

Processes in Linux

A program/command when executed, a special instance is provided by the system to the process. This instance consists of all the services/resources that may be utilised by the process under execution.

- Whenever a command is issued in Unix/Linux, it creates/starts a new process. For example, pwd when issued which is used to list the current directory location the user is in, a process starts.
- Through a 5 digit ID number Unix/Linux keeps an account of the processes, this number is called process ID or PID. Each process in the system has a unique PID.
- Used up pid's can be used in again for a newer process since all the possible combinations are used.
- At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

Types of Process in Linux

In Linux, processes are fundamental units of execution that run independently and perform various tasks. There are different types of processes in Linux, which can be categorised based on various criteria. Here are some common types of processes in Linux:

1. ****Foreground Processes****:

- These are processes that run in the foreground and interact with the user. They typically receive input from the terminal and display output on the terminal. Examples include text editors, shells, and other interactive applications.

2. ****Background Processes****:

- Background processes run independently of the terminal and do not require user interaction. These processes are usually started with an ampersand (&) at the end of the command to run them in the background.

3. ****Daemon Processes****:

- Daemons are background processes that provide services or perform system tasks. They are often started at system boot and run throughout the system's uptime. Examples include web servers, print spoolers, and network services.

4. ****Parent and Child Processes****:

- In Linux, processes can create child processes. The parent process typically spawns one or more child processes. Child processes inherit certain attributes from their parent processes.

5. ****Zombie Processes****:

- Zombie processes are completed processes that have not yet been fully removed from the system's process table. They exist in a "zombie" state until their exit status is collected by their parent process. This typically happens when the parent process is still running.

6. ****Orphan Processes****:

- Orphan processes are those whose parent processes have terminated or become unresponsive. In such cases, these orphan processes are adopted by the init process (usually PID 1) and continue running.

7. ****Real-Time Processes****:

- Real-time processes have higher priority than normal processes and are designed for tasks that require guaranteed and timely execution. They are used in applications like robotics, industrial control, and multimedia processing.

8. ****User and System Processes****:

- User processes are initiated and controlled by users, while system processes are essential for the functioning of the operating system. User processes have higher priority than system processes.

9. ****Multi-Threaded Processes****:

- In Linux, a single process can consist of multiple threads of execution. These threads share the same memory space and resources, allowing for concurrent execution.

10. ****Kernel Threads****:

- These are lightweight threads created and managed by the kernel. They are used for tasks that require kernel-level operations, such as I/O operations and hardware management.

11. ****Interrupt-Driven Processes****:

- These processes are not always running but are triggered by hardware or software interrupts. When a particular event occurs, an interrupt-driven process is started to handle the event.

Understanding the different types of processes in Linux is essential for managing and troubleshooting tasks related to process control, system administration, and performance optimization. Each type of process serves a specific purpose in the Linux operating system.

Filters Used in Linux

In Linux, filters are often used with various commands to manipulate and process text or data. Here are three commonly used filters with their syntax:

1. grep:

- Syntax: ``grep [options] pattern [file(s)]``
- Description: ``grep`` is used to search for a specified pattern (a regular expression) in one or more files or standard input. It prints lines that match the pattern.

Example:

```
bash
grep "search-term" file.txt
```

2. sed (Stream Editor):

- Syntax: ``sed [options] 'script' [file(s)]``
- Description: ``sed`` is a text stream editor used for performing basic text transformations on an input stream (a file or input from a pipeline). It supports various text processing operations like search and replace.

Example:

```
bash
sed 's/old-text/new-text/g' file.txt
```

3. awk:

- Syntax: ``awk [options] 'script' [file(s)]``

- Description: `awk` is a versatile text processing tool that is often used for data extraction and reporting. It operates on records (lines) and fields within those records.

Example:

```
bash
```

```
awk '{print $2}' data.txt
```

These filters are powerful and commonly used for text manipulation and data processing in the Linux command line. The provided syntax examples are just basic illustrations; each of these commands offers a wide range of options and features for more advanced text processing tasks.

Initializing a process

A process can be run in two ways:

Method 1: Foreground Process : Every process when started runs in foreground by default, receives input from the keyboard, and sends output to the screen. When issuing pwd command

```
$ ls pwd
```

Output:

```
$ /home/geeksforgeeks/root
```

When a command/process is running in the foreground and is taking a lot of time, no other processes can be run or started because the prompt would not be available until the program finishes processing and comes out.

Method 2: Background Process: It runs in the background without keyboard input and waits till keyboard input is required. Thus, other processes can be done in parallel with the process running in the background since they do not have to wait for the previous process to be completed.

Adding & along with the command starts it as a background process

```
$ pwd &
```

Since pwd does not want any input from the keyboard, it goes to the stop state until moved to the foreground and given any data input. Thus, on pressing Enter:

Output:

```
[1]  +   Done                pwd
$
```

That first line contains information about the background process – the job number and the process ID. It tells you that the ls command background process finishes successfully. The second is a prompt for another command.

Tracking ongoing processes

ps (Process status) can be used to see/list all the running processes.

```
$ ps
```

PID	TTY	TIME	CMD
19	pts/1	00:00:00	sh
24	pts/1	00:00:00	ps

For more information -f (full) can be used along with ps

```
$ ps -f
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
52471	19	1	0	07:20	pts/1	00:00:00f	sh
52471	25	19	0	08:04	pts/1	00:00:00	ps -f

For single-process information, ps along with process id is used

```
$ ps 19
```

PID	TTY	TIME	CMD
19	pts/1	00:00:00	sh

For a running program (named process) **Pidof** finds the process id's (pids)

Fields described by ps are described as:

- **UID:** User ID that this process belongs to (the person running it)
- **PID:** Process ID
- **PPID:** Parent process ID (the ID of the process that started it)
- **C:** CPU utilization of process
- **STIME:** Process start time
- **TTY:** Terminal type associated with the process
- **TIME:** CPU time is taken by the process
- **CMD:** The command that started this process

There are other options which can be used along with ps command :

- **-a:** Shows information about all users
- **-x:** Shows information about processes without terminals
- **-u:** Shows additional information like -f option
- **-e:** Displays extended information

Stopping a process:

When running in foreground, hitting Ctrl + c (interrupt character) will exit the command. For processes running in background kill command can be used if it's pid is known.

```
$ ps -f
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
52471	19	1	0	07:20	pts/1	00:00:00	sh
52471	25	19	0	08:04	pts/1	00:00:00	ps -f

```
$ kill 19
```

Terminated

If a process ignores a regular kill command, you can use kill -9 followed by the process ID.

```
$ kill -9 19
```

Terminated

Other process commands:

bg: A job control command that resumes suspended jobs while keeping them running in the background

Syntax:

```
bg [ job ]
```

For example:

```
bg %19
```

fg: It continues a stopped job by running it in the foreground.

Syntax:

```
fg [ %job_id ]
```

For example

```
fg 19
```

top: This command is used to show all the running processes within the working environment of Linux.

Syntax:

```
top
```

nice: It starts a new process (job) and assigns it a priority (nice) value at the same time.

Syntax:

```
nice [-nice value]
```

nice value ranges from -20 to 19, where -20 is of the highest priority.

renice : To change the priority of an already running process renice is used.

Syntax:

```
renice [-nice value] [process id]
```

df: It shows the amount of available disk space being used by file systems

Syntax:

```
df
```

Output:

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/loop0	18761008	15246876	2554440	86%	/
none	4	0	4	0%	
/sys/fs/cgroup					
udev	493812	4	493808	1%	/dev
tmpfs	100672	1364	99308	2%	/run

```

none          5120          0          5120      0% /run/lock
none          503352        1764        501588      1% /run/shm
none          102400         20         102380      1% /run/user
/dev/sda3     174766076 164417964 10348112    95% /host

```

free: It shows the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel

Syntax:

free

Output:

```

              total          used          free          shared          buffers
cached
Mem:         1006708        935872        70836              0        148244
346656
-/+ buffers/cache:        440972        565736
Swap:         262140        130084        132056

```

Types of Processes

1. **Parent and Child process :** The 2nd and 3rd column of the ps -f command shows process id and parent's process id number. For each user process, there's a parent process in the system, with most of the commands having shell as their parent.
2. **Zombie and Orphan process :** After completing its execution a child process is terminated or killed and SIGCHLD updates the parent process about the termination and thus can continue the task assigned to it. But at times when the parent process is killed before the termination of the child process, the child processes become orphan processes, with the parent of all processes "init" process, becomes their new pid.
A process which is killed but still shows its entry in the process status or the process table is called a zombie process, they are dead and are not used.
3. **Daemon process :** They are system-related background processes that often run with the permissions of root and services requests from other processes, they most of the time run in the background and wait for processes it can work along with for ex print daemon. When ps -ef is executed, the process with ? in the tty field are daemon processes.

I/O system calls

System calls are the calls that a program makes to the system kernel to provide the services to which the program does not have direct access. For example, providing access to input and output devices such as monitors and

keyboards. We can use various functions provided in the C Programming language for input/output system calls such as create, open, read, write, etc.

Input/Output System Calls

Basically, there are total 5 types of I/O system calls:

1. C create

The create() function is used to create a new empty file in C. We can specify the permission and the name of the file which we want to create using the create() function. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

Syntax of create() in C

```
int create(char *filename, mode_t mode);
```

Parameter

- **filename:** name of the file which you want to create
- **mode:** indicates permissions of the new file.

Return Value

- return first unused file descriptor (generally 3 when first creating use in the process because 0, 1, 2 fd are reserved)
- return -1 when an error

How C create() works in OS

- Create a new empty file on the disk.
- Create file table entry.
- Set the first unused file descriptor to point to the file table entry.
- Return file descriptor used, -1 upon failure.

2. C open

The open() function in C is used to open the file for reading, writing, or both. It is also capable of creating the file if it does not exist. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

Syntax of open() in C

```
int open (const char* Path, int flags);
```

Parameters

- **Path:** Path to the file which we want to open.
 - Use the **absolute path** beginning with "/" when you are **not working in the same directory** as the C source file.

- Use **relative path** which is only the file name with extension, when you are **working in the same directory** as the C source file.
- **flags:** It is used to specify how you want to open the file. We can use the following flags.

Flags	Description
O_RDONLY	Opens the file in read-only mode.
O_WRONLY	Opens the file in write-only mode.
O_RDWR	Opens the file in read and write mode.
O_CREAT	Create a file if it doesn't exist.
O_EXCL	Prevent creation if it already exists.
O_APPEND	Opens the file and places the cursor at the end of the contents.
O_ASYNC	Enable input and output control by signal.
O_CLOEXEC	Enable close-on-exec mode on the open file.
O_NONBLOCK	Disables blocking of the file opened.
O_TMPFILE	Create an unnamed temporary file at the specified path.

How C open() works in OS

- Find the existing file on the disk.
- Create file table entry.
- Set the first unused file descriptor to point to the file table entry.
- Return file descriptor used, -1 upon failure.

3. C close

The `close()` function in C tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. It is defined inside `<unistd.h>` header file.

Syntax of `close()` in C

```
int close(int fd);
```

Parameter

- **fd**: File descriptor of the file that you want to close.

Return Value

- **0** on success.
- **-1** on error.

How C `close()` works in the OS

- Destroy file table entry referenced by element `fd` of the file descriptor table
 - As long as no other process is pointing to it!
- Set element `fd` of file descriptor table to **NULL**

3. C `close`

The `close()` function in C tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. It is defined inside `<unistd.h>` header file.

Syntax of `close()` in C

```
int close(int fd);
```

Parameter

- **fd**: File descriptor of the file that you want to close.

Return Value

- **0** on success.
- **-1** on error.

How C `close()` works in the OS

- Destroy file table entry referenced by element `fd` of the file descriptor table
 - As long as no other process is pointing to it!
- Set element `fd` of file descriptor table to **NULL**

5. C `write`

Writes `cnt` bytes from `buf` to the file or socket associated with `fd`. `cnt` should not be greater than `INT_MAX` (defined in the `limits.h` header file). If `cnt` is zero, `write()` simply returns 0 without attempting any other action.

The `write()` is also defined inside `<unistd.h>` header file.

Syntax of `write()` in C

```
size_t write (int fd, void* buf, size_t cnt);
```

Parameters

- **fd**: file descriptor
- **buf**: buffer to write data to.
- **cnt**: length of the buffer.

Return Value

- returns the number of bytes written on success.
- return 0 on reaching the End of File.
- return -1 on error.
- return -1 on signal interrupts.

Important Points about C write

- The file needs to be opened for write operations
- **buf** needs to be at least as long as specified by **cnt** because if **buf** size is less than the **cnt** then **buf** will lead to the overflow condition.
- **cnt** is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when **fd** has a less number of bytes to write than **cnt**.
- If **write()** is interrupted by a signal, the effect is one of the following:
 - If **write()** has not written any data yet, it returns -1 and sets **errno** to **EINTR**.
 - If **write()** has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

. *poll* and *select* have essentially the same functionality: both allow a process to determine whether it can read from or write to one or more open files without blocking. They are thus often used in applications that must use multiple input or output streams without blocking on any one of them

select and poll functions

In Linux (and UNIX-like systems), the ``select()`` function is a system call used for multiplexing input and output. It allows a program to monitor multiple file descriptors to see if they are ready for reading, writing, or if they have an exception. This is particularly useful in scenarios where an application might be waiting on data from multiple sources, and it avoids the need to use blocking system calls.

Here's a breakdown of how ``select()`` works:

Syntax:

```
```c
int select(int nfds, fd_set *readfds, fd_set *writefds,
 fd_set *exceptfds, struct timeval *timeout);
...
```
```

Parameters:

1. ****nfds****: The highest-numbered file descriptor in any of the three sets, plus 1.

2. **readfds**: A set of file descriptors to be checked for being ready to read.
3. **writfds**: A set of file descriptors to be checked for being ready to write.
4. **exceptfds**: A set of file descriptors to be checked for exceptions.
5. **timeout**: Maximum interval to wait for the select to complete. It's a `struct timeval` with seconds and microseconds. If `timeout` is NULL, `select()` will block indefinitely until one of the file descriptors is ready. If `timeout` is set to 0, `select()` will return immediately.

Return Value:

- **> 0**: The number of ready descriptors that are contained in the descriptor sets.
- **0**: Indicates that the time limit expired.
- **-1**: An error occurred (with the error stored in `errno`).

Usage:

1. Initialize the file descriptor sets using `FD_ZERO`.
2. Add file descriptors to the sets using `FD_SET`.
3. Call `select()` with the desired sets.
4. After the call, check the sets with `FD_ISSET` to see which descriptors are ready.

Example:

Here's a simple example that uses `select()` to wait for data to be ready to read on a socket:

```
``c
#include <stdio.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0); // Assume this is connected to
    some server
    fd_set read_fds;

    FD_ZERO(&read_fds);
    FD_SET(sockfd, &read_fds);

    struct timeval timeout;
    timeout.tv_sec = 5; // Wait up to 5 seconds
    timeout.tv_usec = 0;

    int result = select(sockfd + 1, &read_fds, NULL, NULL, &timeout);

    if (result == -1) {
        perror("select()");
    } else if (result == 0) {
        printf("Timeout occurred! No data after 5 seconds.\n");
    } else {
        if (FD_ISSET(sockfd, &read_fds)) {
```

```

        printf("Data is ready to be read from socket.\n");
        // read data from socket
    }
}

close(sockfd);
return 0;
}
...

```

Notes:

- `select()` can be used to monitor not just sockets, but any file descriptor, including files, pipes, and so on.
- Over time, `select()` has shown its age. It has issues, especially when dealing with a large number of file descriptors because its performance is $O(n)$ for n file descriptors. For modern applications, especially those requiring high-performance I/O multiplexing, other mechanisms like `poll()` and `epoll()` (Linux-specific) are often preferred.

Poll Functions

The `poll()` function in Linux (and other UNIX-like systems) is an I/O multiplexing mechanism similar to `select()`, but with some advantages. It allows a program to monitor multiple file descriptors to see if they are ready for reading, writing, or have some other event pending.

The main advantage of `poll()` over `select()` is that it doesn't have a hardcoded limit on the number of file descriptors, and its interface can be more intuitive due to the use of a structured array of `pollfd` instead of the bitmap approach of `select()`.

Syntax:

```

...c
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
...

```

Parameters:

1. ****fds****: An array of structures describing the file descriptors to be monitored. Each structure is of type `struct pollfd`:

```

...c
struct pollfd {
    int fd;        // File descriptor to poll
    short events;  // Requested events to monitor

```

```
    short revents; // Returned events found
};
...

```

2. **nfds**: The number of structures in the `fds` array.
3. **timeout**: Time in milliseconds to wait for an event to occur. If `timeout` is `-1`, `poll()` will block indefinitely. If `timeout` is `0`, `poll()` will return immediately.

Return Value:

- **> 0**: The number of `pollfd` structures with non-zero `revents` fields (indicating the number of file descriptors with events or errors).
- **0**: The timeout interval passed without any of the requested events occurring.
- **-1**: An error occurred (with the error stored in `errno`).

Usage:

1. Initialize the `pollfd` structures and set the `fd` and `events` fields.
2. Call `poll()` with the array and desired timeout.
3. After the call, check the `revents` field in each `pollfd` structure to see which descriptors have events or errors.

