", buff2);
}
else {
      close(fd1[1]); // Close read-end of fd1 for parent
      close(fd2[0]); // Close read-end of fd2 for parent
      // Child wrote into fd1 first, so read from fd1 first
      read(fd1[0], buff1, sizeof(buff1));
```

```
        printf("Message from child: %s\n", buff1);
        printf("Enter message to child: ");
        scanf(" %[^\n]", buff2);
        write(fd2[1], buff1, sizeof(buff1));
}
```

Note how there is a sequence followed in parent and child, first child writes and parent reads, then parent writes and child reads. The blocking call mechanism of read() and write() handles the synchronization part.
So you cannot have both parent and child writing first and reading second. As the read-ends will not be available for reading in both of the pipes because they are both attempting to write into the respective pipes.

Execution of ls -Rl | grep ^d | wc



We will have to create 3 processes to execute each of the 3 commands.

“ls -Rl” will print to STDOUT but we want to redirect its output to the write-end of pipe1.
So first we will close stdout fd for the process, then call dup(fd[1]). This will assign the lowest available fd value to fd[1] which is the write-end of pipe1 and lowest possible fd value is 1 => STDOUT => Output of “ls -Rl” will be written to the write-end of pipe1.

Similar logic will have to be used to use read-end of pipe1 as stdin for the second process, there you’ll have to use close(0).

Program to execute “ls -l | wc”

```
int main() {
	int fd[2];
	pipe(fd);

	if (!fork()) {
		close(1); // close STDOUT
		close(fd[0]);
		dup(fd[1]); // Duplicate fd[1] to lowest fd value available = 1
		execlp("ls", "ls", "-l", (char*) NULL);
		// execlp() will write output of "ls -l" to fd with value = 1
(write-end of pipe)
	}
	else {
		close(0); // close STDIN
		close(fd[1]);
		dup(fd[0]); // Duplicate fd[0] to lowest fd value available = 0
		execlp("wc", "wc", (char*) NULL);
		// execlp() will read input from fd with value = 0 (read-end of
pipe) as input to “wc” command and show output to fd with value = 1 =>
STDOUT
	}
}
```

We can also use dup2() instead of dup() as,
dup2(fd[0], 0) to replace STDIN by read-end of pipe
dup2(fd[1], 1) to replace STDOUT by write-end of pipe
Remember that dup2() will close the fd specified if it is in use. So STDOUT and STDIN will be automatically closed by dup2() itself.

We can also use fcntl() with F_DUPFD instead of dup() as,
fcntl(fd[1], F_DUPFD, 0);
We have to close the existing fd beforehand in this case though.

Program to execute “ls -Rl | grep ^d | wc”

```
int main() {
	int fd1[2], fd2[2];

	pipe(fd1);
	pipe(fd2);
```

```
	// This child will only be used for writing output of "ls -Rl" to
pipe 1 write-end = STDOUT
	if (!fork()) {
		dup2(fd1[1], 1); // Write-end of pipe 1 = new STDOUT
		close(fd1[0]);
		close(fd2[0]);
		close(fd2[1]);
		execlp("ls", "ls", "-Rl", (char*) NULL);
	}
	else {
		// This child is responsible for running "ls -Rl | grep ^d"
command where input is received from read-end of pipe 1 = STDIN and output
is written to write-end of pipe 2 = STDOUT
		if (!fork() {
			dup2(fd2[1], 1);
			dup2(fd1[0], 0);
			close(fd1[1]);
			close(fd2[0]);
			execlp("grep", "grep", "^d", (char*) NULL);
		}
		// This parent is responsible for running "ls -Rl | grep ^d |
wc" command where input is received from read-end of pipe 2 = STDIN and
output is written to STDOUT
		else {
			dup2(fd2[0], 0);
			close(fd2[1]);
			close(fd1[0]);
			close(fd1[1]);
			execlp("wc", "wc", (char*) NULL);
		}
	}
}
```

If you want to test output of such commands via C programs you can use the system() and popen() system calls, e.g. system("ls -Rl | grep ^d | wc");
But system() is highly discouraged to use and popen() is not recommended for system programmers. However for just testing purposes it's fine.

28/09/21

FIFO

- FIFO works much like a pipe
  - Half duplex, data passed in FIFO order, circular buffer and zero buffering capacity.
- FIFO is created on a file system as a device special file
- It can be used to communicate between unrelated processes
- It can be reused.
- Persist till the file is deleted.

- FIFO can be created in a shell by using mknod or mkfifo command.
  - mknod myfifo p
  - mkfifo a=rw myfifo
- In a C program mknod system call or mkfifo library function can be used.
  - int mkfifo (char *file_name, mode_t mode);
  - int mknod (char *file_name, mode_t mode, dev_t dev);
    - mknod("./MYFIFO", S_IFIFO|0666, 0);

84

- Pipe could only be used between related processes, i.e. parent and child whereas FIFO can be used to communicate between unrelated processes as well.
- This is possible because the FIFO itself is created as a file that processes can open in read or write mode as per requirements.
- It can be reused because even if the processes using it for communicating have terminated, the FIFO itself is still stored on disk and can later be used by other processes for communication.
- Recall that FIFO is a pseudo device special file.

- Once a FIFO is created, you can use file's related system calls (open, read, write, select, close etc.,) to access the FIFO.
- For example: Process 1 may open a FIFO in write only mode and write some data.
- Process 2 may open the FIFO in read only mode, read the data and display on the monitor.

FIFO - Disadvantages

- Data cannot be broadcast to multiple receivers.
- If there are multiple receivers, there is no way to direct to a specific reader or vice versa.
- Cannot store data and you cannot use FIFO across network.
- Less secure than a pipe, since any process with valid access permission can access data.
- No message boundaries. Data is treated as a stream of bytes.

FIFO Limitation

- System imposed limits on pipes
  1. Maximum number of files can be open within a process is determined by OPEN_MAX macro.
  2. Maximum amount of data that can be written to a pipe of FIFO atomically is determined by PIPE_BUF macro (size of a circular buffer).

```
#include <unistd.h>
main () {
 long PIPE_BUF, OPEN_MAX;
 PIPE_BUF = pathconf(".", _PC_PIPE_BUF);
 OPEN_MAX = sysconf (_SC_OPEN_MAX);
 printf ("Pipe_buf = %ld\t OPEN_MAX =
                    %ld\n", PIPE_BUF, OPEN_MAX);
}
```


mkfifo() Library Function and mknod() System Call

mknod() is to create device special files which include FIFO by specifying the S_IFIFO parameter.
mkfifo("filename", file_permissions); // Returns 0 on success else -1

mknod("filename", S_IFIFO | file_permissions, device_special_number);
For FIFO files, device_special_numbers = 0 since it is a pseudo device special file.

Example: mkfifo(“myfifo”, 0744);

Which one is the better command, mknod or mkfifo?
Answer: Analyze using strace command.

strace Command

System call trace command.
It executes the command specified to strace and displays the execution process in sequence with each system call that is executed as part of the command.
The shell first forks a child and then the child calls execve() with the specified command and then returns the result to its parent, i.e. the shell.

For example,
strace mkfifo myfifo => Creates FIFO file with name “myfifo”
You can check the output returned and you’ll see that mkfifo actually calls mknod() and this mkfifo() is just doing some extra steps on top of mknod() because mkfifo() is a library function whereas mknod() is the actual system call.
Therefore, mknod() is always going to be faster than mkfifo().

You can also handily debug system calls from the terminal using strace!

-c option will tell you the time taken for each system call used and how many times those system calls were called.

One-Way Communication using FIFO file

Once the FIFO file is created we can communicate between two processes using it.

Example programs for demonstrating this.
Writing to FIFO file

```
int main() {
      fd = open("myfifo", O_WRONLY);
      printf('Enter the text:");
      scanf(" %[^\n]", buff);
      write(fd, buff, sizeof(buff));
}
```

Reading from FIFO file

```
int main() {
      fd = open("myfifo", O_RDONLY);
      read(fd, buff, sizeof(buff));
      printf("The text from FIFO file: %s\n", buff);
```

```
}
```

Now open two terminals and execute both programs and you can see the blocking calls in action and how FIFO communication happens.

Two-Way Communication using FIFO File

Create two FIFO files and use the same logic like we did with pipes except here we don't have to worry about closing the unused end of the pipes.

Program 1

```
int main() {
      fd1 = open("myfifo1", O_WRONLY);
      fd2 = open("myfifo2", O_RDONLY);
      printf("Enter the text:");
      scanf(" %[^\n]", buff1);
      write(fd, buff1, sizeof(buff1));

      read(fd2, buff2, sizeof(buff2));
      printf("The text from FIFO file: %s\n", buff2);
}
```

Program 2 is just the opposite of this, it first reads from "myfifo1" and then writes to "myfifo2".

Using select() with FIFO files

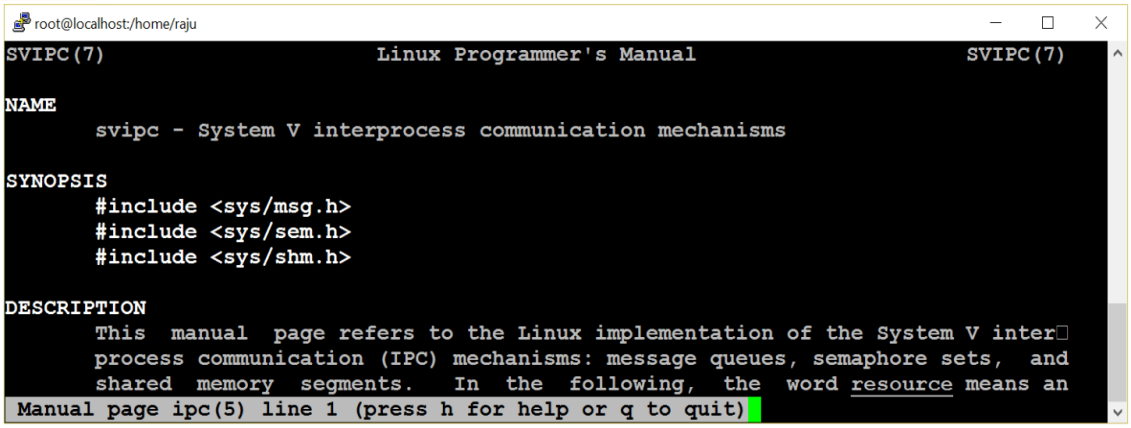
If the write-end or read-end is not open then the other end will keep on waiting forever for this end to open.
Note: Blocking will happen at the open() call itself.
Example, execute the read program but don't open the write program.

You can use the select() system call to specify a time-limit that you would like to wait for.
So once the timer expires and if select() returns 0, then you can just skip the read() or write() call and go on to do other things.

```
int main() {
      fd_set rfds;
      struct timeval tv;
      tv.tv_sec = 5;
      fd = open("myfifo", O_RDONLY);
      FD_ZERO(&rfds);
      FD_SET(fd, &rfds);
```

```
      if (!select(fd + 1, &rfds, NULL, NULL &tv))
            printf("No data is available for reading yet\n");
      else {
            printf("Data is available now\n");
            read(fd, buff, sizeof(buff));
            printf("Data from FIFO: %s\n", buff);
      }
      // Do other tasks
}
```

System V IPC



- **Pipe and FIFO do not satisfy many requirements of many applications.**
- **Sys V IPC is implemented as a single unit**
- **System V IPC Provides three mechanisms namely : MQ, SHM and SEM**
- **Persist till explicitly delete or reboot the system**

- Includes Message Queues, Semaphores & Shared Memory
- They fall under the same package and their details can be checked at “man 7 svipc”.
- All these mechanisms are kernel persistent because such communication happens in the kernel space.
- ipcs command can be used to show specifications of these mechanisms.

System V IPC Attributes

- **Each IPC objects has the following attributes.**
  - **key**
  - **id**
  - **Owner**
  - **Permission**
  - **Size**
    - **Message queue – used-bytes, number of messages**
    - **Shared memory – size, number of attach, status**
    - **Semaphore – number of semaphores in a set**
- **The ipc_perm structure holds the common attributes of the resources.**

Key:

- **the first step is to create a shared unique identifier.**
- **The simplest form of the identifier is a number**
- **the system generates this number dynamically by using the *ftok* library function.**
- **Syntax: key_t ftok (const char *filename, int id);**

88

To create message queues, semaphores or shared memory you need to have a unique key.
This is the System V IPC Key.
This key for System V IPC Mechanisms is analogous to the file descriptors used for files.

To get this key you have to use the ftok() function,
key_t ipcv_key = ftok(char *pathname, int proj_id);

pathname = Path to some existing file
The ftok() function uses the pathname and the least significant 8 bits of the proj_id to generate the System V IPC Key.

Note: You can in practice pass anything in as the IPC key, like “Name + Roll Number” but the only requirement is that it should be unique. So that’s why the ftok() function is handy in the first place.

ID

- The syntax for a *get* function is:

int xxxget (key_t key, int xxxflg);

(xxx may be msg or shm or sem)

- If successful, returns to an identifier; otherwise -1 for error.
- The key can be generated in three different ways
  - from the *ftok* library function
  - by choosing some static positive integer value
  - by using the IPC_PRIVATE macro
- flags commonly used with this function are IPC_CREAT and IPC_EXCL.

- The syntax for the *control* function is:

int xxxctl (int xxxid, int cmd, struct xxxid_ds *buffer); (xxx may be msg or shm or sem);

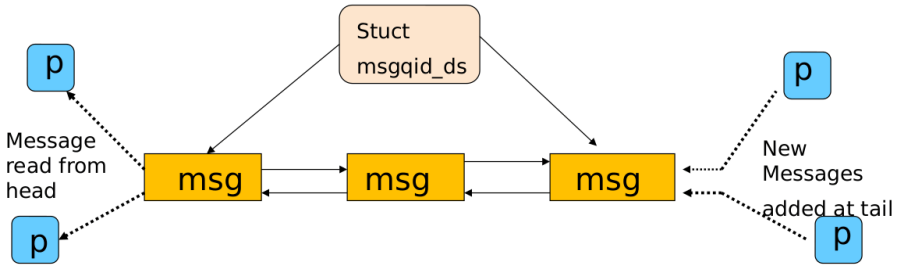
- If successful, the *xxxctl* function returns zero, otherwise it returns -1.
- The command argument may be
  - IPC_STAT
  - IPC_SET
  - IPC_RMID

89

More details on these later.
xxxcntl() is similar to fcntl() for files.

Message Queues

- Message queue overcomes FIFO limitation like storing data and setting message boundaries.
- Create a message queue
- Send message (s) to the queue
- Any process who has permission to access the queue can retrieve message (s).
- remove the message queue.



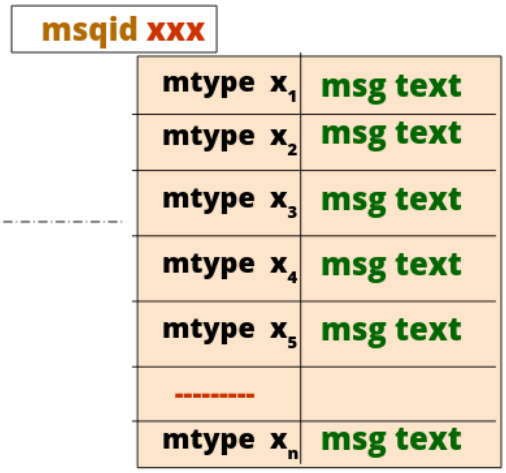
90

So with Message Queues we can actually store the messages and use any type of messages and not be limited by just character data as was the case with FIFO.
Note that the Message Queue is a Doubly Linked List.

Check Message Queues created using “ipcs -q”.

Message Structure