Example:

Here's a simple example that uses `poll()` to wait for data to be ready to read on a socket:

```
``c
#include <stdio.h>
#include <sys/poll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0); // Assume this is
    connected to some server
    struct pollfd pfd;

```

```

pfd.fd = sockfd;
pfd.events = POLLIN; // Monitor for incoming data

int timeout = 5000; // Wait up to 5000 milliseconds (5 seconds)

int result = poll(&pfd, 1, timeout);

if (result == -1) {
    perror("poll()");
} else if (result == 0) {
    printf("Timeout occurred! No data after 5 seconds.\n");
} else {
    if (pfd.revents & POLLIN) {
        printf("Data is ready to be read from socket.\n");
        // read data from socket
    }
}

close(sockfd);
return 0;
}
...

```

Notes:

- The `poll()` system call is more scalable than `select()` when dealing with a moderate number of file descriptors. However, for applications that require handling a very large number of file descriptors or require more complex event monitoring, `epoll()` (which is Linux-specific) can offer even better performance and scalability.
- Besides `POLLIN` for readable data, there are other events you can monitor, such as `POLLOUT` for write-ability, `POLLERR` for errors, and more.

[Filters in UNIX](#)

In UNIX/Linux, filters are the set of commands that take input from standard input stream i.e. **stdin**, perform some operations and write output to standard output stream i.e. **stdout**. The stdin and stdout can be managed as per preferences using redirection and pipes. Common filter commands are: [grep](#), [more](#), [sort](#).

1. [grep](#) Command:It is a pattern or expression matching command. It searches for a pattern or regular expression that matches in files or directories and then prints found matches.

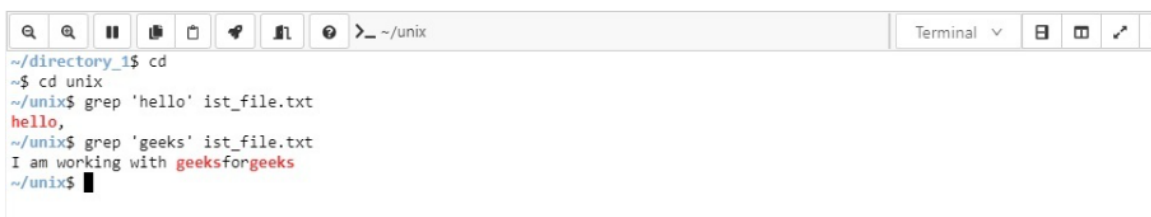
Syntax:

```
$grep[options] "pattern to be matched" filename
```

Example:

Input : \$grep 'hello' ist_file.txt

Output : searches hello in the ist_file.txt and outputs/returns the lines containing 'hello'.

A screenshot of a terminal window with a light gray title bar. The terminal shows a series of commands and their outputs. The prompt is ~/directory_1\$. The user enters 'cd', then '\$ cd unix', then '~/unix\$ grep 'hello' ist_file.txt', which outputs 'hello,'. Then the user enters '~/unix\$ grep 'geeks' ist_file.txt', which outputs 'I am working with geeksforgeeks'. The prompt is ~/unix\$.

The Options in grep command are:

| S.no | OPTIONS | DESCRIPTION |
|------|---------|---|
| 01 | -v | Returns all lines that do not match the specified regular expression. |
| 02 | -n | Returns all lines that match the specified regular expression along with line no. |
| 03 | -l | Returns only names of files matching the specified regular expression. |
| 04 | -c | Returns count of lines that match regular expression. |
| 05 | -i | It is case sensitive option and matches either upper-case or lower-case |

Grep command can also be used with meta-characters:

Example:

Input : \$grep 'hello' *

Output : it searches for *hello* in all the files and directories.

* is a meta-character and returns matching 0 or more preceding characters

2. sort Command: It is a data manipulation command that sorts or merges lines in a file by specified fields. In other words it sorts lines of text alphabetically or numerically, **default sorting is alphabetical.**

Syntax:

```
$sort[options] filename
```

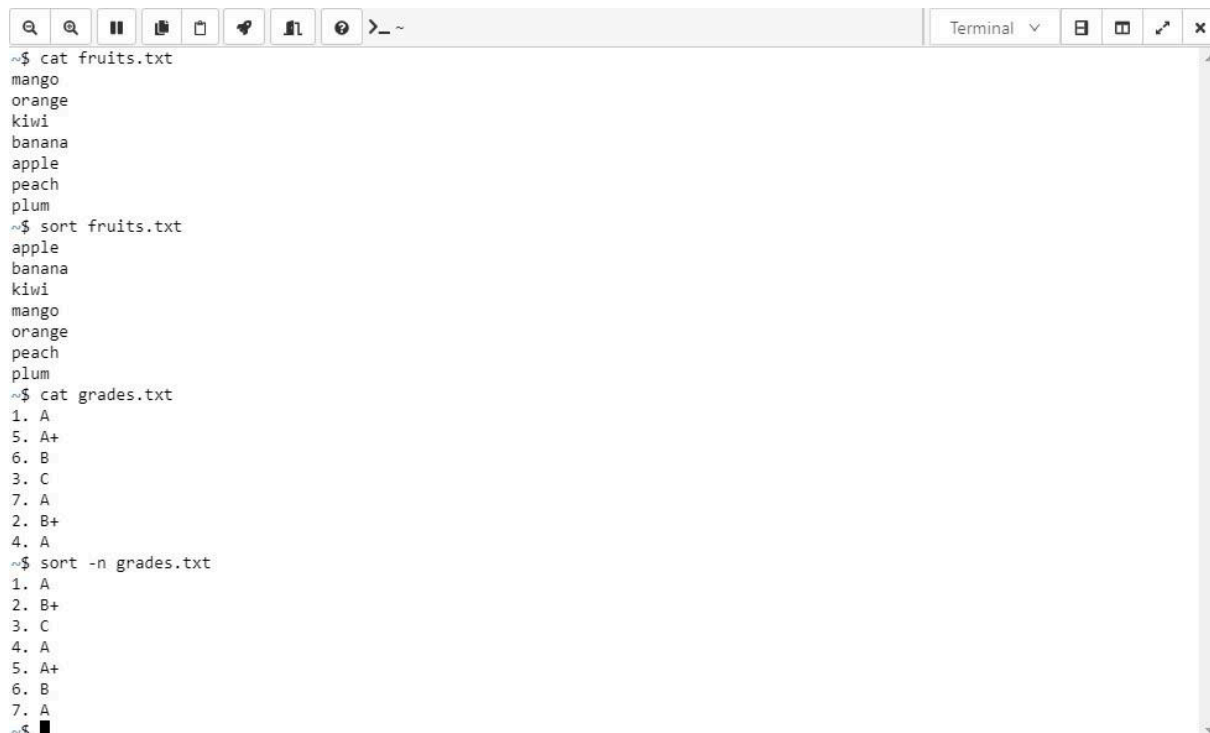
The options include:

| S.NO. | OPTIONS | DESCRIPTION |
|-------|---------|--|
| 01. | -n | It sorts the lines in numerical order. |
| 02. | -r | It reverses the order of sorting.
Eg.
If the lines were sort from a to z, it will sort them from z to a. |
| 03. | -f | It sorts uppercase and lowercase together. |
| 04. | +x | It doesn't consider ist x fields while sorting. |

Example:

```
$sort fruits.txt
```

```
$sort -n grades.txt
```

A terminal window with a toolbar at the top. The command prompt is '~\$'. The user enters 'cat fruits.txt' and the terminal displays the contents of the file: mango, orange, kiwi, banana, apple, peach, plum. Then the user enters 'sort fruits.txt' and the terminal displays the sorted contents: apple, banana, kiwi, mango, orange, peach, plum. Next, the user enters 'cat grades.txt' and the terminal displays a list of grades: 1. A, 5. A+, 6. B, 3. C, 7. A, 2. B+, 4. A. Finally, the user enters 'sort -n grades.txt' and the terminal displays the numerically sorted grades: 1. A, 2. B+, 3. C, 4. A, 5. A+, 6. B, 7. A. The prompt is now '~\$' with a cursor.

```
~$ cat fruits.txt
mango
orange
kiwi
banana
apple
peach
plum
~$ sort fruits.txt
apple
banana
kiwi
mango
orange
peach
plum
~$ cat grades.txt
1. A
5. A+
6. B
3. C
7. A
2. B+
4. A
~$ sort -n grades.txt
1. A
2. B+
3. C
4. A
5. A+
6. B
7. A
~$
```

3. [more](#) Command: It is used to customize the displaying contents of file. It displays the text file contents on the terminal with paging controls. Following key controls are used:

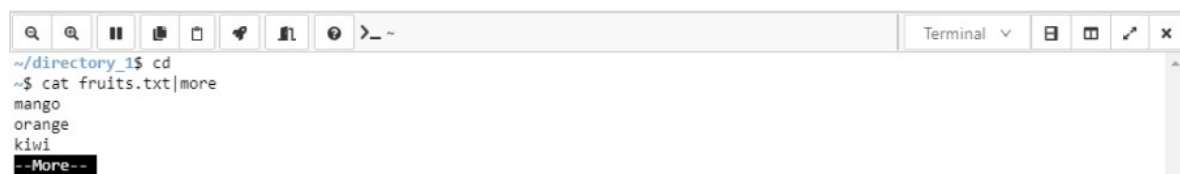
- To display next line, press the enter key
- To bring up next screen, press spacebar
- To move to the next file, press n
- To quit, press q.

Syntax:

`$more[options] filename`

Example:

`cat fruits.txt | more`

A terminal window showing the execution of the 'more' command. The prompt is '~/directory_1\$'. The user enters 'cd' and the prompt changes to '~\$'. Then the user enters 'cat fruits.txt|more'. The terminal displays the first three lines of the file: mango, orange, kiwi. At the bottom, there is a black bar with the text '--More--' in white, indicating that the command is waiting for input to continue displaying the file.

```
~/directory_1$ cd
~$ cat fruits.txt|more
mango
orange
kiwi
--More--
```

While using more command, the bottom of the screen contains more prompt where commands are entered to move through the text.

Redirections

In Linux and other Unix-like operating systems, redirection refers to the practice of directing the input and/or output of commands to and from files, or to other commands. Redirection allows you to manipulate data in various ways without having to modify the commands themselves.

Here's a quick guide to the most common forms of redirection:

1. **Standard Output Redirection (`stdout`)**:**

- **`>`**: Redirects the standard output of a command to a file. If the file doesn't exist, it's created. If it does exist, it's overwritten.

```
```bash
echo "Hello, World" > output.txt
```
```

- **`>>`**: Redirects the standard output of a command to a file. If the file doesn't exist, it's created. If it does exist, new data is appended to it.

```
```bash
echo "Another line" >> output.txt
```
```

2. **Standard Error Redirection (`stderr`)**:**

- **`2>`**: Redirects the standard error of a command to a file. If the file doesn't exist, it's created. If it does exist, it's overwritten.

```
```bash
ls non_existent_file 2> error.txt
```
```

- **`2>>`**: Redirects the standard error of a command to a file and appends the output if the file already exists.

```
```bash
ls another_non_existent_file 2>> error.txt
```
```

3. **Redirecting Both `stdout` and `stderr`**:**

- To the same file:

```
```bash
```

```
some_command > log.txt 2>&1
```
```

- To different files:

```
```bash
some_command > output.txt 2> error.txt
```
```

4. **Standard Input Redirection (`stdin`)**:**

- **`<`**: Takes the input for a command from a file rather than the keyboard.

```
```bash
sort < unsorted.txt
```
```

5. **Pipes (`|`)**:**

- A pipe takes the standard output of one command and sends it as the standard input to another. This is extremely useful for chaining commands together.

```
```bash
cat somefile.txt | grep "search_term"
```
```

6. **Process Substitution**:**

- **`<()`**: Replaces the construct with a file descriptor containing the output of the command inside the parentheses.

- **`>()`**: Can be used to provide input from a file descriptor.

```
```bash
diff <(sort file1.txt) <(sort file2.txt)
```
```

Points to be taken care of:

- We need to be careful with the **`>`** operator, as it will overwrite the target file if it exists.

- The constructs `2>&1` and `>&2` can be thought of as merging streams. In the case of `2>&1`, it's merging `stderr` into `stdout`.
- Understanding redirection is crucial for efficient command-line work and scripting, as it allows for powerful data manipulation and analysis directly from the terminal.

Redirection is a fundamental concept in Linux/Unix and can be combined in versatile ways to achieve complex tasks with simple commands. It underscores the philosophy of Unix-like systems where "everything is a file."

Pipes in UNIX

Piping is used to give the output of one command (written on LHS) as input to another command (written on RHS). Commands are piped together using vertical bar “|” symbol. **Syntax:**

```
command 1|command 2
```

Example:

- **Input:** `ls|more`
- **Output:** `more` command takes input from `ls` command and appends it to the standard output. It displays as many files that fit on the screen and highlighted *more* at the bottom of the screen. To see the next screen hit enter or spacebar to move one line at a time or one screen at a time respectively.

Linux file system navigation

1. Root Directory (`/`):

- Everything in Linux starts from the root directory, denoted by a forward slash (`/`).
- Key subdirectories include:
 - `/bin`: Essential command binaries.
 - `/etc`: System configurations.
 - `/home`: Home directories for users.
 - `/usr`: Utilities and applications.
 - `/var`: Variable data like logs.

2. Current Location:

- `**`pwd`**` (Print Working Directory): Displays the directory you're currently in.

3. Changing Directories:

- **`cd`** (Change Directory): Command to move around.
 - `cd /path/to/directory`: Go to a specific directory.
 - `cd ..`: Go up one directory.
 - `cd`: Go to your home directory.

4. Listing Contents:

- **`ls`**: Lists the contents of a directory.
 - Basic options:
 - `-l`: Long format.
 - `-a`: Include hidden files.
 - `-h`: Human-readable sizes.

5. Paths:

- **Absolute Path**: Begins from the root (`/`). Example:
`/home/user/documents`.
- **Relative Path**: Relative to your current location. If you're in `/home/user/`, `documents` refers to `/home/user/documents`.

6. Finding Files:

- **`find`**: Searches for files.
 - `find /start/path -name "filename"`: Search from a specific location.

7. Viewing File Contents:

- **`cat`**: Display content of a file.
- **`less`**: View content page by page.

8. Autocomplete:

- Pressing the `Tab` key autocompletes commands or file and directory names.

9. File Managers:

- GUI-based tools for file system navigation:
 - **`nautilus`**: GNOME's file manager.
 - **`dolphin`**: KDE's file manager.
 - **`thunar`**: XFCE's file manager.

Tips:

1. ``man [command]``: Use the man command to access the manual pages for a command, e.g., ``man ls``.
2. Directory names are case-sensitive.
3. Hidden files and directories start with a dot (``.``), e.g., ``config``.

Directory Access

Directory access in Linux is about controlling permissions and permissions for users and groups to access, list, and manipulate the contents of directories. This is an important aspect of maintaining security and privacy on a Linux system.

1. **Permissions**:

Linux employs a permission system that controls who can access files and directories. There are three types of permissions:

- **Read (``r``)**: Allows listing files in the directory.
- **Write (``w``)**: Permits creating, deleting, and renaming files in the directory.
- **Execute (``x``)**: Enables traversing into the directory and accessing its contents.

2. **Directory Ownership**:

Every file and directory in Linux is associated with an owner and a group. The owner is the user who created the file or directory, and the group is a collection of users.

3. **Setting Permissions**:

- **``chmod``**: Command to change permissions of files and directories.
 - Example: ``chmod 755 directory_name`` grants read, write, and execute permissions to the owner, and read and execute permissions to the group and others.
 - Numerical values: ``4`` for read, ``2`` for write, ``1`` for execute. Sum them for different combinations.

- Use ``-R`` for recursive changes in subdirectories.

4. **Changing Ownership**:

- **``chown``**: Command to change ownership of files and directories.
- Example: ``chown user:group directory_name``.

5. **Listing Permissions**:

- **``ls -l``**: Displays a long listing that includes permissions, ownership, and other information.

6. **Access Control Lists (ACLs)**:

- ACLs provide a more fine-grained way to control access by allowing you to set specific permissions for users and groups beyond the default owner and group.
- **``getfacl``** and **``setfacl``** commands are used to view and modify ACLs.

7. **Default Permissions**:

You can set default permissions and ownership for newly created files and directories within a directory using the **``umask``** command or by modifying the ``.bashrc`` file.

8. **Special Permissions**:

- **``Setuid (s)``**: Allows a program to be executed with the privileges of the file owner.
- **``Setgid (s)``**: When set on a directory, files created in that directory inherit the group of the parent directory.
- **``Sticky (t)``**: When set on a directory, only the owner of a file can delete it from the directory.

Understanding and managing directory access is vital for maintaining the security and integrity of a Linux system, allowing users and applications to interact with files and directories in a controlled and secure manner.

File System Implementation

The primary file system used in Linux distributions is the **Extended File System (ext)** family, with `ext4` being the most commonly used version. However, there are other file systems available as well, each with its own features and characteristics. Let's delve into the basics of file system implementation in Linux:

1. **File System Concepts**:

- **Blocks**: Files are stored in fixed-size blocks on storage devices.
- **Inodes**: Data structures that store metadata about files (permissions, ownership, timestamps).
- **Directories**: Special files that map filenames to inodes.

2. **File System Types**:

1. **ext2 (Second Extended File System)**:

- The predecessor of `ext3` and `ext4`.
- Basic features including block groups, inodes, and directories.