```
struct msgbuf {
        long mtype;
        char mtext [1];
}; Standard structure
```




```
struct My_msgQ {
        long mtype;
        char mtext [1024];
        void *   xyz;
}; Our own structure
```

So this is how we can send custom structured messages as well to the queue.

msgget()

Used for creating or getting the identifier for a Message Queue.

int msgget(key_t key, int msgflg);

msgflg is like the flags parameter in open().

A message queue is created if,

- "key" = IPC_PRIVATE
- No message queue with given key exists and msgflg contains "IPC_CREAT|file_permissions"

If a message queue with a given key exists and msgflg = 0, then msgget() returns the unique identifier for the message queue.

msgsnd() & msgrcv()

[man 2 msgsnd() for more details]
msgsnd() is used for sending messages to a Message Queue.
msgrcv() is used for receiving messages from a Message Queue.

int msgsnd(int msqid, void* msgp, size_t msgsz, int msgflg);

ssize_t msgrcv(int msqid, void* msgp, size_t msgsz, long msgtyp, int msgflg);

int msqid => Unique id for the message queue which you'll get from msgget()

void* msgp => Pointer to msgbuf structure
The msgbuf structure defines the message that is to be sent and looks as follows,

```
struct msgbuf {
        long mtype; // Message type, must be > 0
        char mtext[length_of_message]; // Message data, can be a struct as well instead of char
}
```

msgtype Argument

- **Each message in a message queue must have an id, this id is specified by msgbuf.mtype attribute.**
- **Note: This id need not be unique,** so there can be multiple messages in the same queue with the same mtype attribute, in which case, FIFO order will be observed between them.
- So while sending a message, you have to update msgbuf struct with the mtype.
- While receiving the message, the "msgtyp" parameter can be interpreted in different ways as follows,
    - 0 = Retrieve in FIFO order
    - +ve integer = Retrieve message with exact message type value
    - -ve integer = Retrieve first message with message type <= to the absolute value of integer specified.

size_t msgsz => Specifies the length of msgp->mtext array

int msgflg => 0 or more Bitwise OR'd flags which can be checked in the man page.
Usually we pass msgflg = 0 or IPC_NOWAIT

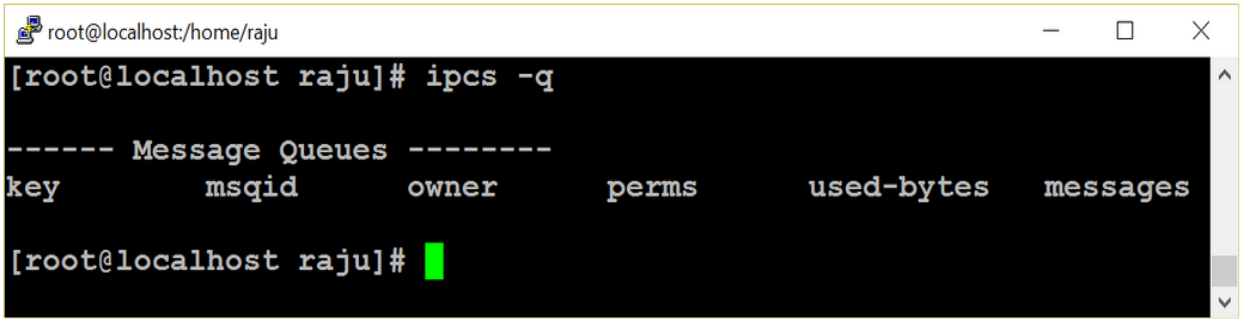
As mentioned before, these mechanisms have kernel persistence, so when a message is sent, it is copied from user buffer to kernel buffer. So there should be space available in the kernel buffer for the message.

For msgsnd(),
If msgflg = 0, then msgsnd() waits till enough space is available.
If msgflg = IPC_NOWAIT, then msgsnd() doesn't wait and will return -1 => error.

For msgrcv(),
If msgflg = 0, then msgrcv() waits till a message of specified msgtyp is available in the message queue.
If msgflg = IPC_NOWAIT, then msgrcv() doesn't wait and will return -1 => error.

Message Queue Limitations

- Message queues are effective if a small amount of data is transferred.
- Very expensive for large transfers.
- During message sending and receiving, the message is copied from user buffer into kernel buffer and vice versa
- So each message transfer involves two data copy operations, which results in poor performance of a system.
- A message in a queue can not be reused



94

A message in a queue cannot be reused => Once a process reads a message from the queue it will be removed from the queue and so that message cannot be read again.

Program to demonstrate Message Queues

Creating a Message Queue

```
int main() {
      // 'a' = ASCII value of "a" given as proj_id and "." will be string
used to create key
      key = ftok(".", 'a');
      msgid = msgget(key, IPC_CREAT|IPC_EXCL|0744);
      // %0x for Hexadecimal value
      printf("key=0x%0x\t msgid=%d\n", key, msgid);
}
```

Sending message to a Message Queue

```
int main() {
      struct msg {
            long int m_type;
            char message[80];
      } myq;
      key = ftok(".", 'a');
```

```
      mqid = msgget(key, 0);
      printf("Enter message type: ");
      scanf("%ld", &myq.m_type);
      printf("Enter message text:");
      scanf("%[^\n]", myq.message);
      size = strlen(myq.message);
      // size + 1 to accommodate terminating character
      msgsnd(mqid, &myq, size + 1, 0);
}
```

Receiving message from a Message Queue

```
int main() {
      struct msg {
      long int m_type;
      char message[80];
      } myq;
      key = ftok(".", 'a');
      mqid = msgget(key, 0);
      printf("Enter message type: ");
      scanf("%ld", &myq.m_type);
      ret = msgrcv(mqid, &myq, sizeof(myq.message), myq.m_type,
IPC_NOWAIT|MSG_NOERROR);
      if (ret == -1)
            exit(-1);

      printf("Message type: %ld\n Message: %s\n", myq.m_type, myq.message);
}
```

msgcntl() can be used to get msgid_ds and ipc_perm structures which will tell us metadata about the message queue.

Also do note that msgsnd() and msgrcv() are non-blocking calls with NO_WAIT specified. And even in the regular case, they are blocking only in very special cases like the kernel buffer is full or the message type doesn't exist in the queue which is very different from the FIFO case where blocking was a mandatory condition required for it to function.
This is because FIFO has zero buffering capacity whereas Message Queues can actually store data.

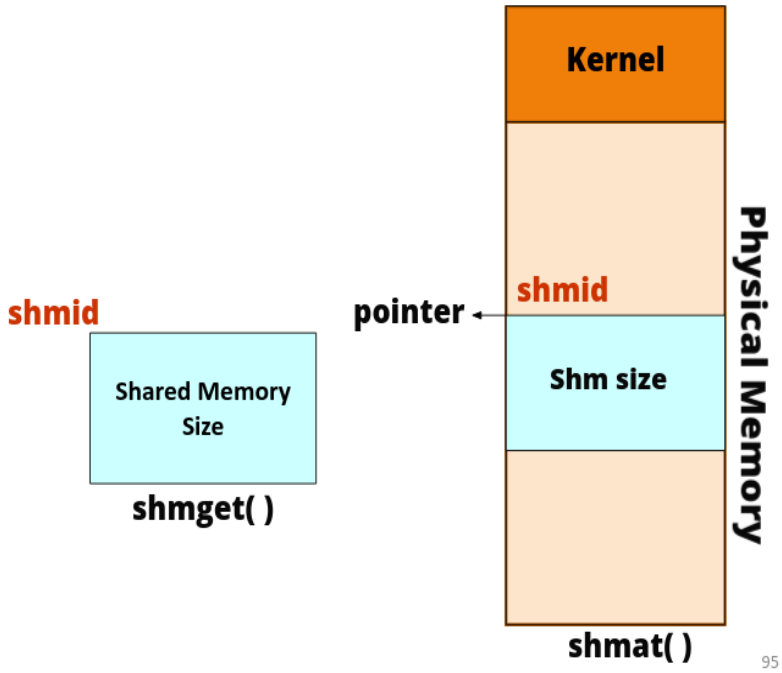
01/09/21

Shared Memory

There were some disadvantages with Message Queues like it is inefficient because of the two copy operations from kernel to user buffers and vice versa and messages cannot be reused, etc.
These can be overcome by the concept of Shared Memory.

Introduction

- Very flexible and ease of use.
- Fastest IPC mechanisms
- shared memory is used to provide access to
  - Global variable
  - Shared libraries
  - Word processors
  - Multi-player gaming environment
  - Http daemons
  - Other programs written in languages like Perl, C etc.,



We create a shared memory of a given size using shmget(). Creating a shared memory block will return its id “shmid” which is a pointer to the start of the shared memory and using this pointer we can now access the shared memory.

We have to then attach it to physical memory for which we’ll use shmat(). This will actually allocate physical memory for it **in the user space** which is why it is the fastest IPC mechanism.

We’ll specify whether shared memory is read-only and whether it is executable while calling shmat(). Because the OS supports virtual memory and memory management capabilities, the attachment part is done by the kernel itself to some unused space. In embedded systems or microcontrollers, no memory management module is present and so physical memory mapping layout should be known and actual physical address must be specified.

- Shared memory is a much faster method of communication than either semaphores or message queues.
- Does not require an intermediate kernel buffer
- Using shared memory is quite easy. After a shared memory segment is set up, it is manipulated exactly like any other memory area.

- The steps involved are
  - Creating shared memory
  - Connecting to the memory & obtaining a pointer to the memory
  - Reading/Writing & changing access mode to the memory
  - Detaching from memory
  - Deleting the shared segment

shm System Calls

- shmget system call is used to create a shared memory segment.
- The syntax:

  int shmget (key_t key, int size, int shmflg);

  key: the return value of ftok function.

  size: size of the shared memory.

  shmflg: IPC_CREAT|0744
- On success the shmget returns the shared memory ID or else it returns -1.

- used to attach the created shared memory segment onto a process address space.
- void *shmat(int shmid,void *shmaddr,int shmflg)
- Example: data=shmat(shmid,(void *)0,0);
- A pointer is returned on the successful execution of the system call and the process can read or write to the segment using the pointer.

For shmget() the same logic follows as that with msgget().
If you pass shmflag = 0 and key is not IPC_PRIVATE, it will return shmid.
If key=IPC_PRIVATE or shared memory with given key doesn't exist, and shmflag specifies IPC_CREAT, then it will create a shared memory segment.

For shmat() we have to specify the starting address of the shared memory but we can have memory management of the OS handle it, so we just pass 0 and let the kernel handle it (we have to cast it to void* because that's what the required parameter type is).
shmflg specifies the following,
shmflg = 0, Read-Write privileges
shmflg = SHM_RDONLY, Read-only privileges
There are a few other values (check man page) but not that relevant.

Reading & Writing to Shared Memory

- **Reading or writing to a shared memory is the easiest part.**
- **The data is written on to the shared memory as we do it with normal memory using the pointers**

Example -

- **Read:**
  - **printf ("SHM contents : %s \n", data);**
- **Write:**
  - **printf (""Enter a String : ");**
  - **scanf (" %[^\n]",data);**

- **The detachment of an attached shared memory segment is done by shmdt to pass the address of the pointer as an argument.**
- **Syntax: int shmdt (void *shmaddr);**
- **To remove shared memory call:**

int shmctl (shmid,IPC_RMID,NULL);

- **These functions return –1 on error and 0 on successful execution.**

Note: data => Return value of shmat(), i.e. shmid pointer.

Detaching => Removing shared memory segment from physical memory but its entry will still be present.
To remove even the entry you have to use shmctl(shmid, IPC_RMID, NULL);

- shmid = shmget (key, 1024, IPC_CREAT|0744);
- void *shmat (int shmid, void *shmaddr, int shmflg); if the shm is read only pass SHM_RDONLY else 0
- (void *)data = shmat (shmid, (void *)0, 0);
- int shmdt (void *shmaddr);
- int shmctl (shmid, IPC_RMID, NULL);

- Data can either be read or written only. Append ?
- Race condition
  - Since many processes can access the shared memory, any modification done by one process in the address space is visible to all other processes.
  - Since the address space is a shared resource, the developer should implement a proper locking mechanism to prevent the race condition in the shared memory.

99

You cannot append data to a Shared Memory, i.e., you cannot find out the “end” of a Shared Memory segment at the end of which you can append new data. So you can only read existing data or write new data or overwrite existing data.
This is a disadvantage of Shared Memory.

Multiple processes can access Shared Memory simultaneously so naturally race condition comes into the picture.
Reading from Shared Memory is allowed for any number of processes but only one process should be able to write into Shared Memory at once. This responsibility is given to the developer and the developer should write programs such that they use locking protocol to write into shared memory and ensure no race conditions.

Programs to demonstrate Shared Memory

Writing into a Shared Memory

```
int main() {
      int key, shmid;
      char *data;
      key = ftok(".", 'b');
      // Create shared memory of size 1024 bytes
      shmid = shmget(key, 1024, IPC_CREAT|0744);
      // Return value of shmat is pointer to start of shared memory which
we can just as char* pointer for reading and writing
      data = shmat(shmid, 0, 0);
```

```
	printf("Enter the text: ");
	scanf("%[^\n]", data);
}
```

Check the Shared Memory segments in your system by running “ipcs -m”.
The nattch column tells you how many processes are using that Shared Memory segment.

Reading from a Shared Memory

```
int main() {
	int key, shmid;
	char *data;
	key = ftok(".", 'b');
	// Pass third parameter as zero to get shmid of existing shared
memory
	shmid = shmget(key, 1024, 0);
	// Return value of shmat is pointer to start of shared memory which
we can just as char* pointer for reading and writing
	data = shmat(shmid, 0, 0);
	printf("Text from shared memory: %s\n", data);
	// This is just to check nattch value in ipcs -m output
	getchar();
}
```

You can try running this program from several terminals and see that multiple processes can access your shared memory block at the same time.

Semaphores

- Synchronization Tool
- An Integer Number
- P ( ) And V ( ) Operators
- Avoid Busy Waiting
- Types of Semaphore

Used in :

- shared memory segment
- message queue
- file

100

Semaphores are a synchronization tool and not used for message passing like the other IPC mechanisms.
Each Semaphore stores an integer value which indicates its status.
This integer is initialized to maximum possible value at the start.

Semaphores are of two types, Binary and Counting.

If max value = 1, then it is a Binary Semaphore.
If max value > 1, then it is a Counting Semaphore.

Application for Counting Semaphore, we only want to allow 100 concurrent users on a website, so you have a Counting Semaphore with value = 100 and it decrements for each user logging in and increments for each user logging out. If the semaphore value ever becomes zero, then all new users should be blocked until someone logs out.
So being able to login is the critical section in this case and we're controlling entry to it using the semaphore.

Note: Mutex is the same as binary semaphore.

Semaphores are like signals and they do two operations P() and V(), one is for locking or decrementing semaphore value and the other is for unlocking or incrementing semaphore value. These are atomic operations (executed as single instruction) so they cannot be interrupted in between.

Busy Waiting

When a process in execution requires some data or something for which it waits but it waits in the CPU itself then that is called as busy waiting because the CPU is kept busy due to this waiting.
In the case of semaphores, if a process in execution needs to wait for a semaphore then it'll be sent to the sleep state and won't result in busy waiting.

We can use semaphores in Shared Memory segments, Message Queues and files for locking purposes.

Semaphore Operations

Incrementing Operations:

```
int v (int i)
 {
  i = i + 1; (unlock)
  return i;
 }
```

Decrementing Operations:

```
int p (int i)  {
  if (i > 0)
   then
    i--; (lock)
   else
    wait till i > 0;
   return i;
  }
```

- If a process wants to use the shared object, it will "lock" it by asking the semaphore to decrement the counter
- Depending upon the current value of the counter, the semaphore will either be able to carry out this operation, or will have to wait until the operation becomes possible
- The current value of counter is >0, the decrement operation will be possible. Otherwise, the process will have to wait

- System V semaphore provides a semaphore set - that can include a number of semaphores. It is up to user to decide the number of semaphores in the set
- Each semaphore in the set can be a binary or a counting semaphore. Each semaphore can be used to control access to one resource - by changing the value of semaphore count

101

[Remember these two operations for exam]
System V provides a semaphore set which is like an array of semaphores.
This is useful because one semaphore can only be used to synchronize one resource, i.e. one critical section, so if your program has like 10 critical sections then you'll need 10 semaphores.
This semaphore set concept avoids the hassle required with the previous approach.
All of the semaphores in a set will be controlled by a single semaphore id.

Semaphore Creation

```
union semun {
 int val;     // value for SETVAL
 struct semid_ds *buf;  // buffer for IPC_STAT,
  IPC_SET
 unsigned short int *array;  // array for GETALL,
  SETALL
};

union semun arg;
semid = semget (key, 1, IPC_CREAT | 0644);
arg.val = 1;
/* 1 for binary else > 1 for Counting Semaphore */
semctl (semid, 0, SETVAL, arg);
```

Semaphore Definition

unsigned short semval;   /* semaphore value */
unsigned short semzcnt; /* # of threads waiting for semaphore to become zero */
unsigned short semncnt; /* # of threads waiting for semaphore to increase */
pid_t sempid;   /* PID of process that last accessed semaphore */

Union semun

This union is used to modify properties of a semaphore via the semctl() method.
It is analogous to flock struct with file locking.
Here we use union instead of struct.
Key difference between the two is that size of union is the size of its largest member whereas size of struct is the sum of sizes of all its members.
Depending on whether we use semctl with SETVAL, IPC_STAT or IPC_SET, GETALL or SETALL, the corresponding member in semctl is used.
We'll only require SETVAL and IPC_STAT, IPC_SET commands.

semget() method

Creates a semaphore set and returns it id or returns semid for existing semaphore set.
int semget(key_t key, int nsems, int semflg);
key = Output of ftok()
nsems = Number of semaphores in set
semflg = Same as that of shmflg and msgflg in Shared Memory and Message Queues.

semctl() method

Once a semaphore set has been created, we have to specify the type of each semaphore in the set, i.e. binary or counting.
We do this using the semun union and setting val = 1 for binary semaphore else > 1 for counting semaphores.
Then we call semctl()
int semctl(int semid, int semnum, int cmd, ...)
semnum = Semaphore number in the set that we are trying to modify with this call.
In the example given in the slide, we had created a semaphore set with only 1 semaphore in it, so semnum can only take the value of 0 => First semaphore in the set.

Then we apply the semun union on the semaphore set as,
semctl(semid, 0, SETVAL, arg); // arg = semun union variable

This is analogous to what we did with file locking.
We first defined the flock struct values like lock_type (semnum union here), then opened the file using open() to get fd (semget() to get semaphore id) and then applied lock on the file using fcntl(SETLKW) (semctl() call with SETVAL).

Check semaphore arrays/sets present in the system using the “ipcs -s” command.

Semaphore Operations

```
struct sembuf {
  short sem_num;  /* semaphore number: 0 means first */
  short sem_op;   /* semaphore operation: lock or unlock */
  short sem_flg;  /* operation flags : 0, SEM_UNDO, IPC_NOWAIT */
   };
struct sembuf buf = {0, -1, 0};  /* (-1 + previous value) */
semid = semget (key, 1, 0);