2. **ext3 (Third Extended File System)**:

- An enhancement over `ext2` with journaling support.
- Journaling ensures data consistency and faster recovery after crashes.

3. **ext4 (Fourth Extended File System)**:

- The latest in the `ext` family, offering improvements over `ext3`.
- Larger file system and file sizes, better performance, and more.

4. **Btrfs (B-tree File System)**:

- Modern file system with advanced features like snapshots, data compression, and checksums.
- Aimed at addressing scalability and fault tolerance.

5. **XFS (Extended File System)**:

- High-performance file system known for scalability, handling large files and devices efficiently.

- Supports features like journaling, extent-based allocation, and online defragmentation.

6. **ZFS (Zettabyte File System)**:

- Not part of the Linux kernel but can be used through third-party implementations.
- Offers advanced features including data integrity checks, storage pooling, snapshots, and more.

3. **File System Operations**:

1. **Formatting a File System**:

- Before using a storage device, it must be formatted with a specific file system type. This process creates the necessary data structures for file organization.

2. **Mounting and Unmounting**:

- Mounting attaches a file system to a directory hierarchy, making its contents accessible.
- Unmounting detaches the file system from the directory hierarchy.

3. **File Allocation**:

- Allocation strategies determine how files are stored on disk blocks. Common methods include contiguous, linked list, and indexed allocation.

4. **Journaling**:

- Journaling is a mechanism that records changes to the file system in a journal (log) before they're applied to the actual file system structures. This helps recover from crashes or system failures.

4. **Commands and Utilities**:

- **mkfs**: Used to create a file system on a storage device.
- **mount**: Command to attach a file system to the directory hierarchy.
- **umount**: Used to detach a file system.
- **df**: Displays disk space usage of file systems.
- **du**: Shows disk space used by directories and files.

5. **File System Maintenance**:

- Regular maintenance includes defragmentation (for file systems that require it) and periodic checks for errors (`fsck` command).

Inode in Linux

What is an inode?

Linux® must allocate an index node (inode) for every file and directory in the filesystem. Inodes do not store actual data. Instead, they store the metadata where you can find the storage blocks of each file's data.

Metadata in an inode

The following metadata exists in an inode:

- File type
- Permissions
- Owner ID
- Group ID
- Size of file
- Time last accessed
- Time last modified
- Soft/Hard Links
- Access Control List (ACLs)

Check the inode number in a specific file

There are different ways to check the inode number. The following example shows the creation of a file named mytestfile. The command `stat` displays the file statistics, including the unique inode number:

```
[root@Rackspace-Server /]# touch mytestfile
[root@Rackspace-Server /]# stat mytestfile
File: mytestfile
Size: 0          Blocks: 0          IO Block: 4096   regular empty file
Device: ca01h/51713d  Inode: 13         Links: 1
Access: (0644/-rw-r--r--)  Uid: (  0/   root)  Gid: (  0/   root)
Context: unconfined_u:object_r:etc_runtime_t:s0
Access: 2021-03-26 15:51:27.036124392 -0500
Modify: 2021-03-26 15:51:27.036124392 -0500
Change: 2021-03-26 15:51:27.036124392 -0500
Birth: -
```

You can also check the inode number of mytestfile by listing the contents of the directory. You can run a combination of commands in the directory by using `ls` or `grep`, as shown in the following examples:

```
[root@Rackspace-Server /]# ls -lhi | grep mytestfile
13 -rw-r--r--. 1 root root 0 Mar 26 15:51 mytestfile
```

```
[root@Rackspace-Server /]# ls -li mytestfile
13 mytestfile
```

Check the inode usage on filesystems

The following example checks the inodes on all the mounted filesystems, focusing on /dev/xvda1, which has a maximum inode allocation of 1,310,720:

```
[root@Rackspace-Server ~]# df -i
Filesystem      Inodes IUsed  IFree IUse% Mounted on
devtmpfs        99906  318  99588   1% /dev
tmpfs           103934   1 103933   1% /dev/shm
tmpfs           103934  500 103434   1% /run
tmpfs           103934  17 103917   1% /sys/fs/cgroup
/dev/xvda1     1310720 47034 1263686   4% /
```

Adding the flag -h to the preceding df command does not give you an exact number, but it provides a more readable output:

```
[root@Rackspace-Server ~]# df -ih
Filesystem      Inodes IUsed IFree IUse% Mounted on
devtmpfs        98K  318  98K   1% /dev
tmpfs           102K   1 102K   1% /dev/shm
tmpfs           102K  500 102K   1% /run
tmpfs           102K  17 102K   1% /sys/fs/cgroup
/dev/xvda1     1.3M  46K 1.3M   4% /
```

Count inodes under a certain directory

To check the number of inodes in a specific directory, run the following command:

```
find <DIRECTORY> | wc -l
```

The following example checks the file count in the /root directory.

```
[root@Rackspace-Server ~]# pwd
/root
[root@Rackspace-Server ~]# find . | wc -l
11
```

In this case, it shows 11 files created under /root.

What happens to the inode assigned when moving or copying a file?

When you copy a file, Linux assigns a different inode to the new file, as shown in the following example:

```
[root@Rackspace-Server inodes]# touch file1
[root@Rackspace-Server inodes]# ls -lhi
total 0
262446 -rw-r--r--. 1 root root 0 Mar 26 19:30 file1
```

```
[root@Rackspace-Server inodes]# cp file1 file2
```

```
[root@Rackspace-Server inodes]# ls -lhi
total 0
262446 -rw-r--r--. 1 root root 0 Mar 26 19:30 file1
262440 -rw-r--r--. 1 root root 0 Mar 26 19:31 file2
```

Things are different when moving a file. As long as the file does not change filesystems, the inode remains the same:

```
[root@Rackspace-Server inodes]# ls -lhi directory1/file1
262440 -rw-r--r--. 1 root root 0 Mar 26 19:34 directory1/file1
```

```
[root@Rackspace-Server inodes]# mv directory1/file1 directory2/
```

```
[root@Rackspace-Server inodes]# ls -lhi directory2/file1
262440 -rw-r--r--. 1 root root 0 Mar 26 19:34 directory2/file1
```

The following example moves file1 from /dev/xvda1 to the /dev/xvdb1 filesystem:

```
[root@Rackspace-Server inodes]# df -hP {/,/backups}
Filesystem      Size  Used Avail Use% Mounted on
/dev/xvda1      20G  2.4G  17G  13% /
/dev/xvdb1      391M    0  391M   0% /backups
```

```
[root@Rackspace-Server inodes]# pwd
/inodes
```

```
[root@Rackspace-Server inodes]# mv test /backups
```

```
[root@Rackspace-Server inodes]# ls -lhi /backups/test
117329 -rw-r--r--. 1 root root 0 Mar 26 19:34 /backups/te
```

How to Change Directory Permissions in Linux for the Group Owners and Others

The command for changing directory permissions for group owners is similar, but add a “g” for group or “o” for users:

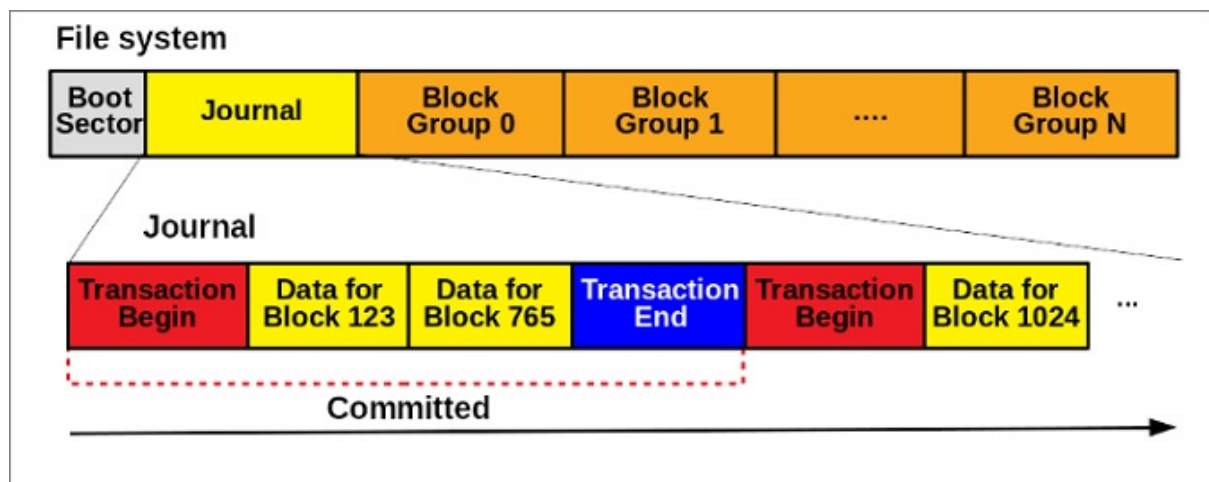
```
chmod g+w filename
chmod g-wx filename
chmod o+w filename
chmod o-rwx foldername
```

To change directory permissions for everyone, use “u” for users, “g” for group, “o” for others, and “ugo” or “a” (for all).

chmod ugo+rw foldername to give read, write, and execute to everyone.

chmod a=r foldername to give only read permission for everyone.

Disk Layout of ext3 file system



The ext3 file system is an extension of the ext2 file system and includes features like journaling for improved reliability. The disk layout consists of several components:

1. ****Boot Block:****

- Located at the beginning of the disk.
- Contains the boot loader and other essential information for bootstrapping the operating system.

2. ****Superblock:****

- Follows the boot block.
- Contains metadata about the entire file system, such as the file system type, size, block size, and other configuration parameters.

3. **Block Group Descriptors:**

- Follow the superblock.
- Describes the layout of data blocks within each block group.

4. **Block Bitmaps:**

- Represent the allocation status of data blocks in the block group.
- Each bit in the block bitmap corresponds to a block, indicating whether it is in use (allocated) or free.

5. **Inode Bitmap:**

- Indicates which inodes in the block group are in use or free.

6. **Inode Table:**

- Contains inodes, which are data structures that store metadata about files and directories (e.g., permissions, owner, size, timestamps).

7. **Data Blocks:**

- Store the actual file data and directory contents.
- Divided into regular data blocks, indirect blocks, double indirect blocks, and triple indirect blocks to efficiently handle file data of varying sizes.

8. **Journal:**

- Stores a log of transactions to provide a journaling feature for improved recovery in case of a system crash or unexpected shutdown.

These components are organized into block groups, and the file system can have multiple block groups depending on its size. Each block group contains a copy of the superblock and block group descriptors for redundancy.

Keep in mind that this is a simplified description, and the actual layout may vary based on factors like file system size, block size, and other configuration options. If you need a visual representation, consider looking for diagrams or illustrations online that depict the ext3 file system layout.

How to Change Groups of Files and Directories in Linux

By issuing these commands, you can change groups of files and directories in Linux.

```
chgrp groupname filename
```

```
chgrp groupname foldername
```

Note that the group must exist before you can assign groups to files and directories.

Changing ownership in Linux

Another helpful command is changing ownerships of files and directories in Linux:

```
chown name filename
```

```
chown name foldername
```



These commands will give ownership to someone, but all sub files and directories still belong to the original owner.

You can also combine the group and ownership command by using:

```
chown -R name:filename /home/name/directoryname
```

Changing Linux permissions in numeric code

You may need to know how to change permissions in numeric code in Linux, so to do this you use numbers instead of “r”, “w”, or “x”.

0 = No Permission

1 = Execute

2 = Write

4 = Read

Basically, you add up the numbers depending on the level of permission you want to give.



Permission numbers are:

0 = ---

1 = --x

2 = -w-

3 = -wx

4 = r-

5 = r-x

6 = rw-

7 = rwx

For example:

`chmod 777 foldername` will give read, write, and execute permissions for everyone.

`chmod 700 foldername` will give read, write, and execute permissions for the user only.

`chmod 327 foldername` will give write and execute (3) permission for the user, w (2) for the group, and read, write, and execute for the users.

What is the Linux File System?

Linux file system is generally a built-in layer of a **Linux operating system** used to handle the data management of the storage. It helps to arrange the file on the disk storage. It manages the file name, file size, creation date, and much more information about a file.

If we have an unsupported file format in our file system, we can download software to deal with it.

Linux File System Structure

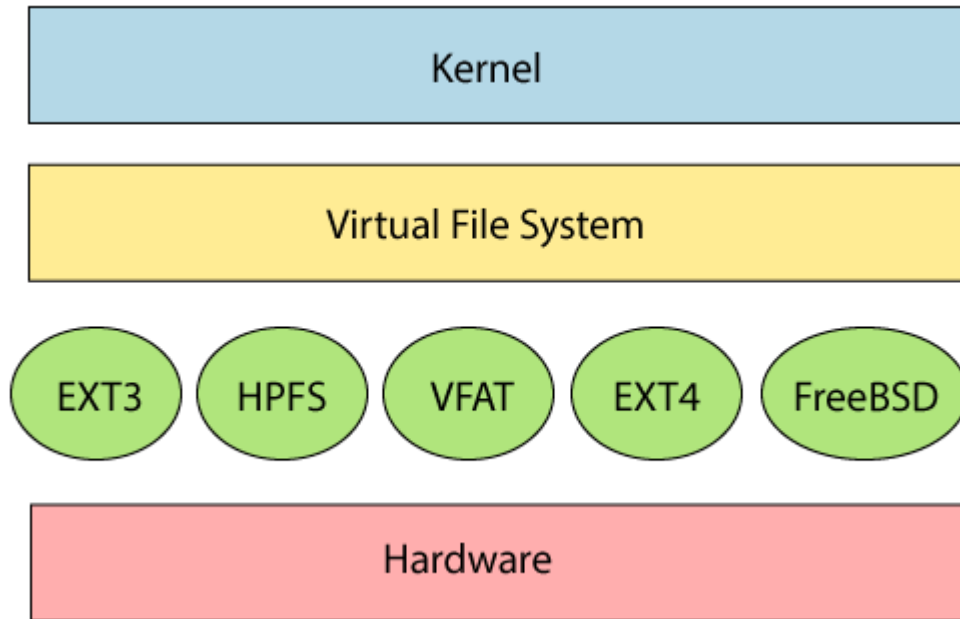
Linux file system has a hierarchal file structure as it contains a root directory and its subdirectories. All other directories can be accessed from the root directory. A partition usually has only one file system, but it may have more than one file system. A file system is designed in a way so that it can manage and provide space for non-volatile storage data. All file systems required a namespace that is a naming and organizational methodology. The namespace defines the naming process, length of the file name, or a subset of characters that can be used for the file name. It also defines the logical structure of files on a memory segment, such as the use of directories for organizing the specific files. Once a namespace is described, a Metadata description must be defined for that particular file.

The data structure needs to support a hierarchical directory structure; this structure is used to describe the available and used disk space for a particular block. It also has the other details about the files such as file size, date & time of creation, update, and last modified.

Also, it stores advanced information about the section of the disk, such as partitions and volumes.

The advanced data and the structures that it represents contain the information about the file system stored on the drive; it is distinct and independent of the file system metadata.

Linux file system contains two-part file system software implementation architecture. Consider the below image:



The file system requires an API (Application programming interface) to access the function calls to interact with file system components like files and directories. **API** facilitates tasks such as creating, deleting, and copying the files. It facilitates an algorithm that defines the arrangement of files on a file system.

The first two parts of the given file system together called a **Linux virtual file system**. It provides a single set of commands for the kernel and developers to access the file system. This virtual file system requires the specific system driver to give an interface to the file system.

Directory Structure

The directories help us to store the files and locate them when we need them. Also, directories are called folders as they can be assumed of as folders where files reside in the form of a physical desktop analogy. Directories can be organized in a tree-like hierarchy in Linux and several other operating systems.

The directory structure of Linux is well-documented and defined in the Linux FHS (Filesystem Hierarchy Standard). Referencing those directories if accessing them is achieved via the sequentially deeper names of the directory linked by '/' forward slash like /var/spool/mail and /var/log. These are known as paths.

The below table gives a very short standard, defined, and well-known top-level Linux directory list and their purposes:

- **/ (root filesystem):** It is the top-level filesystem directory. It must include every file needed to boot the Linux system before another filesystem is mounted. Every other filesystem is mounted on a well-defined and standard mount point because of the root filesystem directories after the system is started.
- **/boot:** It includes the static kernel and bootloader configuration and executable files needed to start a Linux computer.
- **/bin:** This directory includes user executable files.
- **/dev:** It includes the device file for all hardware devices connected to the system. These aren't device drivers; instead, they are files that indicate all devices on the system and provide access to these devices.

- **/etc:** It includes the local system configuration files for the host system.
- **/lib:** It includes shared library files that are needed to start the system.
- **/home:** The home directory storage is available for user files. All users have a subdirectory inside /home.
- **/mnt:** It is a temporary mount point for basic filesystems that can be used at the time when the administrator is working or repairing a filesystem.
- **/media:** A place for mounting external removable media devices like USB thumb drives that might be linked to the host.
- **/opt:** It contains optional files like vendor supplied application programs that must be placed here.
- **/root:** It's the home directory for a root user. Keep in mind that it's not the '/' (root) file system.
- **/tmp:** It is a temporary directory used by the OS and several programs for storing temporary files. Also, users may temporarily store files here. Remember that files may be removed without prior notice at any time in this directory.
- **/sbin:** These are system binary files. They are executables utilized for system administration.
- **/usr:** They are read-only and shareable files, including executable libraries and binaries, man files, and several documentation types.
- **/var:** Here, variable data files are saved. It can contain things such as MySQL, log files, other database files, email inboxes, web server data files, and much more.