semop (semid, &buf, 1); /* locked */
-----Critical section--------
buf.sem_op = 1;
semop (semid, &buf, 1); /* unlocked */
```

The general flow is very similar to file locking and unlocking.

semop() Semaphore Operations

Now that semaphores are created we want to perform operations on it.
For it we’ll need to specify which semaphore we are referring to and which operations we want to perform on it.

int semop(int semid, struct sembuf *sops, size_t nsops);

Here,
nsops = Number of semaphores in the semaphore set
*sops = Array of sembuf structs of size “nsops” that defines the operation to be performed on each semaphore in the set.

sembuf Struct

It’s members are sem_num, sem_op and sem_flg.

sem_num = Semaphore Number in the set

sem_op can take values of 3 types, > 0, 0 and < 0

- If sem_op > 0, then sem_op will be added to the semval of the specified semaphore. So **it is the increment operation (or unlock operation) for the semaphore.**
- If sem_op = 0, then the operation waits (process enters sleep state) till semval becomes zero after which the process will be able to proceed.
- If sem_op < 0, then the operation waits (process enters sleep state) till semval becomes greater than or equal to the absolute value of sem_op after which the process can proceed. **This is the decrement operation (or lock operation) for the semaphore.**
- It just provides extra facility for acquiring multiple locks at the same time. For example, if sem_op = -3, then the process will wait till semval becomes 3 meaning that 3 locks can be acquired once the process comes out of wait/sleep state.

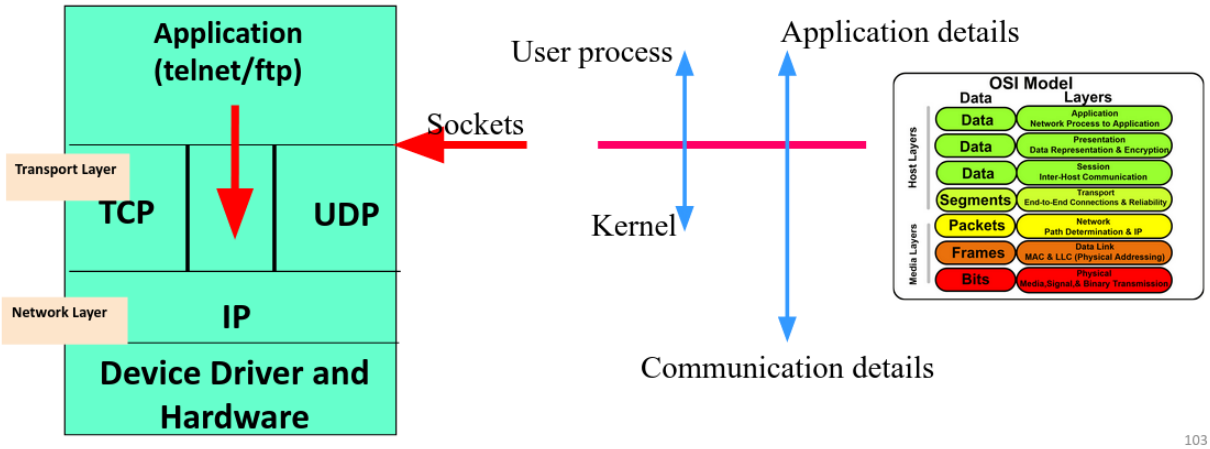
sem_flg can take three values

- 0 - No flag specified, default behaviour for sem_op as specified above.
- IPC_NOWAIT - All wait operations specified by sem_op parameter become no wait operations. If the condition for proceeding is not available then command fails immediately and returns -1
- SEM_UNDO - If a process terminates then the semaphore’s semval will be incremented by the number of locks that it had to show that its locks have been released. So situations where a process terminates without releasing the locks preventing other processes from accessing semaphores will be avoided.

Note: All of this syntax is a bit complicated because binary as well as counting semaphores concepts are implemented using the same functions.

Socket Programming

A socket is used to communicate between different machines (different IP addresses). Socket of type SOCK_STREAM is full-duplex byte streams.



103

Sockets are similar to unnamed pipes but they support full duplex communication and communication across the network.
Socket will reside between the Application & Transport Layers.
We can use TCP or UDP for the socket based on which it will be determined whether socket communication has to be fully online (like a phone call) or partially online (like a WhatsApp message), respectively.

Socket will connect to the Network layer and then access the Device Driver and Hardware.

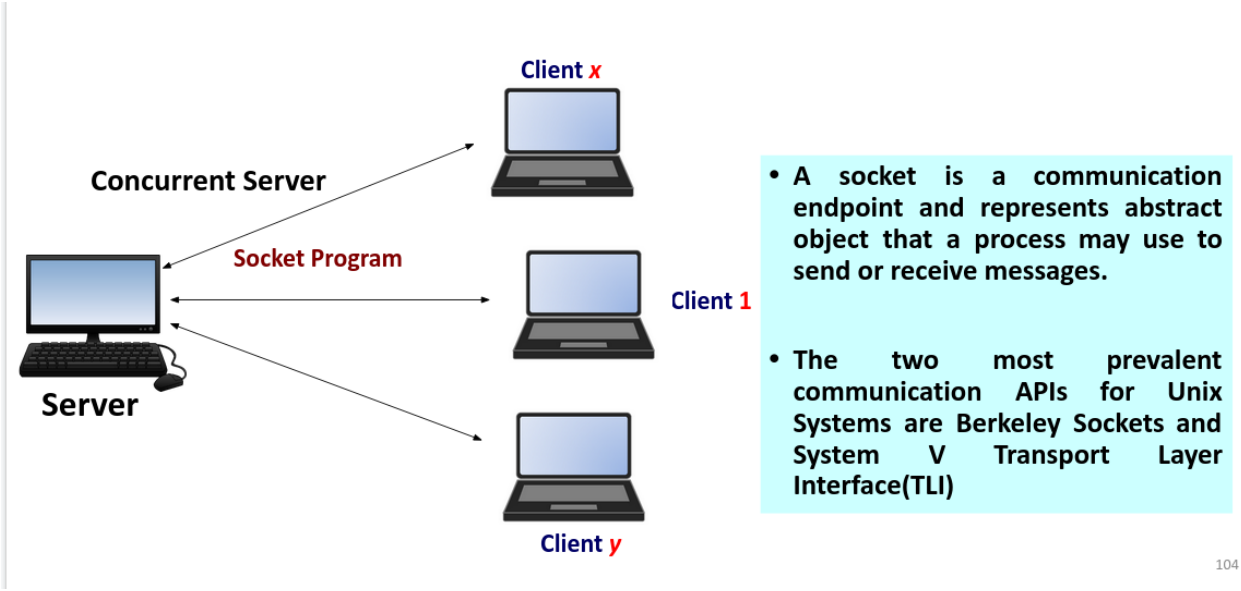
Network devices will be used for communication between two machines across a network.
It is also a device special file but not really because we use network interfaces instead of network devices.
You can check this interface using “ifconfig”.

Explanation for why Network interfaces are not device special files,
There are a few important differences between mounted disks and packet-delivery interfaces. To begin with, a disk exists as a special file in the /dev directory, whereas a network interface has no such entry point. The normal file operations (read, write, and so on) do not make sense when applied to network interfaces, so it is not possible to apply the Unix "everything is a file" approach to them. Thus, network interfaces exist in their own namespace and export a different set of operations.

Only the Application Layer of the OSI Model along with the Sockets are in the User space, everything else is in the Kernel space.

Note: Socket Address = IP Address + Port Address

Client-Server Model



There are two types of Servers,
Iterative Server
Concurrent Server

Iterative Server only allows one client to be connected with the server at a time. So if another client requests connection with the server while the first client is still connected, then it will have to wait till the first client terminates its connection.

Concurrent Server allows multiple clients to be connected with the server at a time.
How is this possible?
When a client establishes connection with the server, the server calls fork() and creates a child process. This child process will be responsible for handling that client and this method generalizes to n clients being connected with a single server at the same time.
You can also use multi-threading programming instead of the fork() call. So for each client a new thread will be created which will be responsible for handling it.

socket() System call

- The typical client -server relationship is not symmetrical
- Network Connection can be connection-oriented or connectionless
- More parameters must be specified for network connection, than for file I/O
- The Unix I/O system is stream oriented
- The network interface should support multiple communication protocol

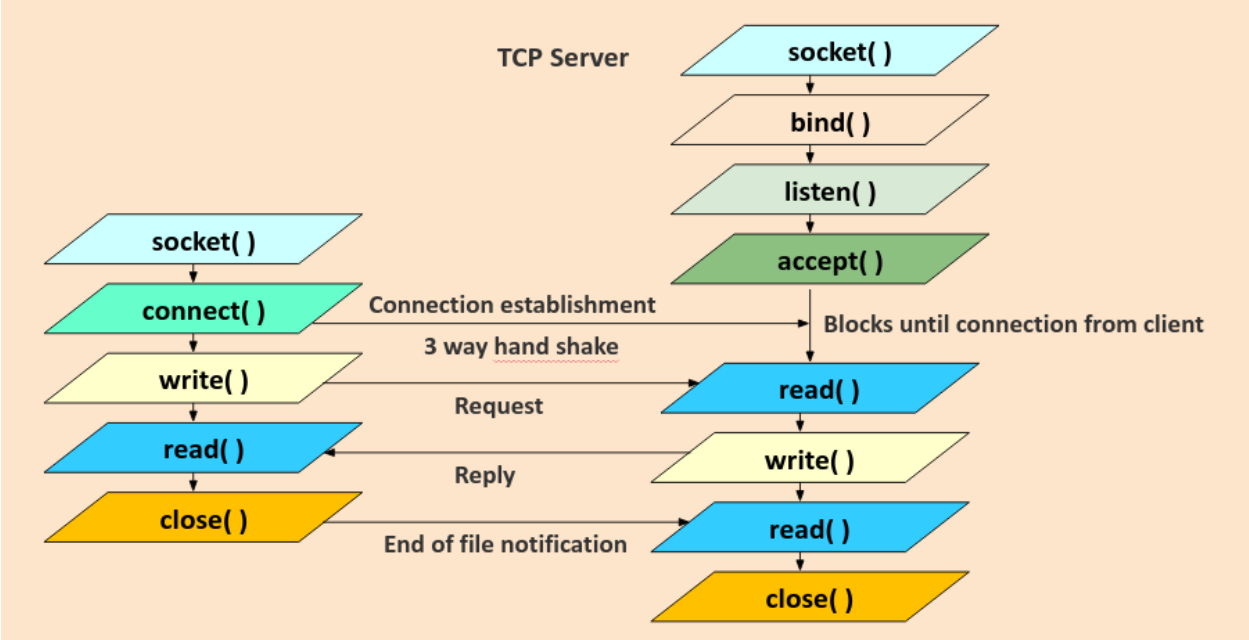
- int socket (int domain,int type,int protocol);
- Domain (AF – Address Family)
  - AF_UNIX for UNIX domain
  - AF_INET for Internet domain
- Socket type
  - SOCK_STREAM for TCP (Connection Oriented)
  - SOCK_DGRAM for UDP (Connectionless)
- Protocol
  - Protocol number is used to identify an application. List of the protocol number and the corresponding applications can be seen at /etc/protocols.
- The socket system call returns a socket descriptor on success and -1 for failure.

Client-Server relationship is not symmetrical => Cannot use client as server or vice-versa (because their implementations are different).

AF_UNIX = Unix Domain = Client and Server on the same machine
AF_INET = Internet Domain = Client and Server are not on the same machine
There's many other domains as well in the man page but it's not relevant for us right now.

Protocol can just be passed as 0. Check the man page.
If you're manually passing in this value, check that the port you have specified is not already being used by some standard service by checking "cat /etc/services".
Note: Protocol/Port Number/Port Address

Socket Functions



[Remember this diagram for exam]
The normal functioning of Client and Server sockets are shown here.

Create server and client socket using socket().
This returns a raw socket address which only specifies the socket domain and type and such.

Server binds to a port address using bind() and listens for client connections using listen().
bind() actually attaches the IP address to the socket.

When a client wants to connect to a server, it will use connect() which the server accepts using accept().
Connection is 3-way handshake, so it goes SYN, SYN + ACK, ACK.
When the client wants to request data from the server, it will use write() to send a message to the server.
Server reads this using read() and sends back the requested data/service by sending a message via write().
Client reads this using read().
This can repeat n number of times and finally the client closes the connection using close().
Server reads this close() and terminates the connection.
Eventually the server can itself close using the close() call.

Example program to demonstrate creation of iterative server.

Server-Side

```
int main() {
```

```
	struct sockaddr_in serv, cli;
	int sd, nsd;
	char buf[80];
	// sd = Socket Descriptor
	sd = socket(AF_UNIX, SOCK_STREAM, 0);

	serv.sin_family = AF_UNIX;
	// Specify destination address of socket
	// INADDR_ANY = Automatically assign IP address (of current machine)
	serv.sin_addr.s_addr = INADDR_ANY;
	// Meaning of htons() will be explained later
	serv.sin_port = htons(3558);

	// sd has become the server now
	bind(sd, (void*) (&serv), sizeof(serv));

	listen(sd, 5); // 5 = size of wait queue for connections
	sz = sizeof(cli);

	// Accept connection from client and put its details in cli
	// Server will wait at this line till a client connects with proper
IP address and port number
	// nsd = New Socket Descriptor
	nsd = accept(sd, (void*) (&cli), &sz);

	// Read data sent by cli (nsd) and store it in buf
	read(nsd, buf, sizeof(buf));
	printf("Message from client: %s\n", buf);
	write(nsd, "ACK from Server\n", 17);
}
```

Client-Side

```
int main() {
	struct sockaddr_in serv;
	int sd;
	char buf[80];
	sd = socket(AF_UNIX, SOCK_STREAM, 0);

	serv.sin_family = AF_UNIX;
	// Specify destination address of socket
	// INADDR_ANY = Automatically assign IP address (of current machine)
	serv.sin_addr.s_addr = INADDR_ANY;
```

```
	serv.sin_port = htons(3558); // Must be same port as specified in
server
	// Connect client socket sd to server
	connect(sd, (void*) (&serv), sizeof(serv));
	// Send message to server
	write(sd, "Hello Server\n", 17);
	read(sd, buf, sizeof(buf));
	printf("Message from server: %s\n", buf);
}
```

More details below.

05/10/21

Run these programs using sudo privileges.
Also if the programs don't seem to work then try changing the port number, it is possible that they are still in use, maybe by the previous run of the same program itself!

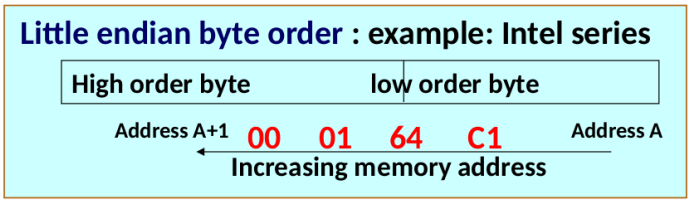
For the project the server will run in the background in an infinite loop and not display anything as such.

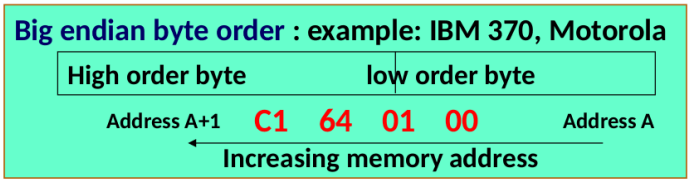
sock Structure

- **struct sockaddr_in {**
  short int sin_family;
  unsigned short int sin_port;
  struct in_addr sin_addr;
  }
- **sin_family - address family**
- **sin_port - port number**
- **sin_addr - internet address (IP addr)**

- **The in_addr structure used to define sin_addr is as under**

```
struct in_addr {
    unsigned long s_addr;
/* refers to the four byte IP address */
 }
```





Byte ordering ex: 91,329 hex: 00 01 64 C1

107

s_addr = INADDR_ANY in the case where we want IP address on the same machine
However in the general case we'll have to specify it as,
s_addr = inet_addr("IP Address")

The inet_addr() function converts the Internet host address cp from IPv4 numbers-and-dots notation into binary data in network byte order.

htons(): Little Endian vs Big Endian Byte Ordering

They are both concerned with byte orderings in hexadecimal notation.
For example, 91329 in decimal = 00 01 64 C1 in hex
Now how to store this in memory?
Little Endian stores bytes in the correct order (LSB -> MSB), so higher order bytes are given to higher memory address values.
Big Endian stores it in the reverse order (MSB -> LSB), so lower order bytes are given to higher address values.
Note: All this is for integer datatype only, not float or anything like that.

IP addresses and port numbers use Big Endian byte ordering. So we don't have to change anything in the case of Big Endian ordering whereas we'll have to convert Little Endian to Big Endian (which we'll have to because we're using Intel systems and not IBM/Motorola).

- **Internet protocols use big endian byte ordering called network byte order**
- **The following functions allow conversions between the formats.**

#include <netinet/in.h>

htons() – "Host to Network Short"

htonl() – "Host to Network Long"

ntohs() – "Network to Host Short"

ntohl() – "Network to Host Long"

- **h stands for host n stands for network**
- **s stands for short l stands for long**

- **int bind (int sockfd, struct sockaddr *my_addr,int addrlen);**
- **sockfd - the socket file descriptor returned by socket().**
- **my_addr - a pointer to a struct sockaddr that contains information about IP address and port number.**
- **addrlen - set to sizeof (struct sockaddr)**

For example, htons(5996);
The integer that we give in is stored in the ordering specified by the system. We use Intel systems and so its Little Endian. But because networks use Big Endian (Network byte ordering), we convert 5996 to Big Endian ordering using htons().

Of these functions we'll only require htons().

connect() and listen()

int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);

- **sockfd - the socket file descriptor returned by socket().**
- **serv_addr - is a struct sockaddr containing the destination port and IP address.**
- **addrlen - set to sizeof (struct sockaddr).**

int listen (int sockfd,int backlog);

- **sockfd - the socket file descriptor returned by socket().**
- **backlog - the number of connections allowed on the incoming queue.**
- **Backlog should never be zero as servers always expect connection from client.**
- **The listen function converts an unconnected socket into a passive socket,**
- **On successful execution of listen is indicating that the kernel should accept incoming connection requests directed to this socket.**

109

accept() and close()

- **int accept (int sockfd, void *addr, int *addrlen);**
- **sockfd**
  - **the socket file descriptor returned by socket().**
- **addr**
  - **a pointer to a struct sockaddr_in. The information about the incoming connection like IP address and port number are stored.**
- **addrlen**
  - **a local integer variable that should be set to sizeof (struct sockaddr_in) before its address is passed to accept().**

- **Socket descriptor can be closed like file descriptor.**

close (sockfd);

- **Close system call prevents any more reads and writes to the socket. For attempting to read or write the socket on the remote end will receive an error.**

110

Remember that sockets are like pipes only.
One end of the pipe is at the client and the other is at the server.
Server end of the socket is always open once listen() is called.
Client client end of the socket is open once connect() is called successfully.
So after connect() is called, the socket descriptor for the client is pointing to the server and the socket descriptor for the server is pointing to the client.
That's why we can directly execute read() and write() calls using socket descriptor "sd".

shutdown()

- **int shutdown (int sockfd, int how);**

- **sockfd - socket file descriptor of the socket to be shutdown.**
- **how – if it is**
  - **0 - Further receives are disallowed**
  - **1 - Further sends are disallowed**
  - **2 - Further sends and receives are disallowed.**

- **The shutdown system call gives more control (than close (sockfd) over how the socket descriptor can be closed.**

Iterative vs Concurrent Server

- **One client request at a time.**

```
nsd = accept (sd, &cli,...);
while (1) {
  read/write(nsd, ...);
}
```

Many clients requests can be serviced concurrently

```
while (1) {
  nsd =(accept (sd, &cli, ....);
   if (!fork( )) {
        close(sd);
        read/write(nsd, .....);
        exit();
   } else
       close(nsd);
}
```

In the iterative server, the server can accept one connection at a time and read/write with that client.
In the concurrent server, the server can accept more than one connection at a time.
Once you accept you receive a new socket descriptor (nsd) pointing to the corresponding client.
So the server forks a child and this child is responsible for communicating with the client using “nsd”, it doesn’t use the raw socket descriptor “sd” for communication, that’s why sd is closed.
Once the child does all the reads and writes, the child is terminated along with the connection to the client.
The parent only uses the raw socket descriptor to accept new connections and has no usage of “nsd”, that’s why “nsd” is closed for it.

So the parent handles accepting the connections and for each new connection it creates a child process that is responsible for actually communicating with that client.
As mentioned before, instead of forking a child we can also create a new thread for it which also can be implemented using the same logic.
The only difference will be that the thread will be in the same memory space as the process whereas a forked child occupies a different memory space from its parent.

Exam Q. You have a concurrent server and you want to restrict the number of client connections to 100. How will you implement this?
Clearly counting semaphore is required but what will the general implementation be like?

Persistence Level of IPC Mechanisms

Unnamed Pipe => Process Level => Pipe terminates when the process terminates.

Named Pipe (FIFO) => File System Level => FIFO only terminates when the file system is detached or the FIFO file is explicitly deleted.

Message Queue, Shared Memory, Semaphores => Kernel Level => Created in the kernel, so they terminate on reboot or when they are explicitly deleted.
On reboot, the original kernel is loaded again which clears whatever modifications were made to it in the last session which includes these IPC mechanisms.

Signals

Check the list of available signals using “kill -l”.
1 to 32 => Standard Signals
33 onwards => Real-time Signals => Starts with SIGRT
Check what each signal does using “man 7 signal”.

Whenever a signal is sent, some function has to be executed which is called the signal handler.
Each standard signal has a default signal handler associated with it.
The real-time signals don’t have a default signal handler.

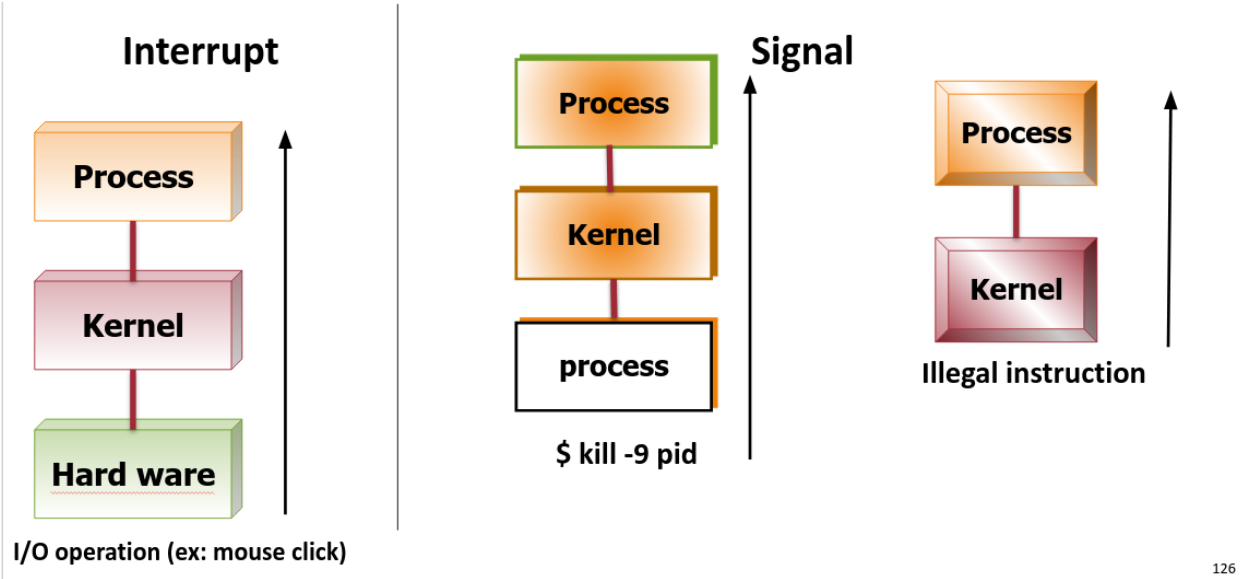
- Signals are a fundamental method for inter process communication and are used in everything from network servers to media players.
- A signal is generated when
  - an event occurs (timer expires, alarm, etc.,)
  - a user quota exceeds (file size, no of processes etc.,)
  - an I/O device is ready
  - encountering an illegal instruction
  - a terminal interrupt like Ctrl-C or Ctrl-Z.
  - some other process send (kill -9 pid)

- Each signal starts with macro SIGxxx.
- Each signal may also specifies with its integer number
- For help: $ kill –l , $ man 7 signal
- When a signal is sent to a process, kernel stops the execution and "forces" it to call the signal handler.
- When a process executes a signal handler, if some other signal arrives the new signal is blocked until the handler returns.

125

- User quota is typically set by the system admin and there are two types of limits: soft and hard limit. Soft limit is kind of like a warning whereas exceeding hard limit results in direct stoppage.
- Illegal instructions refer to stuff like divide by zero, segmentation fault, etc.
- Ctrl-C = SIGINT, Ctrl+Z = SIGSTOP
- If the process is in an uninterruptible sleep state while a signal arrives, then that signal will wait till the process enters some other state for it to be received and signal handler to be executed.

Signal vs Interrupt



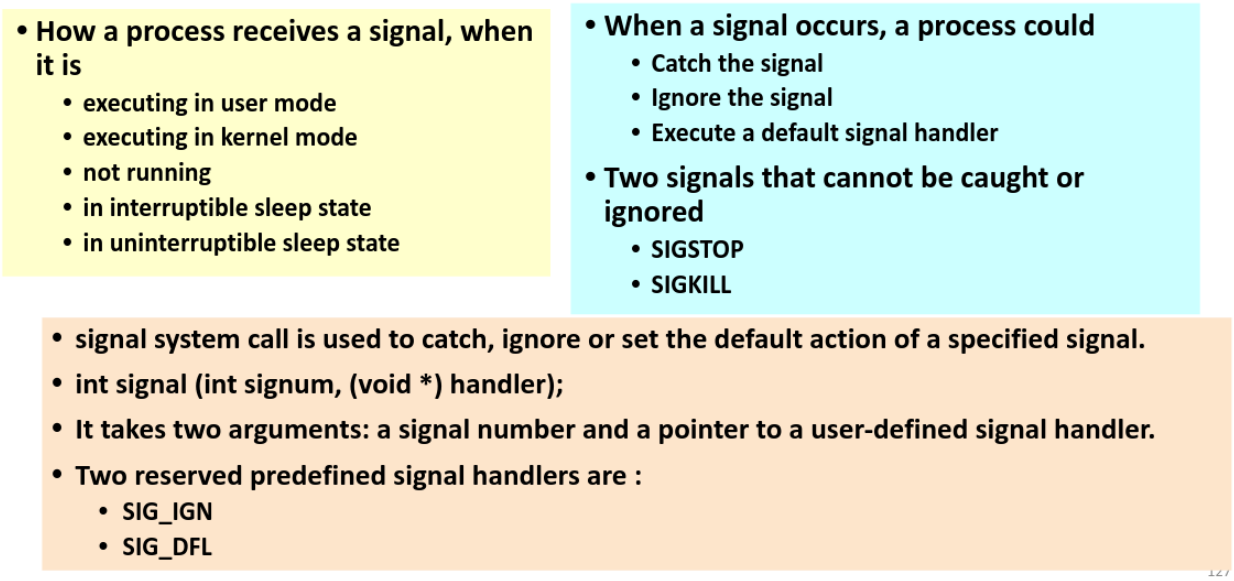
126

Here we can see that Interrupts are typically generated at the hardware side which informs the kernel and starts the ISR (Interrupt Service Routine) which is a process.
[Note: Trap is just a software interrupt]

A signal is generated by a process. The signal informs the kernel after which the corresponding signal handler is executed which is another process.

So you can also see how signal handling is done by the Kernel (because it involves system calls).

signal() System Call



When the process is in kernel mode, the signal has to wait till it enters user mode because the process can modify the kernel data structure in kernel mode which is global and allowing signal to enter can result in race conditions.
When the process is in the uninterruptible sleep state, the signal has to wait till the process gets out of that state.
In all other cases the signal is immediately received by the process.

Catching the signal is just catching an exception in typical programming languages. We catch a specified signal and execute our own function after it instead of executing the signal handler associated with that signal.

SIGSTOP suspends the process which can be resumed later using SIGCNT.
SIGKILL kills the process (kill -9 pid).

SIGKILL and SIGINT both kill the process however SIGINT can be ignored which is why SIGKILL is called the sure-kill signal.

SIG_IGN = Predefined signal handler to ignore a signal.

SIG_DFL = Predefined signal handler to give the default signal handler for the specified signal indicated by signum parameter in signal() call.

sigaction() & kill() System Calls

- **sigaction ( ) is same as signal( ) but it has lot of control over a given signal.**
- **The syntax of sigaction is:**

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

  - **signum, is a specified signal**
  - **act is used to set the new action of the signal signum;**
  - **oldact is used to store the previous action, usually NULL.**

- **Kill system call is used to send a given signal to a specific process**
- **int kill (pid_t process_id, int signal_number);**
- **it accepts two arguments, process ID and signal number**
- **If the pid is positive, the signal is sent to a particular process.**
- **If the pid is negative, the signal is sent to the process whose group ID matches the absolute value of pid.**

128

Do note that this kill() isn't just for killing the process, it is for sending any signal to the process which can include SIGKILL which is what actually kills a process.
Like how "kill -9 pid" is required for killing a process.

Example Program to demonstrate Signals

The SIGINT signal is used here which is **sent to a process by its controlling terminal when a user wishes to interrupt the process**. This is typically initiated by pressing Ctrl + C, but on some systems, the "delete" character or "break" key can be used. Interrupting just refers to terminating the process.

```
// Function prototype for signal handler, it will be defined after main()
void my_handler(int sig);

int main(void) {
      // Catching a signal
      signal(SIGINT, my_handler);
      printf("Catching SIGINT\n");
      sleep(3);
      printf("No SIGINT within 3 seconds\n");

      // Ignoring a signal
      signal(SIGINT, SIG_IGN):
```

```
	printf("Ignoring SIGINT\n");
	sleep(3);
	printf("No SIGINT within 3 seconds\n");

	// Calling default action for a signal
	signal(SIGINT, SIG_DFL);
	printf("Default action for SIGINT\n");
	sleep(3);
	printf("No SIGINT within 3 seconds\n");

	return 0;
}
// Custom signal handler
void my_handler(int sig) {
	printf("I got SIGINT number %d\n", sig);
	exit(0);
}
```

In this program, on execution we have 3 scenarios executing depending on when we press “Ctrl + C” => send SIGINT signal.
If we send in the first 3 seconds of execution, then we catch the signal and print the SIGINT number and terminate the process. Number for the SIGINT signal is 2, so that is what will be printed.
If we send in time 3-6 seconds of execution, then we ignore the signal, so it just proceeds with execution.
If we send in time 6-9 seconds of execution, then we call the default signal handler which for SIGINT just terminates the process.
After this program will terminate automatically because it will have reached return 0.

Note: The handler function must end with exit() call otherwise it will just execute the signal handler and control just stays there. In the case of the SIGSEGV handler this caused an infinite loop as well.

Alarm and Timers

- **unsigned int alarm (unsigned int seconds);**
- **It is used to set an alarm for delivering SIGALARM signal.**
- **On success it returns zero.**

- **Three interval timers.**
  - **ITIMER_REAL**
    - This timer counts down in real (i.e., wall clock) time. At each expiration, a ***SIGALRM*** signal is generated.
  - **ITIMER_VIRTUAL**
    - This timer counts down against the user-mode CPU time consumed by the process. (The measurement includes CPU time consumed by all threads in the process.) At each expiration, a ***SIGVTALRM*** signal is generated.
  - **ITIMER_PROF**
    - This timer counts down against the total (i.e., both user and system) CPU time consumed by the process. At each expiration, a ***SIGPROF*** signal is generated.

The interval timers are used to determine how the time given to an alarm is to be interpreted, i.e. what time should be counted against the alarm.

ITIMER_REAL measures real-world time.
ITIMER_VIRTUAL only measures time taken to execute user-mode functions (no system calls) but doesn't include time taken for waiting.
ITIMER_PROF measures time taken to execute user-mode functions as well as kernel-mode functions (system calls) but doesn't include time taken for waiting for I/O.

In summary,
ITIMER_REAL = User-Mode (CPU + I/O) + Kernel-Mode time => SIGALRM
ITIMER_VIRTUAL = User-Mode (CPU) time => SIGVTALRM
ITIMER_PROF = User-Mode (CPU) + Kernel-Mode (CPU) time => SIGPROF

get and set timer

- get value of an interval timer
- int *getitimer* (int which, struct itimerval *val);
- On success it returns zero and the timer value is stored in the itimverval structure.
- Example: ret = getitimer (ITIMER_REAL, val);

- Set value for a interval timer
- int *setitimer* (int interval_timers, const struct itimerval *val, struct itimvferval *old_value);
- On success it returns zero.
- Example: ret = setitimer(ITIMER_REAL, &value, 0);

115

getitimer is used to return how much time is left for the timer to go off.

Resource Limits

- The OS imposes limits for certain system resources it can use.
- Applicable to a specific process.
- The “ulimit” shell built-in can be used to set/query the status.
- “ulimit –a” returns the user limit values