Two type of modes in linux file system

In its life span a process executes in [user mode and kernel mode](#). The **User mode** is normal mode where the process has limited access. While the **Kernel mode** is the privileged mode where the process has unrestricted access to system resources like hardware, memory, etc. A process can access I/O Hardware registers to program it, can execute OS kernel code and access kernel data in Kernel mode. Anything related to Process management, IO hardware management, and Memory management requires process to execute in Kernel mode. This is important to know that a process in Kernel mode gets power to access any device and memory, and same time any crash in kernel mode brings down the whole system. But any crash in user mode brings down the faulty process only. The kernel provides System Call Interface (**SCI**), which are the entry points for kernel. System Calls are the only way through which a process can go into kernel mode from user mode. Below diagram explains user mode to kernel mode transition in detail.

In modern operating systems, software runs in two distinct modes: user mode and kernel mode. User mode is a restricted mode that limits the software's access to system resources, while kernel mode is a privileged mode that allows software to access system resources and perform privileged operations.

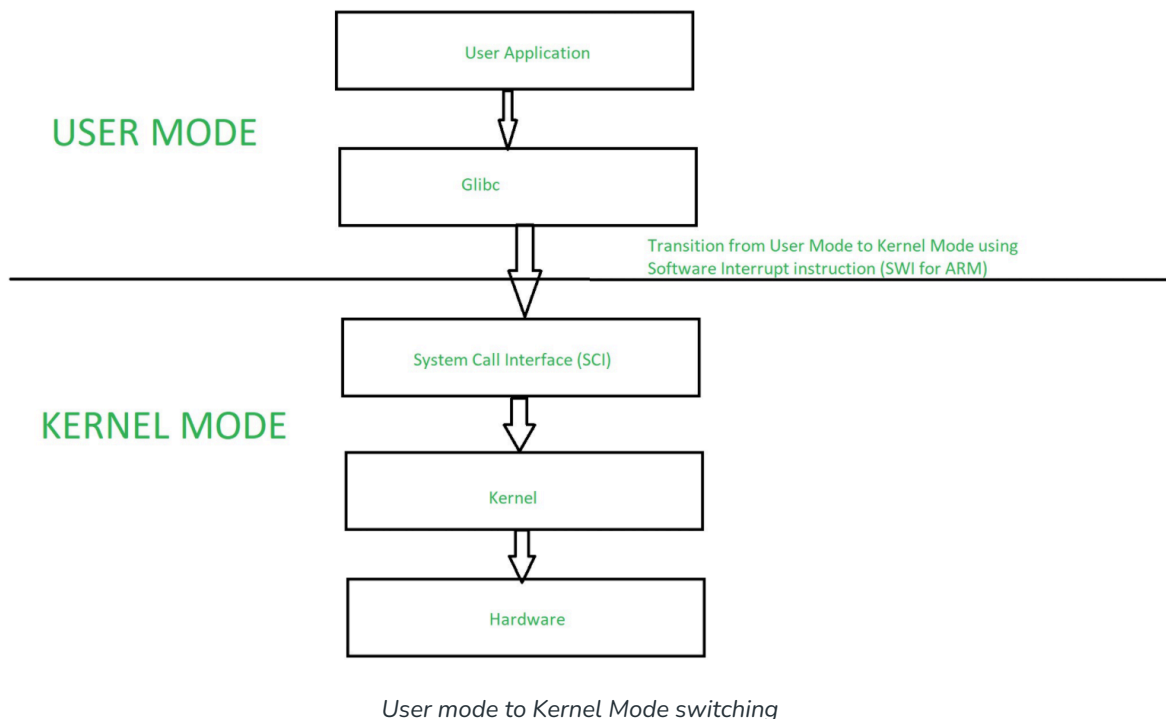
When a user-level application needs to perform an operation that requires kernel mode access, such as accessing hardware devices or modifying system settings, it must make a system call to the operating system kernel. The operating system switches the processor from user mode to kernel mode to execute the system call, and then switches back to user mode once the operation is complete.

This switching between user mode and kernel mode is known as mode switching or context switching. Mode switching involves saving the current context of the processor in memory, switching to the new mode, and loading the new context into the processor. This process can be time-consuming and resource-intensive, as it involves switching between different privilege levels and saving and restoring processor state.

Here are some key points about user mode and kernel mode switching:

1. User mode is a restricted mode that limits access to system resources, while kernel mode is a privileged mode that allows access to system resources.
2. User mode applications must make system calls to access kernel mode resources or perform privileged operations.
3. Mode switching involves saving the current context of the processor, switching to the new mode, and loading the new context into the processor.
4. Mode switching can be time-consuming and resource-intensive, and can impact system performance.

5. Modern operating systems use various techniques to minimize mode switching, such as caching kernel mode data in user mode, and using hardware support for virtualization and context switching.



To go into Kernel mode, an application process.

- Calls the *Glibc* library function.
- *Glibc* library knows the proper way of calling System Call for different architectures. It setup passing arguments as per architecture's Application Binary Interface (ABI) to prepare for System Call entry.
- Now *Glibc* calls SWI instruction (Software Interrupt instruction for ARM), which puts processor into Supervisor mode by updating Mode bits of CPSR register and jumps to vector address 0x08.
- Till now process execution was in User mode. After SWI instruction execution, the process is allowed to execute kernel code. Memory

Management Unit (MMU) will now allow kernel Virtual memory access and execution, for this process.

- From Vector address `0x08`, process execution loads and jumps to SW Interrupt handler routine, which is `vector_swi()` for ARM.
- In `vector_swi()`, System Call Number (SCNO) is extracted from SWI instruction and execution jumps to system call function using SCNO as index in system call table `sys_call_table`.
- After System Call execution, in return path, user space registers are restored before starting execution in User Mode.

To support kernel mode and user mode, processor must have hardware support for different privilege modes. For example ARM processor supports seven different modes.

| Processor Mode | CPSR Mode bits | Remark |
|----------------|----------------|---------------------------|
| User | 10000 | No privilege or user mode |
| FIQ | 10001 | Fast Interrupt mode |
| IRQ | 10010 | Interrupt mode |

| | | |
|------------|-------|--|
| Supervisor | 10011 | Kernel mode |
| Abort | 10111 | Mode for memory violation handling |
| Undefined | 11011 | Undefined instruction handling mode |
| System | 11111 | Same as Supervisor mode but with re-entrancy |

Conclusion : For any system, privilege mode and non-privilege mode is important for access protection. The processor must have hardware support for user/kernel mode. System Call Interfaces (SCI) are the only way to transit from User space to kernel space. Kernel space switching is achieved by Software Interrupt, which changes the processor mode and jump the CPU execution into interrupt handler, which executes corresponding System Call routine.

Linux File System Features

In Linux, the file system creates a tree structure. All the files are arranged as a tree and its branches. The topmost directory called the **root (/) directory**. All other directories in Linux can be accessed from the root directory.

Some key **features of Linux** file system are as following:

- **Specifying paths:** Linux does not use the backslash (\) to separate the components; it uses forward slash (/) as an alternative. For example, as in Windows, the data may be stored in C:\ My Documents\ Work, whereas, in Linux, it would be stored in /home/ My Document/ Work.
- **Partition, Directories, and Drives:** Linux does not use drive letters to organize the drive as Windows does. In Linux, we cannot tell whether we are addressing a partition, a network device, or an "ordinary" directory and a Drive.
- **Case Sensitivity:** Linux file system is case sensitive. It distinguishes between lowercase and uppercase file names. Such as, there is a difference between

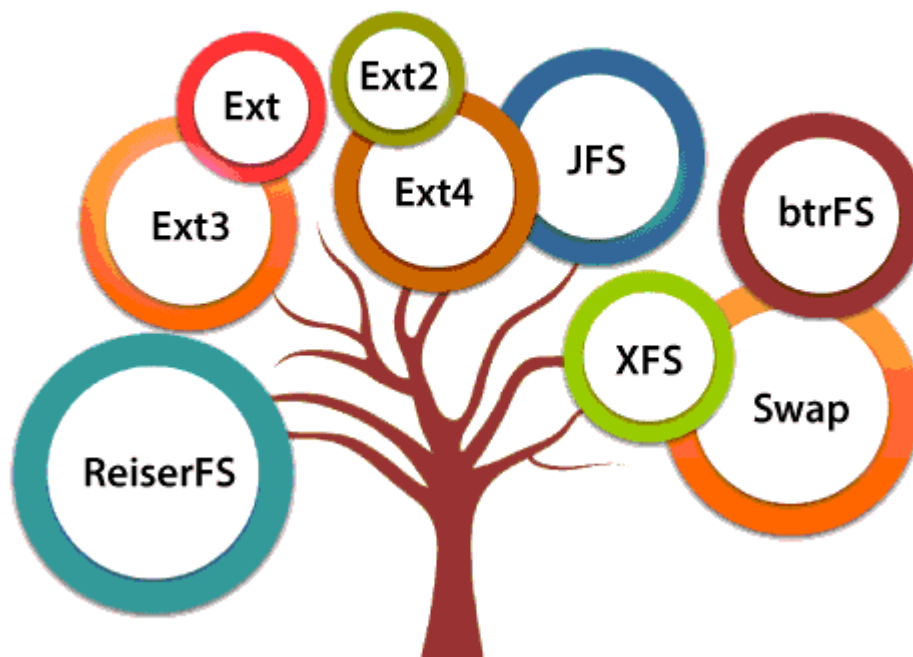
test.txt and Test.txt in Linux. This rule is also applied for directories and Linux commands.

- **File Extensions:** In Linux, a file may have the extension '.txt,' but it is not necessary that a file should have a file extension. While working with Shell, it creates some problems for the beginners to differentiate between files and directories. If we use the graphical file manager, it symbolizes the files and folders.
- **Hidden files:** Linux distinguishes between standard files and hidden files, mostly the configuration files are hidden in Linux OS. Usually, we don't need to access or read the hidden files. The hidden files in Linux are represented by a dot (.) before the file name (e.g., .ignore). To access the files, we need to change the view in the file manager or need to use a specific command in the shell.

Types of Linux File System

When we install the Linux operating system, Linux offers many file systems such as Ext, Ext2, Ext3, Ext4, JFS, ReiserFS, XFS, btrfs, and swap.

Types of Linux File System



Let's understand each of these file systems in detail:

1. Ext, Ext2, Ext3 and Ext4 file system

The file system Ext stands for **Extended File System**. It was primarily developed for **MINIX OS**. The Ext file system is an older version, and is no longer used due to some limitations.

Ext2 is the first Linux file system that allows managing two terabytes of data. Ext3 is developed through Ext2; it is an upgraded version of Ext2 and contains backward compatibility. The major drawback of Ext3 is that it does not support servers because this file system does not support file recovery and disk snapshot.

Ext4 file system is the faster file system among all the Ext file systems. It is a very compatible option for the SSD (solid-state drive) disks, and it is the default file system in Linux distribution.

2. JFS File System

JFS stands for **Journaled File System**, and it is developed by **IBM for AIX Unix**. It is an alternative to the Ext file system. It can also be used in place of Ext4, where stability is needed with few resources. It is a handy file system when **CPU** power is limited.

3. ReiserFS File System

ReiserFS is an alternative to the Ext3 file system. It has improved performance and advanced features. In the earlier time, the ReiserFS was used as the default file system in SUSE Linux, but later it has changed some policies, so SUSE returned to Ext3. This file system dynamically supports the file extension, but it has some drawbacks in performance.

4. XFS File System

XFS file system was considered as high-speed JFS, which is developed for parallel I/O processing. NASA still using this file system with its high storage server (300+ Terabyte server).

5. Btrfs File System

Btrfs stands for the **B tree file system**. It is used for fault tolerance, repair system, fun administration, extensive storage configuration, and more. It is not a good suit for the production system.

6. Swap File System

The swap file system is used for memory paging in Linux operating system during the system hibernation. A system that never goes in hibernate state is required to have swap space equal to its **RAM** size.

Soft and Hard links in Unix/Linux

A link in UNIX is a pointer to a file. Like pointers in any programming language, links in UNIX are pointers pointing to a file or a directory. Creating links is a kind of shortcut to access a file. Links allow more than one file name to refer to the same file, elsewhere.

There are two types of links :

1. Soft Link or Symbolic links
2. Hard Links

These links behave differently when the source of the link (what is being linked to) is moved or removed. Symbolic links are not updated (they merely contain a string which is the path name of its target); hard links always refer to the source, even if moved or removed.

For example, if we have a file a.txt. If we create a hard link to the file and then delete the file, we can still access the file using hard link. But if we create a

soft link of the file and then delete the file, we can't access the file through soft link and soft link becomes dangling. Basically hard link increases reference count of a location while soft links work as a shortcut (like in Windows)

1. Hard Links

- Each hard linked file is assigned the same Inode value as the original, therefore they reference the same physical file location. Hard links more flexible and remain linked even if the original or linked files are moved throughout the file system, although hard links are unable to cross different file systems.
- `ls -l` command shows all the links with the link column shows number of links.
- Links have actual file contents
- Removing any link, just reduces the link count, but doesn't affect other links.
- Even if we change the filename of the original file then also the hard links properly work.
- We cannot create a hard link for a directory to avoid recursive loops.
- If original file is removed then the link will still show the content of the file.
- The size of any of the hard link file is same as the original file and if we change the content in any of the hard links then size of all hard link files are updated.
- The disadvantage of hard links is that it cannot be created for files on different file systems and it cannot be created for special files or directories.
- Command to create a hard link is:

```
$ ln [original filename] [link name]
```

2. Soft Links

- A soft link is similar to the file shortcut feature which is used in Windows Operating systems. Each soft linked file contains a separate Inode value that points to the original file. As similar to hard links, any changes to the data in either file is reflected in the other. Soft links can be linked across different file systems, although if the original file is deleted or moved, the soft linked file will not work correctly (called hanging link).

- `ls -l` command shows all links with first column value `l?` and the link points to original file.
- Soft Link contains the path for original file and not the contents.
- Removing soft link doesn't affect anything but removing original file, the link becomes "dangling" link which points to nonexistent file.
- A soft link can link to a directory.
- The size of the soft link is equal to the length of the path of the original file we gave. E.g if we link a file like `ln -s /tmp/hello.txt /tmp/link.txt` then the size of the file will be 14bytes which is equal to the length of the `"/tmp/hello.txt"`.
- If we change the name of the original file then all the soft links for that file become dangling i.e. they are worthless now.
- Link across file systems: If you want to link files across the file systems, you can only use symlinks/soft links.
- Command to create a Soft link is:

```
$ ln -s [original filename] [link name]
```

"**Asynchronous I/O**" essentially refers to the ability of a process to perform input/output on multiple sources at one time. For instance a process may be processing a file on disk as well as serving a client over the network, essentially unrelated tasks; the question is: which should it give priority? If the process was to read from the network socket, it could be waiting ("blocking") quite a while before the data arrived, time which could have been spent processing the file. On the other hand the opposite could also be true (especially if the file resides on a slow medium such as a floppy disk, or a networked filesystem).

Updated: 15/7/2005

In general **Asynchronous I/O** revolves around two functions: The ability to determine that *data is available* (in the case of a network connection, terminal, or certain other devices) or that a *pending I/O operation has completed*. The first case can be generalised to include being able to determine that a device or network connection is *ready to receive new data*. All these (italicised) things are **Asynchronous** events, that is, they can happen at an arbitrary time during program execution.

Process Signal Mask

The collection of signals that are currently blocked is called the *signal mask*. Each process has its own signal mask. When you create a new process, it inherits its parent's mask. You can block or unblock signals with total flexibility by modifying the signal mask.

Swap Space Mechanism

Swap space in Linux is used when the amount of physical memory (RAM) is full. If the system needs more memory resources and the RAM is full, inactive pages in memory are moved to the swap space. While swap space can help machines with a small amount of RAM, it should not be considered a replacement for more RAM. Swap space is located on hard drives, which have a slower access time than physical memory. Swap space can be a dedicated swap partition (recommended), a swap file, or a combination of swap partitions and swap files. Note that *Btrfs* does *not* support swap space.

In years past, the recommended amount of swap space increased linearly with the amount of RAM in the system. However, modern systems often include hundreds of gigabytes of RAM. As a consequence, recommended swap space is considered a function of system memory workload, not system memory.

[Table 15.1, “Recommended System Swap Space”](#) illustrates the recommended size of a swap partition depending on the amount of RAM in your system and whether you want sufficient memory for your system to hibernate. The recommended swap partition size is established automatically during installation. To allow for hibernation, however, you need to edit the swap space in the custom partitioning stage.

Recommendations in [Table 15.1, “Recommended System Swap Space”](#) are especially important on systems with low memory (1 GB and less). Failure to allocate sufficient swap space on these systems can cause issues such as instability or even render the installed system unbootable.