```
[root@localhost ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7892
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7892
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
[root@localhost ~]#
```

117

It is possible that depending on which user you are logged in as, you will get different outputs for “ulimit -a”. Like root vs regular user.

What is the core file mentioned in the output of “ulimit -a”?
It contains the core dump file which contains error logs. Only created for segmentation faults.
If core file size = 0 then that means core dump will not happen.
So that means we have to set it to something larger in order to have core dumps to happen.

08/10/21

Hard and Soft Limits

-c Maximum size of "core" files created.

-f Maximum size of the files created.

-l Maximum amount of memory that can be locked using mlock() system call.

-n Maximum number of open file descriptors.

-s Maximum stack size allowed per process.

-u Maximum number of processes available to a single user.

- Each resource has two limits –Hard and Soft
- Hard Limits
  - Absolute limit for a particular resource. It can be a fixed value or "unlimited"
  - Only superuser can set hard limit.
- "ulimit" command has –H or –S option to set hard/soft limits. Default is soft limit.
- Hard limit cannot be increased once it is set.

- Soft Limits
  - User-definable parameter for a particular resource.
  - Can have a value of 0 till <hard limit> value.
  - Any user can set soft limit.
- Limits are inherited (the new values are applicable to the descendent processes).

getrlimit() and setrlimit() System Calls

- getrlimit()/setrlimit() are system-call interfaces for getting and setting resource limits.
- Syntax
  - getrlimit(<resource>, &r)
  - setrlimit (<resource>, &r)
  - where r is of type "struct rlimit"

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
int prlimit(pid_t pid, int resource, const struct rlimit *new_limit, struct rlimit *old_limit);

struct rlimit {
        rlim_t rlim_cur;  /* Soft limit */
        rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
      };
```

Resource Usage: rusage struct

```
struct rusage {
        struct timeval ru_utime; /* user CPU time used */
        struct timeval ru_stime; /* system CPU time used */
        long   ru_maxrss;      /* maximum resident set size */
        long   ru_ixrss;       /* integral shared memory size */
        long   ru_idrss;       /* integral unshared data size */
        long   ru_isrss;       /* integral unshared stack size */
        long   ru_minflt;      /* page reclaims (soft page faults) */
        long   ru_majflt;      /* page faults (hard page faults) */
        long   ru_nswap;       /* swaps */
        long   ru_inblock;     /* block input operations */
        long   ru_oublock;     /* block output operations */
        long   ru_msgsnd;      /* IPC messages sent */
        long   ru_msgrcv;      /* IPC messages received */
        long   ru_nsignals;    /* signals received */
        long   ru_nvcsw;       /* voluntary context switches */
        long   ru_nivcsw;      /* involuntary context switches */
    };
```

```
int getrusage(int who, struct rusage *usage);
getrusage()  returns  resource usage
measures
for who, which can be one of the following:
RUSAGE_SELF
RUSAGE_CHILDREN
RUSAGE_THREAD
```

```
sysconf - get configuration information at
run time:

long sysconf(int name);

Example: ret= sysconf(_SC_CLK_TCK);

On success it returns the value of the given
system limits.
```

To know what you can give as input to sysconf() have a look at its man page.

Signals Continued: SIGSEGV

Consider the following program,

```c
int main() {
      int a;
      scanf("%d", a);
}
```

Does this program give an error?
Yes, Segmentation Fault (core dumped).
To get the core dump file (executable file) you have to set core file size to something greater than zero atleast using ulimit command.
We can debug the original program by using the generated core file with GDB.

When a segmentation fault occurs, a SIGSEGV (Segmentation Violation) signal is raised.

Note that we can also catch this signal using the signalling mechanism studied earlier.

Multithreading

Thread is a sequential flow of control through a program.
If a process is defined as a program in execution then a thread is defined as a function in execution.
If a thread is created, it will execute a specified function.
Two type of threading: 1. Single Threading and 2. Multi threading

The created threads within a process share

1. **instructions of a process**
2. **process address space and data**
3. **open file descriptors**
4. **Signal Handlers**
5. **pwd, uid and gid**

The created threads maintain it's own

1. **thread identification number (tid);**
2. **pc, sp, set of registers**
3. **stack**
4. **priority of the threads**
5. **scheduling policy**

Advantages of Threads:
Takes less time for creation of a new thread, termination of a thread and communication between threads are easier.

121

[Single Thread => Just don't create any threads]

Communication between threads is easier than that of a process because threads of the same process share the same memory. So no inter-process communication involved.

The disadvantage of threads is that managing concurrency is difficult. For this we have to synchronize the threads for which we'll require POSIX Semaphores (not System-V Semaphores) which allows us to create unnamed semaphores which can be used by related processes like unnamed pipe.

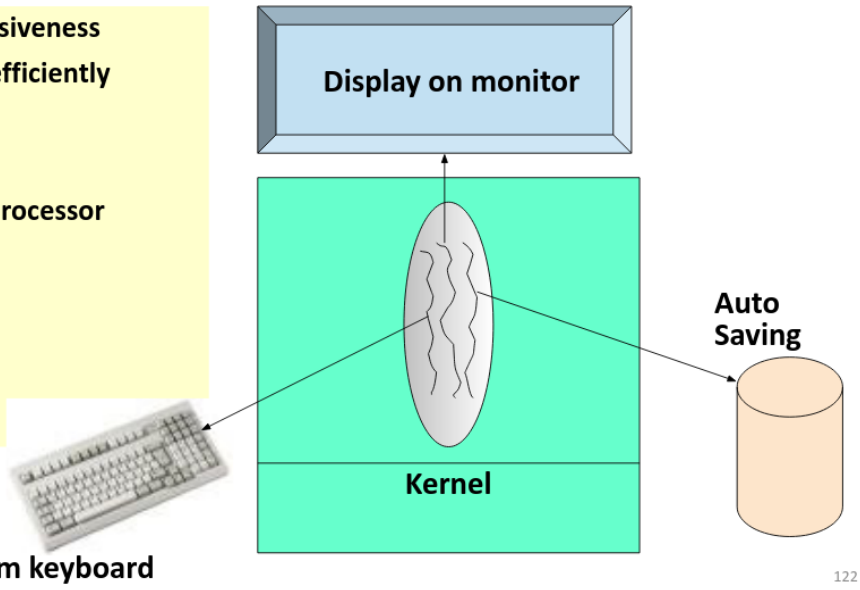
When we create threads, we can also specify the initial values for the parameters mentioned in the slide that are specific to each thread.
If we don't specify them, then the threads will inherit such values from the process that created them.

Advantages of Threads

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- use fewer system resources
- Specific applications in uniprocessor machines

Applications

- A file server on a LAN
- GUI
- web applications



It uses multiprocessors more efficiently because if we had multicore systems with no threading then it is possible that only one core will be in use at a time.
Improves program structure because it forces the code to be more modularized.
Specific applications in uniprocessor machines => Concurrent server

Note that the kernel treats each thread as a separate process; the only thing is that they will share the same address space and all the other properties mentioned before.

Thread Creation

```
#include <pthread.h>
void thread_func(void) {
    printf(" Thread id is %d", pthread_self());
}
main ( ) {
    pthread_t mythread;
    pthread_create ( &mythread, NULL, (void *) thread_func, NULL);
 }
```

This needs to be compiled as follows...$gcc pthread.c –lpthread

pthread_t is type-defined as unsigned long int. It takes the thread address as the first argument, the second argument is used to set the attributes for the thread-like stack size, scheduling policy, priority; if NULL is specified, then it takes default values for the attributes.

The third argument is the function that the thread should execute when created. The fourth argument is the argument for the thread function. If that function has a single argument to be passed, we can specify it here. If it has more than one argument, then we have to use a structure and declare all the arguments and pass the address of the structure.

pthread => p stands for POSIX
All of these are POSIX library functions and not system calls.

When you use the math library in a C program and try to compile it normally then most likely a warning message will be thrown. You resolve this by including a flag “-lm” while compiling to indicate that the math library is used. Similarly, when a pthread library is used, the “-lpthread” flag must be specified while compiling.
[lm = Link to math library, lpthread = Link to pthread library]

Sample Code:

```c
void *myThreadFunction(void *argvp) {
      printf("Printing HELLO WORLD from Thread\n");
      return NULL;
}
int main() {
      pthread_t tid;
      pthread_create(&tid, NULL, myThreadFunction, NULL);
      pthread_join(tid, NULL);
      exit(0);
}
```

The third argument of pthread_create() takes in the function name as the argument because the function name by itself is a pointer (like how an array is a pointer to the first element in the

array). Also note that the only argument given to the function has to be void*, even if you are not passing in any argument.
The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.
This function is useful in case there are some dependencies between the threads being created. So you can ensure that the threads on which the new thread is dependent on finish execution before it starts its own execution.
In this program, pthread_join() ensures that the thread terminates before main() itself terminates.

Summary:
pthread_create for threads => fork() for processes
pthread_join for threads => waitpid() for processes

If you use strace on the pthread program you can see that it uses the clone() system call in the backend which is similar to fork().

Memory Management

[Not much in terms of programming like system calls or library functions but there's a lot in terms of kernel programming => Not relevant for us right now atleast.
So just note the theory from this section]

Swap space is **a partition on the hard disk that is a substitute for physical memory**. It is used as **virtual memory** which contains process memory images.
We determine the size of the swap space and optimally it should have been twice the size of the RAM. Nowadays because of how large RAMs have become, swap space should be half the size of the RAM.

According to the kernel, **RAM + Swap Space = Actual RAM space**

Places to check memory size (including swap),
cat /proc/meminfo
free -m
vmstat 1 1 [1 to return just 1 entry, otherwise it will go on infinitely]
sudo slabtop [like top command but for memory]

Virtual Memory

- **Memory management: one of the most important kernel subsystems**
- **Virtual Memory: Programmer need not to worry about the size of RAM (Large address space)**
- **Static allocation: internal Fragmentation**
- **Dynamic allocation: External Fragmentation**
- **Avoid Fragmentation: Thrashing -overhead**

- **Large address space: virtual memory is many times larger than the physical memory in a system.**
- **For a 32 bit OS, the virtual memory size will be 2 to the power of 32 i.e 4GB. But the RAM size may be much smaller.**
- **Each process has a separate virtual address space.**
- **Each process space is protected from other processes.**
- **It supports shared virtual memory, i.e more than one process can share a shared page.**
- **Uses paging technique.**

Programmers need not worry about the size of RAM

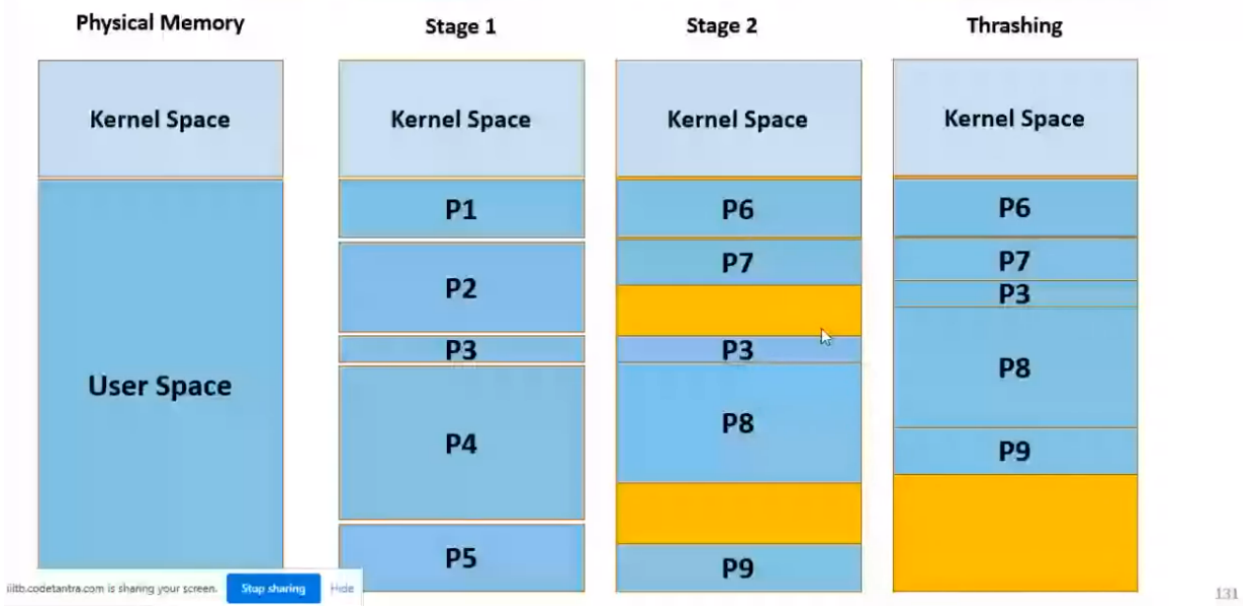
- Because the concept of demand paging is used. The process is split into pages of disk block size (4KiB typically) and pages will be brought into main memory only when they are required (demanded), i.e. a page fault happens for that page.