Table 15.1. Recommended System Swap Space

| Amount of
RAM in the
system | Recommended
swap space | Recommended swap space if
allowing for hibernation |
|-----------------------------------|---------------------------|---|
| <hr/> | | |

| | | |
|----------------|----------------------------|-----------------------------|
| ≤ 2 GB | 2 times the amount of RAM | 3 times the amount of RAM |
| > 2 GB – 8 GB | Equal to the amount of RAM | 2 times the amount of RAM |
| > 8 GB – 64 GB | At least 4 GB | 1.5 times the amount of RAM |
| > 64 GB | At least 4 GB | Hibernation not recommended |

Note

There are two reasons why hibernation is not recommended with systems with more than 64 GB of RAM. Firstly, hibernation requires extra space for an inflated (and perhaps infrequently utilized) swap area. Secondly, the process of moving resident data from RAM to disk and back on can take a lot of time to complete.

At the border between each range listed in [Table 15.1, “Recommended System Swap Space”](#), for example a system with 2 GB, 8 GB, or 64 GB of system RAM, discretion can be exercised with regard to chosen swap space and hibernation support. If your system resources allow for it, increasing the swap space may lead to better performance.

Note that distributing swap space over multiple storage devices also improves swap space performance, particularly on systems with fast drives, controllers, and interfaces.

Important

File systems and LVM2 volumes assigned as swap space *should not* be in use when being modified. Any attempts to modify swap fail if a system process or the kernel is using swap space. Use the `free` and `cat /proc/swaps` commands to verify how much and where swap is in use.

You should modify swap space while the system is booted in `rescue` mode, see [Booting Your Computer in Rescue Mode](#) in the *Red Hat Enterprise Linux 7 Installation Guide*. When prompted to mount the file system, select **Skip**.

15.1. Adding Swap Space

Sometimes it is necessary to add more swap space after installation. For example, you may upgrade the amount of RAM in your system from 1 GB to 2 GB, but there is only 2 GB of swap space. It might be advantageous to increase the amount of swap space to 4 GB if you perform memory-intense operations or run applications that require a large amount of memory. You have three options: create a new swap partition, create a new swap file, or extend swap on an existing LVM2 logical volume. It is recommended that you extend an existing logical volume.

15.1.1. Extending Swap on an LVM2 Logical Volume

By default, Red Hat Enterprise Linux 7 uses all available space during installation. If this is the case with your system, then you must first add a new physical volume to the volume group used by the swap space. After adding additional storage to the swap space's volume group, it is now possible to extend it. To do so, perform the following procedure (assuming `/dev/VolGroup00/LogVol01` is the volume you want to extend by 2 GB):

Procedure 15.1. Extending Swap on an LVM2 Logical Volume

1. Disable swapping for the associated logical volume:
2. `# swapoff -v /dev/VolGroup00/LogVol01`
3. Resize the LVM2 logical volume by 2 GB:
4. `# lvresize /dev/VolGroup00/LogVol01 -L +2G`
5. Format the new swap space:
6. `# mkswap /dev/VolGroup00/LogVol01`
7. Enable the extended logical volume:
8. `# swapon -v /dev/VolGroup00/LogVol01`
9. To test if the swap logical volume was successfully extended and activated, inspect active swap space:

```
$ cat /proc/swaps
```

10. `$ free -h`

15.1.2. Creating an LVM2 Logical Volume for Swap

To add a swap volume group 2 GB in size, assuming `/dev/VolGroup00/LogVol02` is the swap volume you want to add:

1. Create the LVM2 logical volume of size 2 GB:
2. `# lvcreate VolGroup00 -n LogVol02 -L 2G`
3. Format the new swap space:
4. `# mkswap /dev/VolGroup00/LogVol02`
5. Add the following entry to the `/etc/fstab` file:
6. `/dev/VolGroup00/LogVol02 swap swap`
`defaults 0 0`
7. Regenerate mount units so that your system registers the new configuration:
8. `# systemctl daemon-reload`
9. Activate swap on the logical volume:
10. `# swapon -v /dev/VolGroup00/LogVol02`
11. To test if the swap logical volume was successfully created and activated, inspect active swap space:

```
$ cat /proc/swaps
```

12. `$ free -h`

15.1.3. Creating a Swap File

To add a swap file:

Procedure 15.2. Add a Swap File

1. Determine the size of the new swap file in megabytes and multiply by 1024 to determine the number of blocks. For example, the block size of a 64 MB swap file is 65536.
2. Create an empty file:
3. `# dd if=/dev/zero of=/swapfile bs=1024 count=65536`
4. Replace *count* with the value equal to the desired block size.
5. Set up the swap file with the command:
6. `# mkswap /swapfile`
7. Change the security of the swap file so it is not world readable.

8. `# chmod 0600 /swapfile`
9. To enable the swap file at boot time, edit `/etc/fstab` as root to include the following entry:
10.

| | | |
|------------------------|-------------------|-------------------|
| <code>/swapfile</code> | <code>swap</code> | <code>swap</code> |
| <code>defaults</code> | <code>0</code> | <code>0</code> |
11. The next time the system boots, it activates the new swap file.
12. Regenerate mount units so that your system registers the new `/etc/fstab` configuration:
13. `# systemctl daemon-reload`
14. To activate the swap file immediately:
15. `# swapon /swapfile`
16. To test if the new swap file was successfully created and activated, inspect active swap space:

```
$ cat /proc/swaps
```

17. `$ free -h`

Catching and Ignoring Signals-sigaction

The `sigaction` function allows the caller to examine or specify the action associated with a specific signal. The `sig` parameter of `sigaction` specifies the signal number for the action. The `act` parameter is a pointer to a struct `sigaction` structure that specifies the action to be taken. The `oact` parameter is a pointer to a struct `sigaction` structure that receives the previous action associated with the signal. If `act` is `NULL`, the call to `sigaction` does not change the action associated with the signal. If `oact` is `NULL`, the call to `sigaction` does not return the previous action associated with the signal.

For a running process, you can check out `/proc/PID/status` and check the fields `SigBlk`, `SigIgn`, and `SigCgt` for blocked, ignored, and caught signals respectively.

What are conditional waits and signals in multi-threading?

Explanation: When you want to sleep a thread, condition variables can be used. In C under Linux, there is a function `pthread_cond_wait()` to wait or sleep.

On the other hand, there is a function `pthread_cond_signal()` to wake up a

sleeping or waiting thread.

Threads can wait on a condition variable.

Syntax of pthread_cond_wait() :

```
int pthread_cond_wait(pthread_cond_t *restrict cond,  
                      pthread_mutex_t *restrict mutex);
```

Pipes in linux

An introduction to pipes and named pipes in Linux

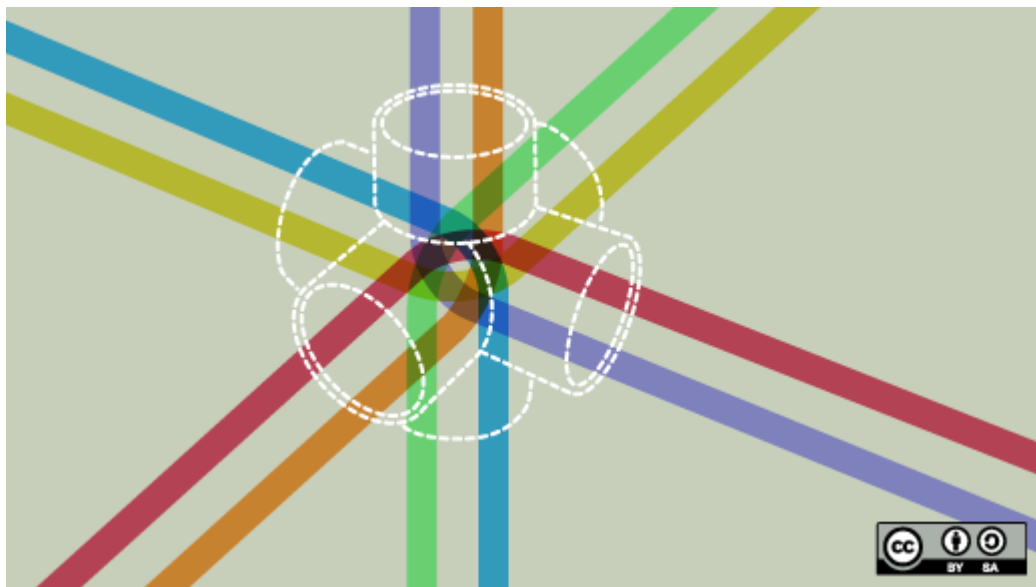


Image by:

Opensource.com

In Linux, the `pipe` command lets you send the output of one command to another. Piping, as the term suggests, can redirect the standard output, input, or error of one process to another for further processing.

The syntax for the `pipe` or `unnamed pipe` command is the `|` character between any two commands:

```
Command-1 | Command-2 | ... | Command-N
```

Here, the pipe cannot be accessed via another session; it is created temporarily to accommodate the execution of `Command-1` and redirect the standard output. It is deleted after successful execution.

```
stack@undercloud-0:~/test
File Edit View Search Terminal Help
[stack@undercloud-0 test]$ cat contents.txt | grep file
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file1
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file2
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file3
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file4
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file5
[stack@undercloud-0 test]$ cat contents.txt | grep "file" | awk '{print $9}'
file1
file2
file3
file4
file5
[stack@undercloud-0 test]$ cat contents.txt | wc -l
9
[stack@undercloud-0 test]$
```

In the example above, `contents.txt` contains a list of all files in a particular directory—specifically, the output of the `ls -al` command. We first `grep` the filenames with the "file" keyword from `contents.txt` by piping (as shown), so the output of the `cat` command is provided as the input for the `grep` command. Next, we add piping to execute the `awk` command, which displays the 9th column from the filtered output from the `grep` command. We can also count the number of rows in `contents.txt` using the `wc -l` command.

A named pipe can last until as long as the system is up and running or until it is deleted. It is a special file that follows the [FIFO](#) (first in, first out) mechanism. It can be used just like a normal file; i.e., you can write to it, read from it, and open or close it. To create a named pipe, the command is:

```
mkfifo <pipe-name>
```

This creates a named pipe file that can be used even over multiple shell sessions.

Another way to create a FIFO named pipe is to use this command:

```
mknod p <pipe-name>
```

To redirect a standard output of any command to another process, use the `>` symbol. To redirect a standard input of any command, use the `<` symbol.

```
stack@undercloud-0:~/test
File Edit View Search Terminal Help
[stack@undercloud-0 test]$ ls -al > contents.txt
[stack@undercloud-0 test]$ cat contents.txt
total 4
drwxrwxr-x.  2 stack stack   91 Aug  8 13:51 .
drwx----- 11 stack stack 4096 Aug  8 13:15 ..
-rw-rw-r--.  1 stack stack    0 Aug  8 13:53 contents.txt
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file1
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file2
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file3
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file4
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file5
[stack@undercloud-0 test]$ tail -n 2 < contents.txt
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file4
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file5
[stack@undercloud-0 test]$
```

As shown above, the output of the `ls -al` command is redirected to `contents.txt` and inserted in the file. Similarly, the input for the `tail` command is provided as `contents.txt` via the `<` symbol.

```
stack@undercloud-0:~/test
File Edit View Search Terminal Help
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file5
[stack@undercloud-0 test]$ tail -n 2 < contents.txt
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file4
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file5
[stack@undercloud-0 test]$ ls
contents.txt file1 file2 file3 file4 file5
[stack@undercloud-0 test]$ clear

[stack@undercloud-0 test]$ mkfifo my-named-pipe
[stack@undercloud-0 test]$ ls -al
total 8
drwxrwxr-x.  2 stack stack  112 Aug  8 13:58 .
drwx----- 11 stack stack 4096 Aug  8 13:15 ..
-rw-rw-r--.  1 stack stack  416 Aug  8 13:53 contents.txt
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file1
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file2
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file3
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file4
-rw-rw-r--.  1 stack stack    0 Aug  8 13:51 file5
prw-rw-r--.  1 stack stack    0 Aug  8 13:58 my-named-pipe
[stack@undercloud-0 test]$ ls -al > my-named-pipe
[stack@undercloud-0 test]$
```

```
stack@undercloud-0:~  
File Edit View Search Terminal Help  
[stack@undercloud-0 ~]$ cat test/my-named-pipe  
total 8  
drwxrwxr-x.  2 stack stack 112 Aug  8 13:58 .  
drwx----- 11 stack stack 4096 Aug  8 13:15 ..  
-rw-rw-r--.  1 stack stack 416 Aug  8 13:53 contents.txt  
-rw-rw-r--.  1 stack stack   0 Aug  8 13:51 file1  
-rw-rw-r--.  1 stack stack   0 Aug  8 13:51 file2  
-rw-rw-r--.  1 stack stack   0 Aug  8 13:51 file3  
-rw-rw-r--.  1 stack stack   0 Aug  8 13:51 file4  
-rw-rw-r--.  1 stack stack   0 Aug  8 13:51 file5  
prw-rw-r--.  1 stack stack   0 Aug  8 13:58 my-named-pipe  
[stack@undercloud-0 ~]$
```

Here, we have created a named pipe, `my-named-pipe`, and redirected the output of the `ls -al` command into the named pipe. We can then open a new shell session and `cat` the contents of the named pipe, which shows the output of the `ls -al` command, as previously supplied. Notice the size of the named pipe is zero and it has a designation of "p".

Socket in Computer Network

A **socket** is one endpoint of a **two way** communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication takes place.

Like 'Pipe' is used to create pipes and sockets are created using '**socket**' system calls. The socket provides a bidirectional **FIFO** Communication facility over the network. A socket connecting to the network is created at each end of the communication. Each socket has a specific address. This address is composed of an IP address and a port number.

Sockets are generally employed in client server applications. The server creates a socket, attaches it to a network port address then waits for the client to contact it. The client creates a socket and then attempts to connect to the

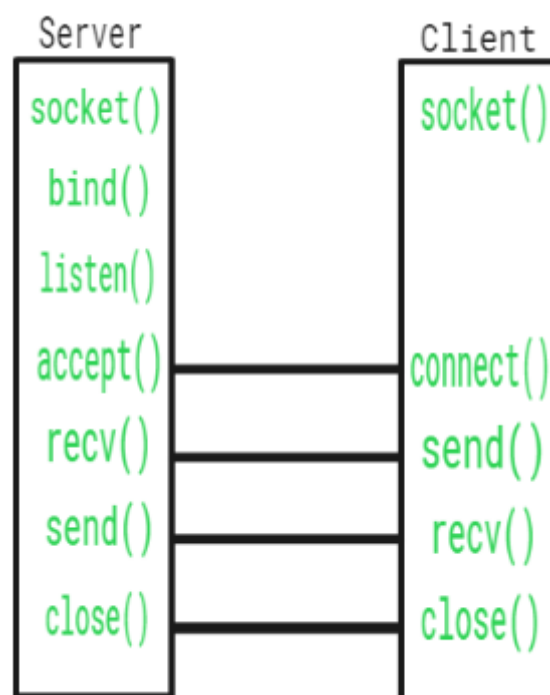
server socket. When the connection is established, transfer of data takes



place.

Types of Sockets : There are two types of Sockets: the **datagram** socket and the **stream** socket.

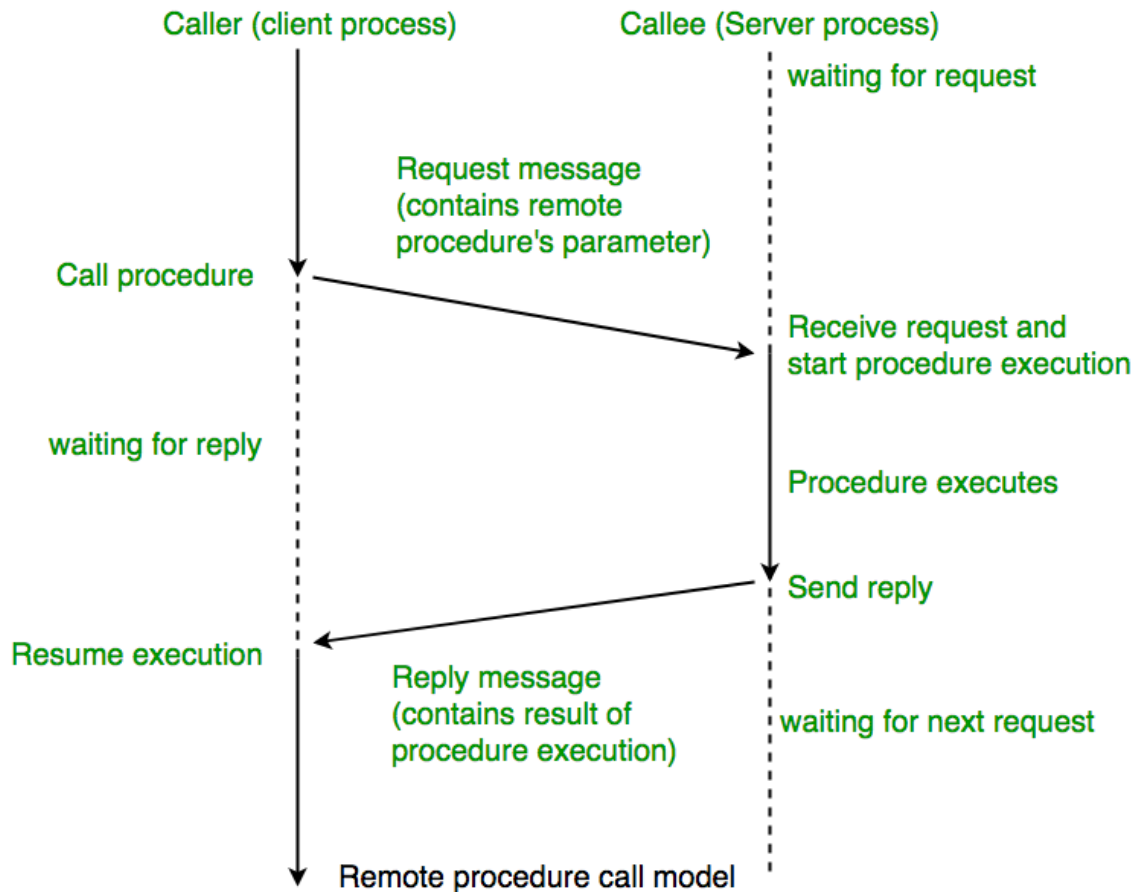
1. **Datagram Socket :** This is a type of network which has connection less point for sending and receiving packets. It is similar to a mailbox. The letters (data) posted into the box are collected and delivered (transmitted) to a letterbox (receiving socket).
2. **Stream Socket** In Computer operating system, a stream socket is a type of [interprocess communications](#) socket or network socket which provides a connection-oriented, sequenced, and unique flow of data without record boundaries with well defined mechanisms for creating and destroying connections and for detecting errors. It is similar to a phone. A connection is established between the phones (two ends) and a conversation (transfer of data) takes place.



| Function Call | Description |
|---------------|--|
| Socket() | To create a socket |
| Bind() | It's a socket identification like a telephone number to contact |
| Listen() | Ready to receive a connection |
| Connect() | Ready to act as a sender |
| Accept() | Confirmation, it is like accepting to receive a call from a sender |
| Write() | To send data |
| Read() | To receive data |
| Close() | To close a connection |

Remote Procedure Call (RPC) is a powerful technique for constructing **distributed, client-server based applications**. It is based on extending the conventional local procedure calling so that the **called procedure need not exist in the same address space as the calling procedure**. The two processes may be on the same system, or they may be on different systems with a network connecting them.

When making a Remote Procedure Call:

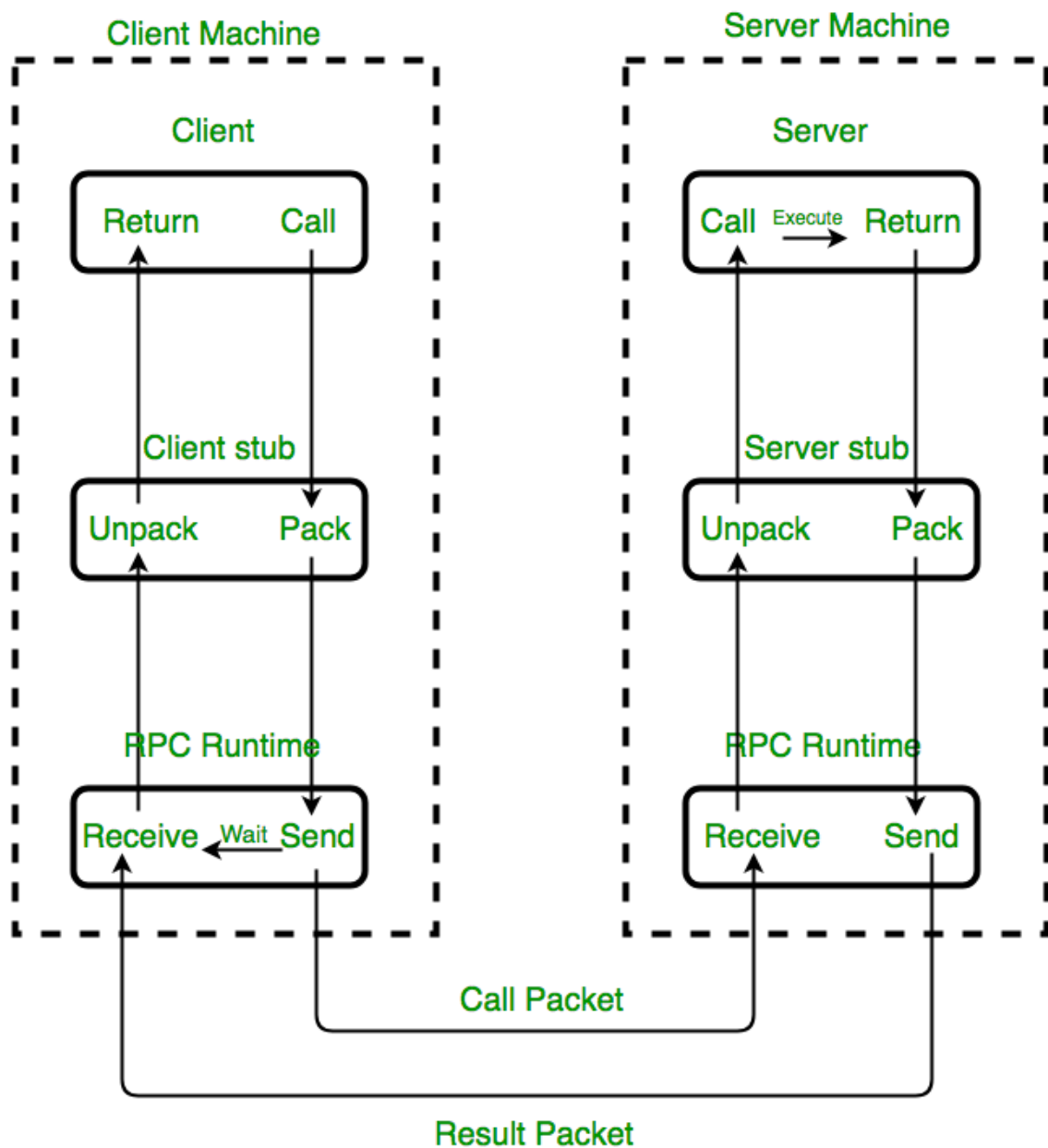


1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.

2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

NOTE: RPC is especially well suited for client-server (e.g. query-response) interaction in which the flow of control **alternates between the caller and callee**. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

Working of RPC



Implementation of RPC mechanism

The following steps take place during a RPC :

1. A client invokes a **client stub procedure**, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub **marshalls(pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a server stub, which **demarshalls(unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (**e.g., via a normal procedure call return**), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

Key Considerations for Designing and Implementing RPC Systems are:

- **Security:** Since RPC involves communication over the network, security is a major concern. Measures such as authentication, encryption, and authorization must be implemented to prevent unauthorized access and protect sensitive data.
- **Scalability:** As the number of clients and servers increases, the performance of the RPC system must not degrade. Load balancing techniques and efficient resource utilization are important for scalability.
- **Fault tolerance:** The RPC system should be resilient to network failures, server crashes, and other unexpected events. Measures such as redundancy, failover, and graceful degradation can help ensure fault tolerance.
- **Standardization:** There are several RPC frameworks and protocols available, and it is important to choose a standardized and widely accepted one to ensure interoperability and compatibility across different platforms and programming languages.
- **Performance tuning:** Fine-tuning the RPC system for optimal performance is important. This may involve optimizing the network protocol, minimizing the data transferred over the network, and reducing the latency and overhead associated with RPC calls.

IPC using Pipes

Program for IPC using pipe() function

The second method for IPC is using the pipe() function. Before writing a program for IPC using pipe() function let us first understand its working.

Syntax:

```
#include<unistd.h>
```

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

pipe() function creates a unidirectional pipe for IPC. On success it return two file descriptors pipefd[0] and pipefd[1]. pipefd[0] is the reading end of the pipe. So, the process which will receive the data should use this file descriptor. pipefd[1] is the writing end of the pipe. So, the process that wants to send the data should use this file descriptor.

The program below creates a child process. The parent process will establish a pipe and will send the data to the child using writing end of the pipe and the child will receive that data and print on the screen using the reading end of the pipe.

//Q. Program to send a message from parent process to child process using pipe()

```
#include<stdio.h>
```

```
#include<unistd.h>
```

```
#include<sys/types.h>
```

```
#include<sys/wait.h>
```

```
int main()
```

```
{
```

```
int fd[2],n;
```

```
char buffer[100];
```

```
pid_t p;
```

```
pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
```

```
p=fork();
```

```
if(p>0) //parent
```

```

{

printf("Parent Passing value to child\n");

write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe

wait();

}

else // child

{

printf("Child printing received value\n");

n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe

write(1,buffer,n);

}

}

```

How it works?

The parent process create a pipe using pipe(fd) call and then creates a child process using fork(). Then the parent sends the data by writing to the writing end of the pipe by using the fd[1] file descriptor. The child then reads this using the fd[0] file descriptor and stores it in buffer. Then the child prints the received data from the buffer onto the screen.

Output

```

baljit@baljit:~/cse325$ ./a.out
Passing value to child
Child printing the received value
hello

```

Figure1: pipe()

output

RPC ISSUES :

Issues that must be addressed:

1. RPC Runtime:

RPC run-time system is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle **binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.**

2. Stub:

The function of the stub is to **provide transparency to the programmer-written application code.**

- **On the client side**, the stub handles the interface between the client's local procedure call and the run-time system, marshalling and unmarshalling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps.
- **On the server side**, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

What is NFS?

Network File System (NFS) is a [networking protocol](#) for distributed [file sharing](#). A [file system](#) defines the way data in the form of files is stored and retrieved from storage devices, such as hard disk drives, solid-state drives and tape drives. NFS is a network file sharing protocol that defines the way files are stored and retrieved from storage devices across networks.

How does the Network File System work?

NFS is a [client-server](#) protocol. An NFS server is a host that meets the following requirements:

- has NFS server software installed;
- has at least one network connection for sharing NFS resources; and
- is configured to accept and respond to NFS requests over the network connection.

An NFS client is a host that meets the following requirements:

- has NFS client software installed;
- has network connectivity to an NFS server;
- is authorized to access resources on the NFS server; and
- is configured to send and receive NFS requests over the network connection.

Thread functions in Linux

In a **Unix/Linux operating system**, the **C/C++ languages** provide the [POSIX thread\(pthread\)](#) standard API(Application program Interface) for all thread

related functions. It allows us to create multiple threads for concurrent process flow

We must include the `pthread.h` header file at the beginning of the script to use all the functions of the pthreads library. To execute the c file, we have to use the `-pthread` or `-lpthread` in the command line while compiling the file.

```
cc -pthread file.c or  
cc -lpthread file.c
```

The **functions** defined in the **pthread library** include:

pthread_create: used to create a new thread

Syntax:

```
int pthread_create(pthread_t * thread,  
                  const pthread_attr_t * attr,  
                  void * (*start_routine)(void *),  
                  void *arg);
```

a. **Parameters:**

- **thread**: pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr**: pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine**: pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg**: pointer to void that contains the arguments to the function defined in the earlier argument

pthread_exit: used to terminate a thread

Syntax:

```
void pthread_exit(void *retval);
```

- b. **Parameters:** This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

pthread_join: used to wait for the termination of a thread.

Syntax:

```
int pthread_join(pthread_t th,  
                void **thread_return);
```

- c. **Parameter:** This method accepts following parameters:

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

pthread_self: used to get the thread id of the current thread.

Syntax:

```
pthread_t pthread_self(void);
```

d.

pthread_equal: compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.

Syntax:

```
int pthread_equal(pthread_t t1,
                  pthread_t t2);
```

e. **Parameters:** This method accepts following parameters:

- t1: the thread id of the first thread
- t2: the thread id of the second thread

pthread_cancel: used to send a cancellation request to a thread

Syntax:

```
int pthread_cancel(pthread_t thread);
```

f. **Parameter:** This method accepts a mandatory parameter **thread** which is the thread id of the thread to which cancel request is sent.

pthread_detach: used to detach a thread. A detached thread does not require a thread to join on terminating. The resources of the thread are automatically released after terminating if the thread is detached.

Syntax:

```
int pthread_detach(pthread_t thread);
```

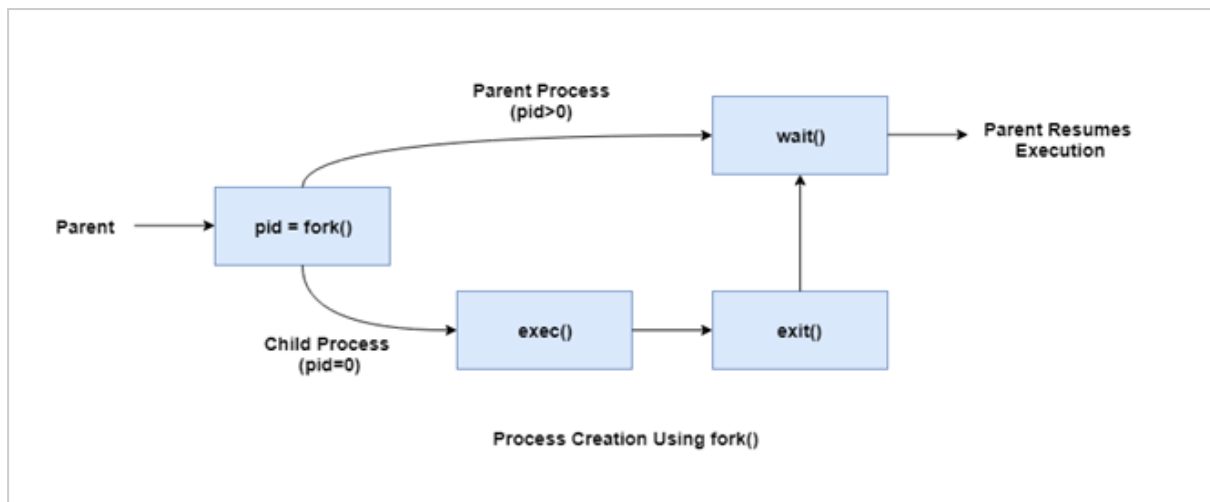
g. **Parameter:** This method accepts a mandatory parameter **thread** which is the thread id of the thread that must be detached.

Process Synchronisation in Linux

Process synchronization in Linux involves providing a time slice for each process so that they get the required time for execution.

The process can be created using the `fork()` command in Linux. The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files and environment strings. However, they have distinct address spaces.

A diagram that demonstrates the fork() command is given as follows –



Orphan Processes

There are some processes that are still running even though their parent process has terminated or finished. These are known as orphan processes. Processes can be orphaned intentionally or unintentionally. An intentionally orphaned process runs in the background without any manual support. This is usually done to start an indefinitely running service or to complete a long-running job without user attention.

An unintentionally orphaned process is created when its parent process crashes or terminates. Unintentional orphan processes can be avoided using the process group mechanism.

Daemon Processes

Some processes run in the background and are not in the direct control of the user. These are known as daemon processes. These processes are usually started when the system is bootstrapped and they terminate when the system is shut down.

Usually the daemon processes have a parent process that is the init process. This is because the init process usually adopts the daemon process after the parent process forks the daemon process and terminates.

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes' access to critical section is controlled by using synchronization techniques. When one

thread starts executing the [critical section](#) (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a [race condition](#) where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

- **Mutex**

1. A Mutex is a lock that we set before using a shared resource and release after using it.
2. When the lock is set, no other thread can access the locked region of code.
3. So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
4. So this ensures synchronized access of shared resources in the code.

- **Working of a mutex**

1. Suppose one thread has locked a region of code using mutex and is executing that piece of code.
2. Now if the scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.
3. Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.
4. Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
5. Mutex lock will only be released by the thread who locked it.
6. So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.

Condition variables

In the context of Linux programming and system administration, "condition variables" refer to a synchronisation mechanism that allows threads to wait for a specific condition to become true before proceeding with their execution.

Condition variables are often used in conjunction with mutexes (mutual exclusion locks) to build more sophisticated thread synchronisation patterns.

Here's an overview of condition variables in Linux:

1. **Purpose and Usage:**

Condition variables are primarily used when one or more threads need to wait for a certain condition to be satisfied before they can continue executing. Instead of busy-waiting (constantly checking if the condition is met), threads can wait efficiently without consuming CPU resources.

2. **Associated Functions:**

In Linux, condition variables are typically used in combination with mutexes to ensure thread safety. The key functions associated with condition variables are:

- `pthread_cond_init()`: Initializes a condition variable.
- `pthread_cond_wait()`: Waits for a condition variable to be signaled. The calling thread releases the associated mutex and waits until another thread signals the condition.
- `pthread_cond_signal()`: Signals a condition variable, waking up one waiting thread.
- `pthread_cond_broadcast()`: Signals a condition variable, waking up all waiting threads.
- `pthread_cond_destroy()`: Destroys a condition variable.