Internal Fragmentation happens with static allocation

- Suppose page size = 4KiB and your program size is 6KiB, then your program shall occupy 2 pages, one fully filled and one in which 2KiB is free.
- This 2KiB is wasted space and this occurrence is called internal fragmentation as wastage happens within a page/block.

However with small page sizes like 4KiB even internal fragmentation is minimal and this is a satisfactory enough solution.

Dynamic Allocation



In this example, no page sizes exist. However once P1 occupies that space, a page of that size has been created and once P1 leaves, new pages that are stored in that page frame must have the same size or less than it.

Little by little this will create small sized empty slots that will remain unused. And that's why this is called external fragmentation.

The solution to this is that in periodic intervals of time, a function must execute that pushes all of the externally fragmented blocks at the end of the user space. After this the entire fragmented space can be used as a single block again which makes it much less susceptible to being externally fragmented.

Page Table

- Hardware support (MMU, TLB) is required.
- Fair share allocation
- static allocation
- Minimize internal fragmentation
- identified by a PFN (Page Frame Number)
- virtual address is split into two parts namely an offset and a virtual page frame number.

- Translate a process virtual address into physical address since processor use only virtual address space
- The size of a page table is normally size of a page
- if it is 4kb, each page address size is 4byte, so 1024 page entries in a page table.
- holds info about
  - Whether valid page table or not?
  - PFN
  - access control information
- Stored in TLB (Translation Look-aside Buffer)

Fair-share allocation => Static allocation and it minimizes internal fragmentation because only the last page of a process will cause small amounts of internal fragmentation, all of the other pages will fully occupy their respective page frames.

You can link this page allocation and address translation concept to the single-indirect, double-indirect, etc. addressing discussed on the ext file system.

Memory Mapping

- Executable image split into equal sized small parts (normally the page size)
- Virtual memory assigns virtual address to each part
- Linking of an executable image into a process virtual memory

Swapping

- Swap space is in hard disk partition
- If a page is waiting for certain event to occur, swap it.
- Use physical memory space efficiently
- If there is no space in Physical memory, swap LRU pages into swap space

- Demand Paging - don't load all the pages of a process into memory
- Load only necessary pages initially
- if a required page is not found, generate page fault then the page fault handler brings the corresponding page into memory.

Kernel Data Structure

[Not that important right now]

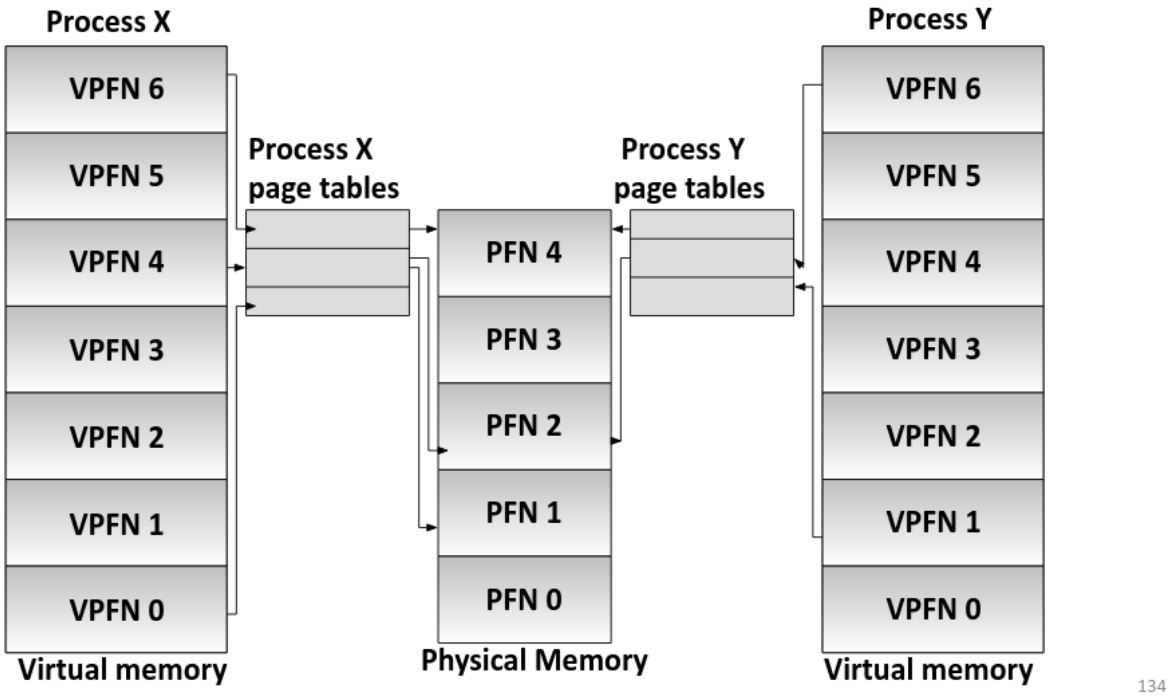
- **Source Code: /usr/src/linux-4.12/mm**
  /usr/src/linux-4.12/include/linux
- **virtual memory is represented by an mm_struct data structure**
- **it has pointers to vm_area_struct data structure**
  - **created when an executable image is mapped with the process virtual address**
  - **has starting and end points of virtual memory**
  - **represents a process's image like text, data and stack portion**
  - **has control access info**

mm_types.h

```
struct mm_struct {
    struct vm_area_struct *mmap;        /* list of VMAs */
    ……
```

```
struct vm_area_struct {
    /* The first cache line has the info for VMA tree walking. */
    unsigned long vm_start;
    unsigned long vm_end;
    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;
    ……
```

Virtual to Physical Memory Translation

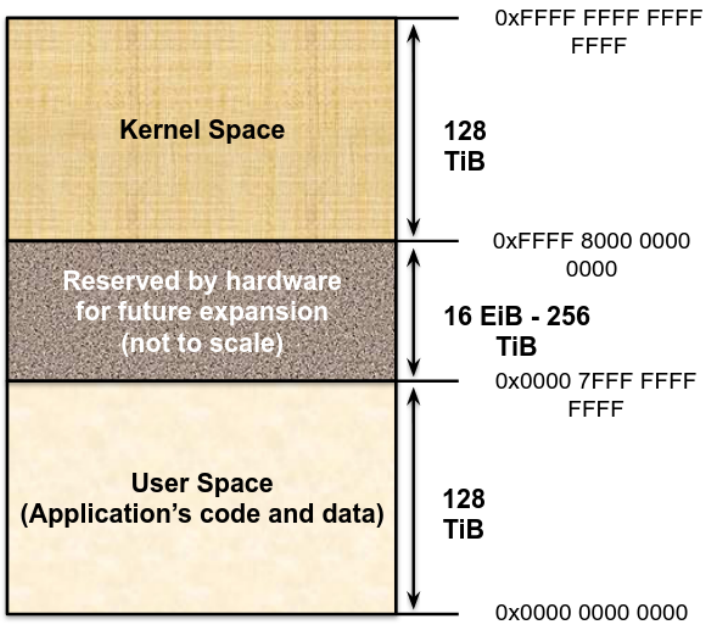


12/10/21

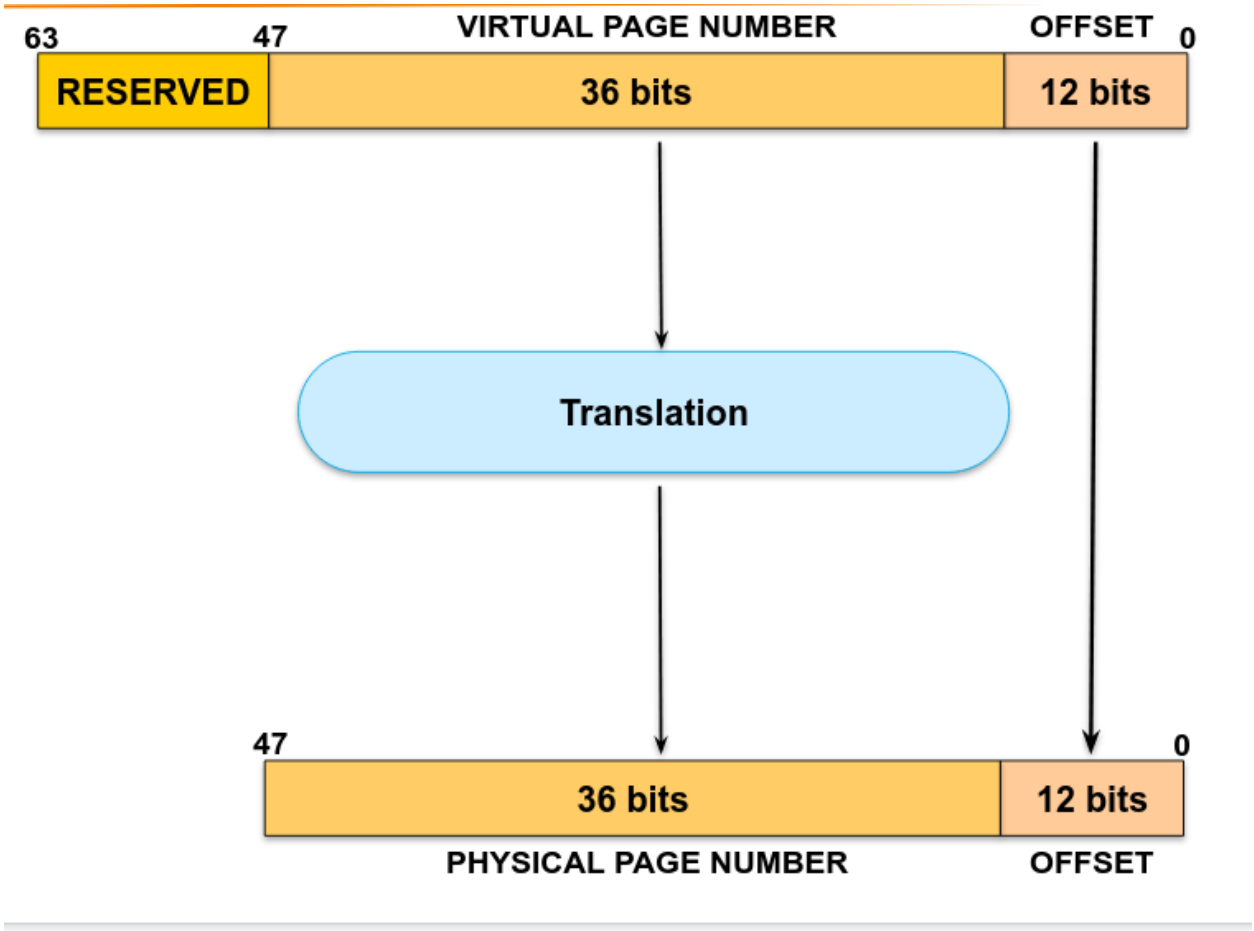
x86-64 Virtual Memory Layout

$pmap –x <pid>

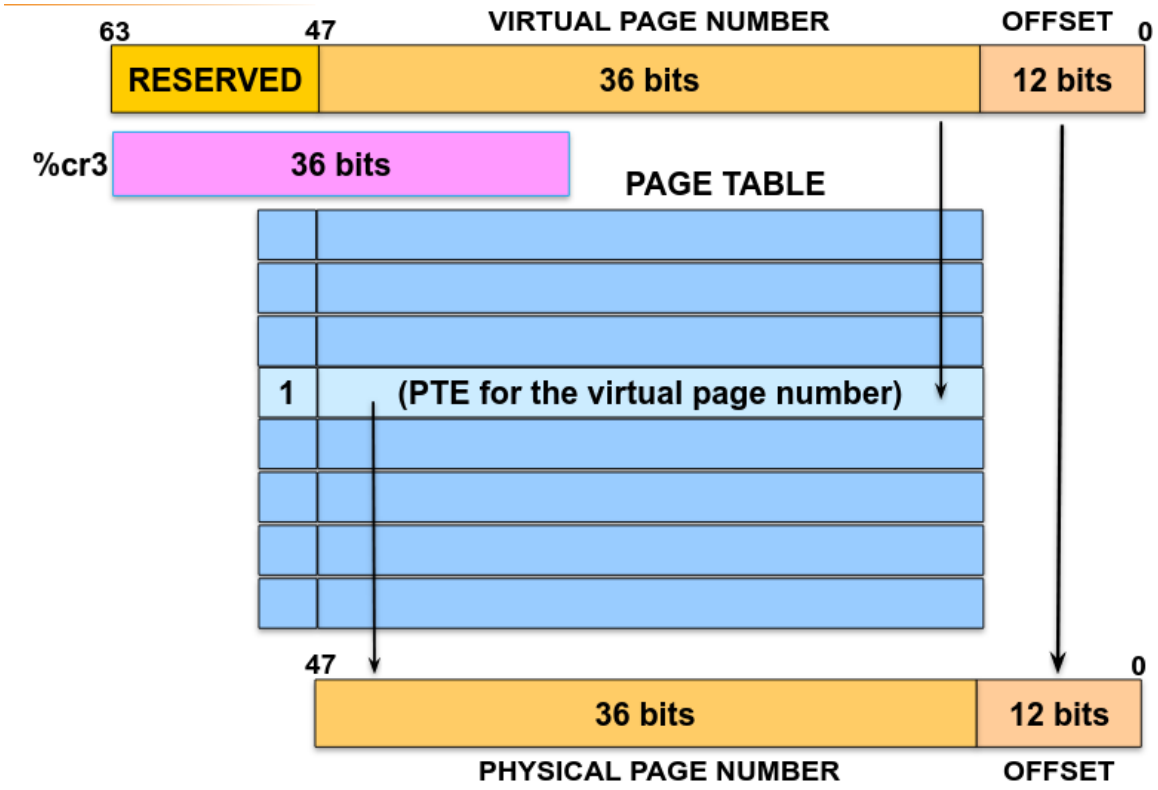
Prefix	Name			
Kibi	binary kilo	1 kibibyte (KiB)	2^{10} bytes	1024 B
Mebi	binary mega	1 Mebibyte (MiB)	2^{20} bytes	1024 KiB
Gibi	binary giga	1 Gibibyte (GiB)	2^{30} bytes	1024 MiB
Tebi	binary tera	1 Tebibyte (TiB)	2^{40} bytes	1024 GiB
Pebi	binary peta	1 Pebibyte (PiB)	2^{50} bytes	1024 TiB
Exbi	binary exa	1 Exbibyte (EiB)	2^{60} bytes	1024 PiB