3. **Example:**

Here's a simple example of using condition variables and mutexes in Linux threads:

```
```\n#include <pthread.h>\n#include <stdio.h>\n\npthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;\npthread_cond_t condition = PTHREAD_COND_INITIALIZER;\nint flag = 0;\n\nvoid* thread_function(void* arg) {
```

```

pthread_mutex_lock(&mutex);
while (flag == 0) {
 pthread_cond_wait(&condition, &mutex);
}
pthread_mutex_unlock(&mutex);

printf("Thread: Condition satisfied!\n");
return NULL;
}

int main() {
 pthread_t tid;
 pthread_create(&tid, NULL, thread_function, NULL);

 // Simulate some work...
 sleep(2);

 pthread_mutex_lock(&mutex);
 flag = 1;
 pthread_cond_signal(&condition);
 pthread_mutex_unlock(&mutex);

 pthread_join(tid, NULL);

 pthread_mutex_destroy(&mutex);
 pthread_cond_destroy(&condition);

 return 0;
}

```

In this example, the main thread signals the condition variable after simulating some work, allowing the waiting thread to proceed.

Condition variables help prevent the problem of busy-waiting and improve efficiency in multithreaded programs. However, using them correctly can be complex due to the potential for deadlocks or missed signals. It's important to follow proper synchronization practices and consider using higher-level synchronization constructs when appropriate.

## Signal handling

Signal handling in a multi-threaded environment, such as Linux, requires careful consideration to ensure that signals are managed properly among the threads in the program. Threads share the same process space, which means that signals sent to the process can affect any thread within that process.

### 1. **Signal Types:**

Signals in Linux can be broadly categorized into two types: asynchronous signals and synchronous signals.

- **Asynchronous Signals:** These are delivered to the process as a whole or to a specific thread. Examples include `SIGTERM`, `SIGINT`, and `SIGKILL`.

- **Synchronous Signals:** These are generated as a result of particular program conditions, such as division by zero (`SIGFPE`) or illegal instruction (`SIGILL`).

### 2. **Default Signal Behavior:**

By default, most signals terminate the process. In the context of multi-threaded applications, if a signal is sent to the process, it can potentially affect any thread. This default behavior can be changed using signal masks and handlers.

### 3. **Signal Masking:**

Signal masking involves blocking certain signals in threads to prevent them from being delivered while a thread is performing critical operations. This can be achieved using the `pthread_sigmask()` function.

### 4. **Signal Handlers:**

Each thread can have its own signal handler for specific signals. A signal handler is a function that gets executed when a signal is received. It's important to note that not all signals can be safely handled in a multi-threaded environment. Certain signals, like `'SIGSEGV'` (segmentation fault), should not be handled by individual threads due to their process-wide implications.

#### 5. **`pthread_kill()` Function:**

The `'pthread_kill()'` function is used to send a signal to a specific thread. This function allows you to target a particular thread within the process to receive a signal.

#### 6. **Thread-Specific Signal Handling:**

Linux threads have the option to handle signals independently using the `'pthread_signal()'` function. This function allows you to set up a signal handler that is specific to a particular thread.

#### 7. **Signal Safety:**

When using signal handlers, it's important to write them in a "signal-safe" manner. This means avoiding unsafe functions (those that may be non-reentrant), avoiding memory allocation, and being cautious with shared resources.

#### 8. **Signal Queues:**

Signals are often queued if multiple instances of a particular signal are sent before the signal is handled. This can lead to a signal being received by a thread later when it becomes unblocked.

Managing signal handling in multi-threaded programs requires careful planning and synchronization to ensure that signals are delivered to the appropriate threads and that the program remains stable. It's crucial to understand the implications of signal handling and to design your application with these considerations in mind.

## Driver Concepts

In Linux, a "driver" refers to a software component that facilitates communication between the operating system's kernel and hardware devices. Drivers play a critical role in enabling the operating system to interact with and control various hardware components such as disk drives, network adapters, graphics cards, and more.

### 1. Kernel Space and User Space:

In Linux, the operating system is divided into two main spaces: kernel space and user space. The kernel space is where the core operating system functions and device drivers reside. User space contains applications and processes that interact with the kernel through system calls and other interfaces.

### 2. Character and Block Devices:

Linux drivers are often classified into two main types: character and block devices.

- Character Devices: These devices transfer data one character at a time and do not involve buffering. Examples include keyboards, mice, and serial ports.

- Block Devices: These devices transfer data in fixed-size blocks and often involve buffering. Examples include hard drives, solid-state drives, and CD/DVD drives.

### 3. Loadable Kernel Modules:

Linux supports loadable kernel modules, which are pieces of code that can be dynamically loaded and unloaded into the kernel without requiring a system restart. Drivers are often implemented as loadable modules. The ``insmod`` and ``modprobe`` commands are used to load modules.

#### 4. Kernel Data Structures for Drivers:

Linux provides various data structures and APIs that drivers use to interact with the kernel. These include structures for character and block device registration, interrupt handling, memory management, and more.

#### 5. File Operations and Virtual Filesystem:

Linux drivers interact with user space through the virtual filesystem (VFS). Drivers define file operations, such as ``open()``, ``read()``, ``write()``, and ``close()``, which allow user space applications to access the driver's functionality as if they were working with regular files.

#### 6. Interrupt Handling:

Hardware devices often generate interrupts to notify the CPU that they need attention. Drivers handle interrupts to manage asynchronous events from devices. Interrupt handlers are usually short and time-critical to ensure that devices are serviced promptly.

#### 7. Memory Mapping:

Some drivers allow memory mapping, where certain regions of device memory can be mapped into user space, providing more efficient data transfer between user space and the device.

#### 8. Bus and Device Model:

Linux uses a hierarchical bus and device model to organize and manage hardware devices. Devices are organized by buses (e.g., PCI, USB), and the driver model allows drivers to be associated with specific device types.

#### 9. Driver Interfaces:

Linux drivers interact with the kernel through various interfaces, including the device driver API, the sysfs filesystem for exposing device information, and the ioctl system call for additional control operations.

## 10. Device Tree and ACPI:

On architectures like ARM, device information is often described using a device tree, which provides a way to describe hardware in a machine-readable format. On x86 systems, ACPI (Advanced Configuration and Power Interface) is used for power management and device configuration.

Developing Linux drivers requires a good understanding of the kernel's internals, programming in C, and knowledge of the specific hardware you're working with. The Linux kernel source code, documentation, and various online resources provide valuable information for driver development.

### Linux Hardware Interface

# Linux hardware interface



Author: Elizabeth Watkins Date: 2023-05-07

When it comes to devices on standard buses, determining the appearance of ISA devices (mostly serial/parallel ports) requires a trial and error approach, reading a small set of typical port addresses. In the case of devices connected to PCI buses,

the PCI configuration space provides a standard that allows for the implementation of arch independent methods for enumeration and initialization."".

Table of contents

- [Linux hardware interface](#)
- [How the OS interfaces between the user, apps , hardware?](#)
- [Original Interface](#)
- [What is a computer interface?](#)
- [How does a device interface work in a driver?](#)
- [What is a human interface device?](#)
- [What is a device interface class in Linux?](#)

## Linux hardware interface

Question:

I'm working on a hardware project that involves running a GNU/Linux OS (specifically, an ARM-based one), and I have a question.

In what manner does the Linux kernel precisely determine the type of hardware connected to the CPU? I am referring to how it identifies a RAM or a drive, for instance.

For network interfaces, determining the specific Ethernet NICs and WiFi transceivers and understanding their hardware connections (such as using a multiplexer, I2C, SPI, etc.) poses a challenge.

Solution 1:

The architecture will be the determining factor for all these tasks that require a low level of expertise.

The BIOS is a valuable resource for a successful start on the most common (x86/IBM PC) platform. When obtaining one, be sure to check your bootlog as it will initiate queries to the BIOS.

*BIOS-provided physical RAM map:*

As you continue scrolling, you may come across something similar to:

*BIOS-e820 includes ACPI data in the memory range from 0x00000000cff80000 to 0x00000000cff8dfff.*

The BIOS provides tables that the OS can rely on for obtaining additional information about peripherals, although it is commonly known that BIOSes can be flawed to varying degrees.

ARM SoCs, on the contrary, are equipped with unique peripheral support from each vendor, typically in a closed-source manner. Additionally, there has been no



consensus on a universal BIOS equivalent among these vendors. This scenario has resulted in notable statements from Linus Torvald.

*I wish for the SoC ARM designers to meet an excruciating fate. [...] Ugh, Dealing with this entire ARM situation is an absolute nightmare.*

Best of luck if your project relies on such hardware; you may need to resort to brute-forcing memory ranges to achieve the desired outcome.

---

Next, there is architecture-independent information data, specifically pertaining to devices on standard buses.

To identify the appearance of ISA devices, which are primarily serial/parallel ports nowadays, it is necessary to use a trial and error approach by reading a small set of common port addresses.

The PCI configuration space, which applies to devices connected to PCI buses, allows for the utilization of arch-independent techniques to enumerate and initialize them.

---

If your hardware lacks standardized probing methods, you will need to find your own solution to address the question, as advised by @telcoM.

*On ARM, it appears to be the norm that the hardware's designers (or reverse-engineers, if applicable) must delineate it using device tree data. This data is subsequently loaded by the bootloader specific to the hardware, alongside the kernel file and potentially the initramfs file. Therefore, if your hardware project employs an architecture lacking a standardized, autoprobeable main bus such as PCI or PCIe, it will be necessary for you, as the hardware designer, to furnish that information for the Linux kernel.*

#### Solution 2:

The x86 architecture encompasses various standard PC components, such as PCI, PCI-E, RAM, etc., that are enumerated by the Linux kernel during the boot process. Certain buses, like USB or SATA, allow for the attachment and detachment of devices on the go. It's important to note that your question is not specific to Linux/Unix.

- The content that requires rephrasing is a link to a webpage titled "[Operating Systems - Input/Output Systems](#)".
- The link provided is from Stack Overflow and it discusses how the operating system determines all the hardware at boot time. ""

Using Driver-Defined Interfaces, Drivers can define device-specific interfaces that other drivers can access. These driver-defined interfaces can consist of a set of callable routines, a set of data structures, or both. The driver typically provides pointers to these routines and structures in a driver-defined interface structure, which the driver ...

Tags:

[interfaces between the user apps hardware](#)

## What is HARDWARE INTERFACE DESIGN? What does

<http://www.theaudiopedia.com> What is HARDWARE INTERFACE DESIGN? What does HARDWARE INTERFACE DESIGN mean? HARDWARE INTERFACE DESIGN meaning - HARD

## How the OS interfaces between the user, apps , hardware?

---

A contemporary computer is comprised of the following elements –.

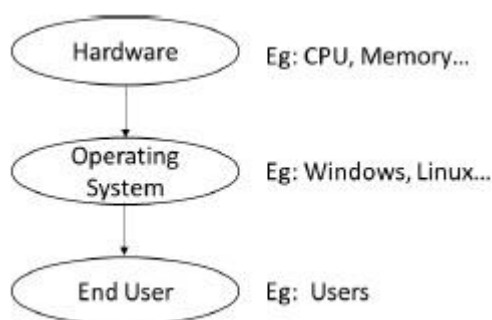
- One or more processors
- Main memory
- Disks
- Printers
- 
- Various input/output devices.

In order to oversee these various components, a software layer known as the Operating System (OS) is needed within the computer system.

An Operating System serves as a mediator or interface between the computer user and the hardware.

The operating system is vital software for computer systems as it enables the user to run application programs.

Below is the configuration of an operating system.



The operating system functions as a governing body in a nation, similar to how a government controls, monitors, and assists the country. Similarly, the operating system oversees all computer components and ensures the proper execution of programs. Additionally, it establishes regulations to prevent conflicts arising from multiple users accessing the same resource.

Operating System - I/O Hardware, Operating System - I/O Hardware. One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped

screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

## Original Interface

The original interface requires specific joystick minidriver callbacks. Before processing the SYS\_DYNAMIC\_DEVICE\_INIT message, the four joystick-specific callbacks should be registered with the VJoyD VJOYD\_Register\_Device\_Driver service. The polling routine should be pointed to by EAX, the configuration handler by EBX, the capabilities callback by ECX, and the identification routine by EDX. An example sequence for registering a joystick minidriver is provided below:

```
Mov eax, offset32 _PollRoutine@8
```

```
Mov ebx, offset32 _CfgRoutine
```

```
Mov ecx, offset32 _HWCapsRoutine@8
```

```
Mov edx, offset32 _JoyIdRoutine@8
```

```
VxDcall VJOYD_Register_Device_Driver
```

Apart from registering, a minidriver has the ability to carry out additional initialization during this period. The joystick minidriver model does not necessitate any particular actions in response to SYS\_DYNAMIC\_DEVICE\_EXIT, although the VxD may still utilize it for concluding internal cleanup.

### Configuration Manager Callback

```
CONFIGRET CM_HANDLER CfgRoutine(CONFIGFUNC cfFuncName,
```

```
SUBCONFIGFUNC scfSubFuncName, DEVNODE dnToDevNode,
ULONG dwRefData, ULONG ulFlags);
```

Detailed guidelines on interacting with the configuration manager can be found in the Plug and Play Environment section of the Windows Me portion of the [Windows Driver Development Kit \(DDK\)](#), which was released prior to the Windows Driver Kit (WDK).

The configuration manager callback is only passed to the minidrivers if they are loaded at the time VJoyD receives the callback. However, this leads to a significant problem with the current implementation of the callback. VJoyD receives callbacks for the device nodes it is associated with during system boot when the device nodes are enumerated, during runtime when the configuration manager sends a message for a special event like device reconfiguration, and during system shutdown. As a result, only the devices that were selected during the previous boot are loaded in time to receive these messages. This limitation restricts the number of functions that can be performed when the callback is invoked, as a user may boot the system, configure a joystick, play games, de-configure the joystick, and shut down without triggering this callback.

For devices that don't require hardware resources, there's no problem. However, devices that do need these resources have multiple choices: they can operate solely after being configured during boot, they can dynamically assign resources from the configuration manager, or they can seek their allocations in the registry for their device node and request the necessary information.

Based on the information provided, the driver is correctly initialized either upon its initial loading and receiving the SYS\_DYNAMIC\_DEVICE\_INIT message, or during the first iteration of the polling routine. Likewise, resources should be freed when a SYS\_DYNAMIC\_DEVICE\_EXIT message is received.

One concern is that all configuration manager callbacks for currently serviced joystick devices are broadcasted to all loaded minidrivers. Nevertheless, the dnToDevNode parameter can be utilized to search for the device identifier, and you can verify if this driver is capable of handling these devices.

## **Polling Callback**

```
int stdcall PollRoutine(int type, LPJOYOEMPOLLDATA poj_d);
```

The polling routine is triggered either when an application directly calls joyGetPos or joyGetPosEx, or when Windows 95/98/Me periodically calls joyGetPos for an application that has previously called joySetCapture. The minidriver can be certain that an application needs data from it only when it first receives a call on its poll callback. All other callbacks are invoked for all devices currently in the list of 16 available Joysticks, regardless of whether any application has requested data.

The potential outcomes include JOY\_OEMPOLLRC\_YOUPOLL, JOY\_OEMPOLLRC\_FAIL, and JOY\_OEMPOLLRC\_OK. Except for a poll that only

involves buttons, the result is converted to either JOYERR\_NOERROR or JOYERR\_UNPLUGGED before being sent back to the application. It is always assumed that a button poll will be successful.

JOY\_OEMPOLLRC\_FAIL is the return value when the polled device does not respond or when VJoyD requests a poll that the minidriver cannot process. In the first scenario, it is important to avoid reporting a device failure unless it is truly failed, as this would prompt the application to halt and prompt the user to check the device connections. For example, if the first poll fails, a call to joySetCapture will also fail. It is best to only report device failure when user intervention is necessary or when an unrecoverable device error is detected. For instance, if a device uses a packet-based protocol for communication, it should attempt a retry before determining the poll as failed if one packet is invalid.

It is crucial to acknowledge genuine device failure and eventually provide an error message; otherwise, the application will not be able to determine if user intervention is necessary. To prevent excessive delays, the driver should limit the meaning of the returned data, the number of allowed polls, or the time allowed for error recovery if old or inaccurate data is returned. Whenever a minidriver is asked to poll a device it doesn't control, or to perform a poll type or subtype it cannot handle, failure should always be the appropriate response.

If the standard analog poll produces accurate outcomes, JOY\_OEMPOLLRC\_YOUPOLL can be utilized.

In a regular case, when the requested data is filled in the JOYINFOEX structure, you should return JOY\_OEMPOLLRC\_OK.

If you choose to utilize polling for device initialization, caution must be exercised. Applications can initially poll to verify the device's availability by examining the return code. Subsequent polls can then be conducted without error checks, similar to the functionality of joySetCapture. Ideally, all initialization and the initial poll should be completed. In cases where device setup takes up to one second, it is adequate to confirm the presence and functionality of the device and provide inert data for the initial poll.

The application's request is validated and the necessary data is copied into its structure by the API and system drivers. The minidriver is not supposed to duplicate this functionality. Nevertheless, it is possible for one process to poll a device while another process is still in the middle of a poll. The outcome of this situation is similar to having different samples reporting the two axes. If needed, a minidriver can synchronize its own processes.

The return values are determined by the "type" parameter and the device it supports. It is always necessary to return the buttons and button number. If any axes are requested and there is no way to indicate that a POV poll is needed, this value should be returned. If the device does not support a POV, set the value to POV\_UNDEFINED. In the JOYPOLLDATA structure (or VJPOLLDATA structure in DirectX 5.0 and later), the value for a single axis poll is returned in the dwX member. For a request with multiple axes, if the number of axes requested is odd, the do\_other member of the JOYOEMPOLLDATA structure specifies whether the last axis is returned in place or for the following axis. For example, in a three-axis poll, the member specifies whether the axes returned are X, Y, Z or X, Y, R.

Below is a categorized list of potential return values. It is important to note that when dealing with an odd number of axes, the "do\_other" member must be decoded to determine its return value. Additional data can be included as needed.

- 
- JOY\_OEMPOLL\_GETBUTTONS: Nothing extra.
- 
- JOY\_OEMPOLL\_POLL1: The axis specified in the do\_other member is returned in the dwX member.
- 
- JOY\_OEMPOLL\_POLL2: The X and Y axes are returned in the dwX and dwY members.
- 
- JOY\_OEMPOLL\_POLL3: The X and Y axes are returned in the dwX and dwY members.  
If the do\_other member is nonzero, the R axis is returned in the dwR member. Otherwise, the Z axis is returned in the dwZ member.
- 
- JOY\_OEMPOLL\_POLL4: The X, Y, Z and R axes are returned in the dwX , dwY , dwZ , and dwR members respectively.
- 
- JOY\_OEMPOLL\_POLL5: The X, Y, Z and R axes are returned in the dwX , dwY , dwZ , and dwR members respectively.  
If the do\_other member is nonzero, the V-axis is returned in the dwR member. Otherwise, the U-axis is returned in the dwZ member.
- 
- JOY\_OEMPOLL\_POLL6: The X, Y, Z, R, U and V axes are returned in the dwX , dwY , dwZ , dwR , dwU , and dwV members respectively.

In DirectX 3.0, the non-poll type of JOY\_OEMPASSDRIVERDATA was introduced. This type includes a DWORD called do\_other, which is passed by an application. The purpose of this DWORD is to allow the implementation of minidriver-defined functions, with a specific emphasis on facilitating custom setup applications to send device-specific commands and configuration information once the minidriver has been fully identified.

For any unsupported types, a minidriver is expected to return JOY\_OEMPOLLRC\_FAIL.

The axis data is returned in double words, but it is recommended to limit the range of axis values to approximately 10-bit values when using the standard joystick control panel configuration as the only means of configuration. This restriction helps prevent confusion since the user cannot easily perceive the actions. To ensure compatibility with existing applications, the minidriver should return the same axes as an analog joystick, following the same movement patterns. For instance, X-axis movement would be minimized at the left, moving from left to right.

To indicate the direction of an activated POV hat, use the angle in degrees multiplied by 100. In this representation, 0 corresponds to forward, 9000 to right, 18000 to backwards, and 27000 to left.

Since the poll routine call serves as the sole confirmation that the device is still being utilized, minidrivers utilizing shared resources, like communication ports, should maintain a record of their most recent usage. If the amount of time the minidrivers are being used becomes significant, they should pause to check the device and release the resources in case the user has finished using the device and now intends to use the resource for a different purpose. This is especially crucial when there are instances of communication errors, as this may suggest that the device has been disconnected.

## Hardware Capabilities Callback