Virtual to Physical Address Translation

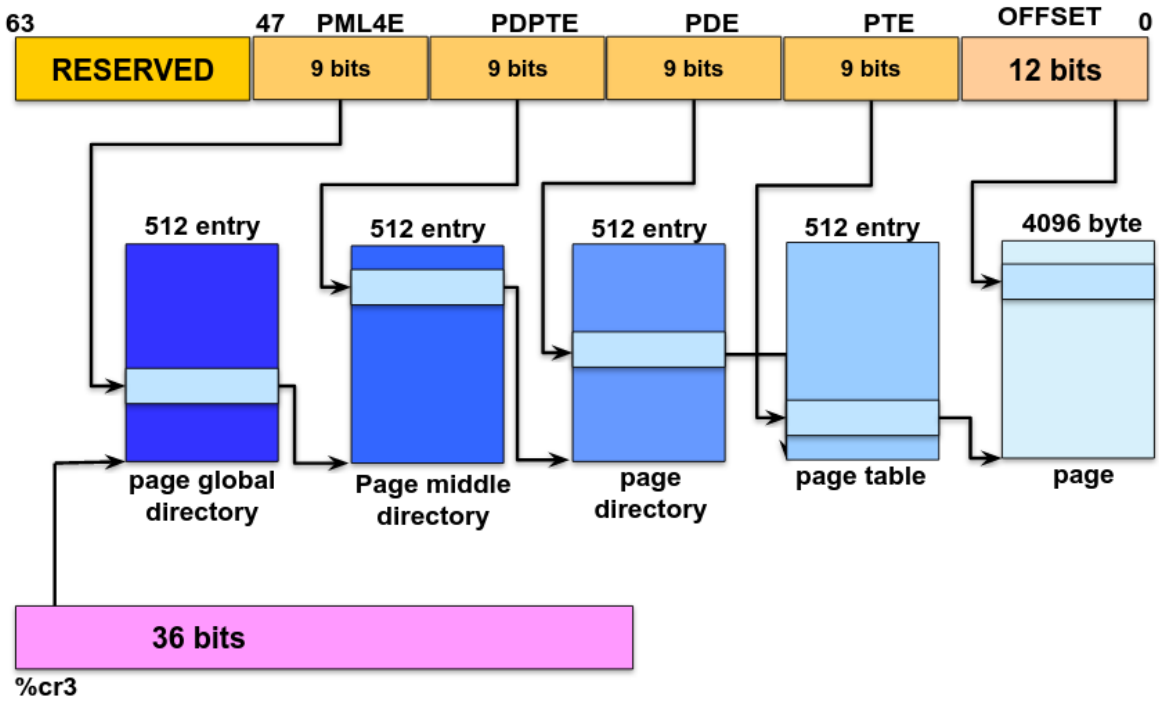


More details on what is happening here is given in the following slide,



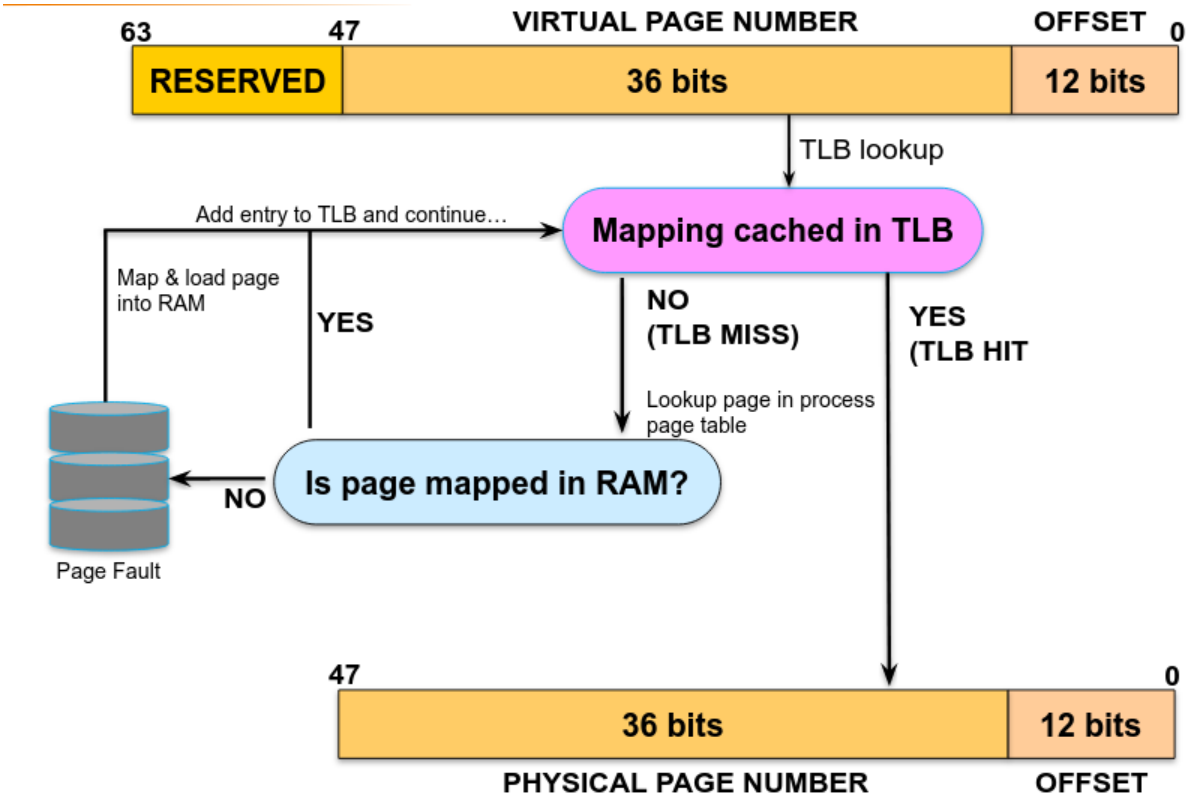
%cr3 = Translation Look-aside Buffer (TLB)
So Virtual Page Number points to the Page Table entry (PTE) which contains the Physical Page Number.
The Page Table is stored in the TLB and is responsible for translation from virtual to physical page number.

Page Table Hierarchy on x86-64



This shows you how multi-level indexing works in Page Table address translation.
Again, this is quite similar to the ext4 file system with its double and triple indirect addressing format.

Paging & TLB



Mapping cached in TLB = Page Table
“Is Page Mapped to RAM” is asking whether the page is present in swap space or not.
This happens when a page remains inactive for some time. It is moved from main memory to swap space.
If the page is not even present in swap space then page fault occurs and by principle of demand paging it is brought into main memory and an entry for it is made in the page table which is present in TLB.

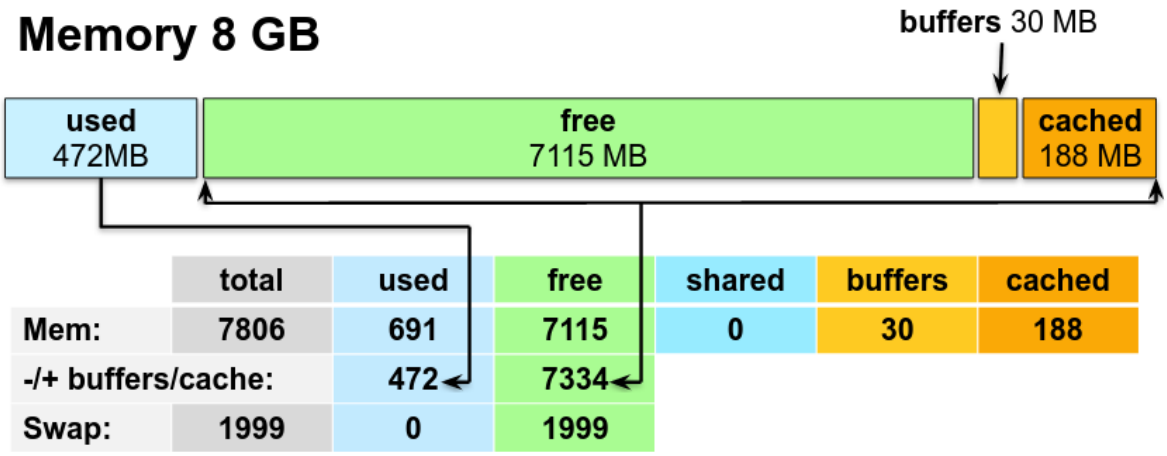
Major & Minor Page Faults

The page fault where page entry is not present in TLB but is present in swap space is called minor page fault.
While the page fault where a page is not even present in swap space but instead has to be fetched from the hard disk is called the major page fault.

Sidenote: SAR - System Activity Report

Prints system logs about all the activity that has happened since boot.
This is really in depth and gives a huge output file.
This is useful because this can be given to system monitoring programs as input and then we can monitor our system with a higher degree of control.

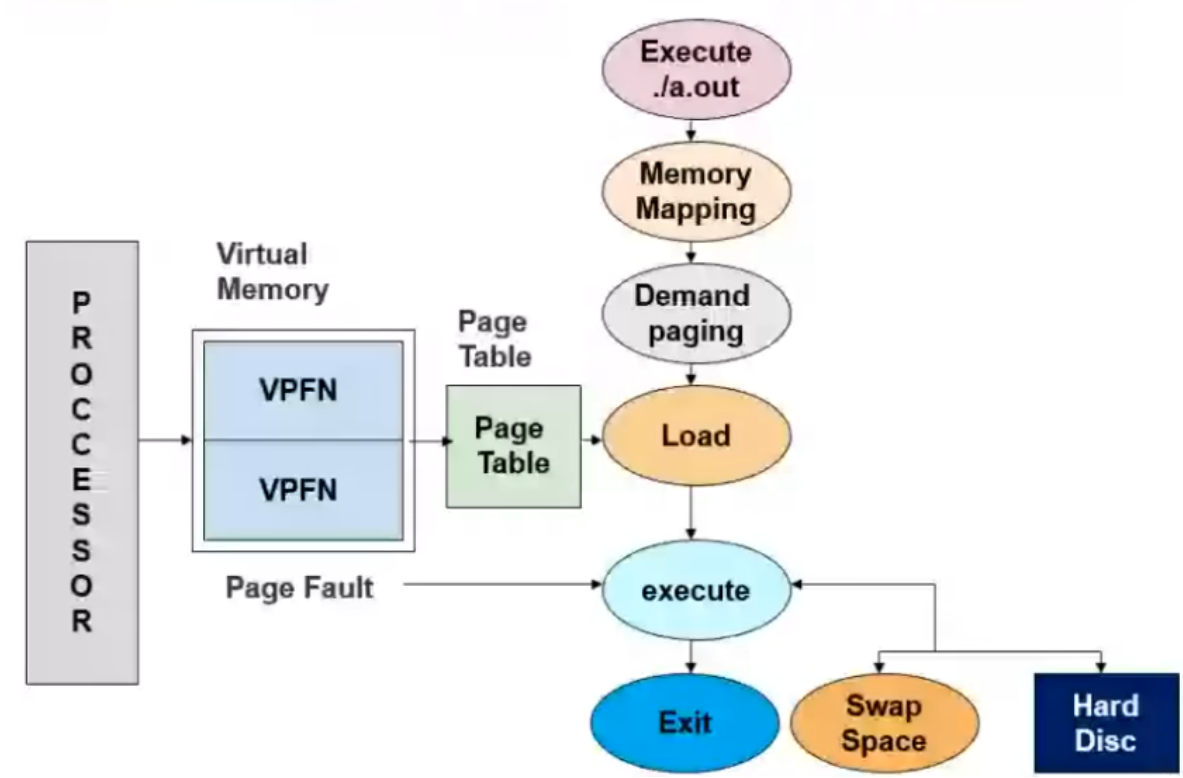
free -m



	total	used	free	shared	buffers	cached
Mem:	7806	691	7115	0	30	188
-/+ buffers/cache:		472	7334			
Swap:	1999	0	1999			

Mem: used = 472 + 30 + 188 = 690

Walk Through Program Execution



When you execute a program “a.out”, it is divided into chunks of size, PAGE_SIZE (4KiB). One virtual address is assigned for each chunk/page and this is the memory mapping part. Then the demand paging concept is used to load only the first page into main memory.

Loading a page involves translating the virtual address of the page (Virtual Page Frame Number, VPFN) into a physical address which requires the Page Table (TLB).
After which the page starts to execute.
Once it finishes execution, the second page has to be loaded into main memory for which it will generate a page fault and the page fault handler will load this into main memory.
The page that has to be loaded after a page fault can either be in swap space (minor page fault) or in the hard disk (major page fault).
The pages will keep on loading and executing till the program exits.