```
int __stdcall HWCapsRoutine(int joyid, LPJOYOEMLHWCAPS pjhwc);
```

The function HWCapsRoutine is invoked whenever the device's hardware capabilities are requested. Specifically, before returning from VJOYD\_Register\_Device\_Driver, VJoyD makes a call to HWCapsRoutine. Therefore, it is crucial for the driver to finish any initialization tasks that this call depends on before registering the device. These values usually remain constant. For instance, in the case of a device with four buttons and three axes, where the third axis can provide either a throttle or a rudder value, the following configuration is suitable:

```
pjhwc->dwMaxButtons = 4; /* This should always be the number of
buttons */

pjhwc->dwMaxAxes = 4; /* The largest axis number which may be
requested */

pjhwc->dwNumAxes = 3; /* The number of axes the device has */
```

## Joystick Identification Callback

```
int __stdcall JoyIdRoutine(int joyid, BOOL used);
```

VJoyD invokes the JoyIdRoutine function whenever a user configures or de-configures a joystick from the available 16 joysticks. If the requested ID in joyid is supported by the minidriver, the JoyIdRoutine function will return a non-zero value. If the minidriver is unable to support the requested ID, the function will return a zero value.

Whenever a change is made and the driver is updated by calling joyConfigChanged, VJoyD iterates through all 16 devices starting with JOYSTICKID1. It sets all devices to unused and then iterates through them again to set the ones required by the system. This callback usage for initialization can be problematic during control panel operations due to the potentially large number of calls involved. This is especially

true if the call is made before the system boot is complete, when other services are unavailable.

Minidrivers handling callbacks for multiple devices should try to maintain a consistent mapping between joystick identifiers and physical devices to prevent confusion among users. This can be implemented effortlessly within a single session, but may not be essential across reboots.

To avoid repetition, the VJoyD VJOYD\_Register\_Device\_Driver service should be used to register the four joystick-specific callbacks. These callbacks, namely the polling routine (pointed by EAX), configuration handler (pointed by EBX), capabilities callback (pointed by ECX), and identification routine (pointed by EDX), need to be registered before processing the SYS\_DYNAMIC\_DEVICE\_INIT message. An example of the registration sequence for a joystick minidriver is provided below.

```
Mov eax, offset32 _PollRoutine@8
```

```
Mov ebx, offset32 _CfgRoutine
```

```
Mov ecx, offset32 _HWCapsRoutine@8
```

```
Mov edx, offset32 _JoyIdRoutine@8
```

```
VxDcall VJOYD_Register_Device_Driver
```

Apart from registration, a minidriver can execute additional initialization tasks during this period. The joystick minidriver model does not mandate any specific actions upon SYS\_DYNAMIC\_DEVICE\_EXIT, although the VxD may still utilize it for internal clean-up purposes.

What are Good and Bad Interface Designs?, Those exchanges can be between software, computer hardware, peripheral devices, etc. Now let us discuss the good



interface and bad interface. A good interface design is user friendly. This is because it is easy to navigate, easy to use, easy to understand, interactive and effective. A good design should always ...

## Writing a simple Character Device Driver

Writing character drivers in Linux involves creating a module that interacts with a character device. A character device is a type of device that deals with data on a character-by-character basis, such as a terminal or serial port. Character drivers communicate with user-space applications by using read and write operations. Here's a step-by-step guide to writing a basic character driver in Linux:

### 1. Set Up Your Development Environment:

Ensure you have a Linux machine with necessary development tools, headers, and kernel sources installed.

### 2. Create the Character Driver Source Code:

Create a new directory for your driver code:

```
mkdir char_driver
cd char_driver
```

Create the source code file for your character driver, such as ``mychar_driver.c``.

### 3. Include Necessary Headers:

In your source code file, include necessary kernel headers:

c

```
#include <linux/init.h>
```

```
#include <linux/module.h>
```

```
#include <linux/kernel.h>
```

```
#include <linux/fs.h> // File operations structure
```

```
#include <linux/cdev.h> // Char device structure
```

```
#include <linux/uaccess.h> // User-space access functions
```

#### 4. Define Your Device Data:

Define the major and minor numbers for your device:

c

```
#define MAJOR_NUM 0 // Use 0 to have the kernel allocate a major number
```

```
#define MINOR_NUM 0 // Use 0 for a single device
```

#### 5. Define File Operations:

Define the file operations that your driver will support (e.g., open, close, read, write):

c

```
static struct file_operations mychar_fops = {
```

```
 .owner = THIS_MODULE,
```

```
 .open = mychar_open,
```

```
 .release = mychar_release,
```

```
 .read = mychar_read,
```

```
 .write = mychar_write,
```

```
};
```

## 6. Implement File Operation Functions:

Implement the functions declared in the file operations structure (``mychar_open``, ``mychar_release``, ``mychar_read``, ``mychar_write``). These functions define how your driver interacts with user-space applications.

## 7. Initialize and Register the Char Device:

In your module's initialization function, allocate and register the character device:

c

```
static int __init mychar_init(void) {
 // Register the character device

 int ret = alloc_chrdev_region(&dev_num, MINOR_NUM, 1,
 "mychar_device");

 if (ret < 0) {
 printk(KERN_ALERT "Failed to allocate character device region\n");
 return ret;
 }

 // Initialize the character device
 cdev_init(&mychar_cdev, &mychar_fops);
 mychar_cdev.owner = THIS_MODULE;

 // Add the character device to the system
 ret = cdev_add(&mychar_cdev, dev_num, 1);
 if (ret < 0) {
 unregister_chrdev_region(dev_num, 1);
```

```

 printk(KERN_ALERT "Failed to add character device\n");
 return ret;
 }

 printk(KERN_INFO "Character driver initialized\n");
 return 0;
}

```

## 8. Cleanup and Unregister:

In your module's cleanup function, unregister and release any resources acquired:

```

c
static void __exit mychar_exit(void) {
 cdev_del(&mychar_cdev);
 unregister_chrdev_region(dev_num, 1);
 printk(KERN_INFO "Character driver exited\n");
}

```

## 9. Module Initialization and Cleanup:

Register your initialization and cleanup functions using the module macros:

```

c
module_init(mychar_init);
module_exit(mychar_exit);

```

## 10. Build and Load the Module:

Compile your module using the appropriate Makefile or build script, and then load it using ``insmod``:

```
make
```

```
sudo insmod mychar_driver.ko
```

## 11. Test the Character Driver:

Create a user-space application that interacts with your character driver using standard file operations (``open``, ``read``, ``write``, ``close``).

Remember that this is a basic example. Real-world character drivers may involve more complex interactions, locking mechanisms, and error handling. Additionally, ensure you have a solid understanding of Linux kernel programming, module management, and driver development practices before creating production-level drivers.

## **Interrupt handling**

Always test your driver in a controlled environment, as kernel errors can lead to system instability.

Interrupt handling in Linux is a crucial mechanism that allows the kernel to respond to hardware events and asynchronous requests from hardware devices. Interrupts enable efficient multitasking by allowing the CPU to execute normal tasks while still being able to handle time-sensitive events without delay. Here's an overview of how interrupt handling works in Linux:

### 1. Hardware Interrupts:

Hardware devices, such as peripherals or external events, generate hardware interrupts when they require attention from the CPU. These interrupts are usually triggered by hardware events like I/O completion, timer expiration, or user actions like pressing a key.

## 2. Interrupt Requests (IRQs):

Each hardware device is assigned a unique Interrupt Request (IRQ) line, which is a hardware signal line used to indicate that an interrupt has occurred. The number of available IRQ lines is limited, and the kernel must manage these resources efficiently.

## 3. Interrupt Service Routine (ISR):

When an interrupt occurs, the CPU is interrupted from its current task, and control is transferred to the Interrupt Service Routine (ISR) associated with the triggered IRQ. The ISR is a kernel function that handles the specific interrupt event for that device.

## 4. Interrupt Context:

The ISR runs in a special context known as the "interrupt context." This context has certain limitations compared to regular kernel or user space contexts. For example, sleeping is not allowed in the interrupt context, as it could cause deadlocks.

## 5. Bottom Halves and Tasklets:

Due to the limitations of the interrupt context, lengthy operations or operations that can sleep are often deferred to "bottom halves." Tasklets and workqueues are kernel mechanisms that allow certain operations to be deferred and executed in a more relaxed context, ensuring that critical work is not performed in the interrupt context.

## 6. Interrupt Handling Mechanism:

Linux uses the Interrupt Descriptor Table (IDT) to manage interrupt handling. The IDT is an array of function pointers, each pointing to an ISR. When an interrupt occurs, the CPU consults the IDT to find the appropriate ISR and calls it.

## 7. Registering Interrupt Handlers:

In Linux, device drivers register their interrupt handlers using functions like `request_irq()`. This associates the ISR with the corresponding IRQ line and provides information about how to handle the interrupt, such as flags for interrupt sharing and edge/level triggering.

## 8. Handling Shared Interrupts:

Many devices may share the same IRQ line. In such cases, the kernel must carefully manage these shared interrupts to prevent conflicts and ensure that the appropriate ISR is invoked for each device. This often involves checking the interrupt source within the ISR itself.

## 9. Interrupt Disabling:

During the execution of an ISR, hardware interrupts for the same IRQ line are usually disabled to prevent nested interrupt handling. Once the ISR completes, interrupts are re-enabled, allowing other pending interrupts to be serviced.

## 10. Interrupt Bottom Halves and Tasklets:

For handling deferred work, Linux provides tasklets and workqueues. These mechanisms allow certain operations to be scheduled to run in a safer context, avoiding potential issues with long-running operations in the ISR.

Interrupt handling is a complex topic, and writing reliable and efficient interrupt handlers requires a deep understanding of the Linux kernel's internal workings, as well as proper synchronization and error handling mechanisms. It's crucial to follow best practices and guidelines provided by the Linux kernel documentation and resources to avoid issues like race conditions and deadlocks.

## Interfacing with hardware

Interfacing with hardware in Linux involves interacting with various hardware devices, such as peripherals, sensors, and other external components, using appropriate drivers

and APIs provided by the Linux kernel. Here's a general overview of how to interface with hardware in a Linux environment:

### 1. Device Drivers:

Linux provides a unified mechanism for interacting with hardware through device drivers. Device drivers are kernel modules that enable the operating system to communicate with specific hardware devices. They abstract the low-level details of hardware communication and provide a standardised interface for applications to access hardware functionality.

### 2. Kernel Modules:

Device drivers are typically implemented as kernel modules, which are dynamically loadable code that can be inserted and removed from the running kernel without rebooting the system. These modules interact with the hardware and provide high-level interfaces for user-space applications to use.

### 3. Device Filesystem (`/dev`):

Linux represents hardware devices as files within the `/dev` directory using special files or device nodes. User-space applications can interact with hardware by performing standard file operations (read, write, open, close) on these device nodes.

### 4. Character and Block Devices:

Hardware devices are classified as either character or block devices. Character devices deal with data on a character-by-character basis, like terminals and serial ports. Block devices manage data in fixed-size blocks and are often used for storage devices like hard drives and SSDs.

### 5. sysfs, procfs, and debugfs:

Linux provides filesystem interfaces like sysfs, procfs, and debugfs, which expose kernel and device information to user-space applications. sysfs is commonly used for querying and configuring device attributes, while procfs and debugfs provide information for debugging and monitoring.

### 6. IOCTL (Input/Output Control):



Device drivers often support IOCTL commands, which are custom commands that applications can use to communicate specific instructions or requests to the device driver. This allows user-space applications to perform various operations beyond simple read and write.

## 7. User-Space Libraries and APIs:

To simplify hardware interaction, Linux provides user-space libraries and APIs that abstract the complexities of low-level hardware access. For example, the GPIO (General Purpose Input/Output) interface provides functions to control GPIO pins on embedded systems.

## 8. Memory-Mapped I/O:

In some cases, memory-mapped I/O is used to directly access hardware registers as if they were memory locations. This provides low-latency access to hardware components. However, this technique requires careful management to avoid issues like bus conflicts and data corruption.

## 9. Interrupt Handling:

As mentioned in a previous response, interrupt handling is critical for hardware interaction. Device drivers use interrupt handlers to respond to hardware events and asynchronous requests from hardware devices.

## 10. Hardware Abstraction Layer (HAL):

In certain cases, hardware vendors provide Hardware Abstraction Layer (HAL) libraries or frameworks that encapsulate the low-level details of hardware access. These libraries provide a consistent interface across different platforms and help abstract hardware-specific intricacies.

## 11. Device Tree (DT) and ACPI:

On embedded systems, the Device Tree (DT) or ACPI (Advanced Configuration and Power Interface) tables describe the hardware configuration to the kernel. These structures help the kernel identify and configure hardware components correctly.

## 12. Cross-Compilation:

When developing for embedded systems or platforms with limited resources, cross-compilation is often used to build device drivers and applications on a development machine targeting the target hardware architecture.

Interfacing with hardware in Linux involves a combination of kernel-level programming, user-space API usage, understanding of device driver development, and knowledge about the specific hardware components you are working with. It's crucial to consult the Linux kernel documentation, relevant hardware datasheets, and resources specific to your hardware platform for accurate and up-to-date information.

**Looping Statements in Shell Scripting:** There are total 3 looping statements which can be used in bash programming

1. while statement
2. for statement
3. until statement

To alter the flow of loop statements, two commands are used they are,

1. break
2. continue

Their descriptions and syntax are as follows:

**while statement:** Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated

**that**

```
while <condition>
do
 <command 1>
 <command 2>
 <etc>
done
```

**for statement:** The for loop operate on lists of items. It repeats a set of commands for every item in a list.

Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

**Syntax:**

```
for <var> in <value1 value2 ... valuen>
do
 <command 1>
 <command 2>
```

```
 <etc>
done
```

**until statement:** The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

**Syntax:**

```
until <condition>
do
 <command 1>
 <command 2>
 <etc>
done
```

```
#Start of for loop
for a in 1 2 3 4 5 6 7 8 9 10
do
 # if a is equal to 5 break the loop
 if [$a == 5]
 then
 break
 fi
 # Print the value
 echo "Iteration no $a"
done
```

```
a=0
-lt is less than operator

#Iterate the loop until a less than 10
while [$a -lt 10]
do
 # Print the values
 echo $a

 # increment the value
 a=`expr $a + 1`
done
```

```
a=0
-gt is greater than operator

#Iterate the loop until a is greater than 10
until [$a -gt 10]
do
 # Print the values
 echo $a
```

```
 # increment the value
 a=`expr $a + 1`
done
```

```
COLORS="red green blue"
```

```
the for loop continues until it reads all the values from the COLORS
```

```
for COLOR in $COLORS
do
 echo "COLOR: $COLOR"
done
```

Example 8: Checking for user input

```
CORRECT=n
while ["$CORRECT" == "n"]
do
 # loop discontinues when you enter y i.e.e, when your name is correct
 # -p stands for prompt asking for the input
 read -p "Enter your name:" NAME
 read -p "Is ${NAME} correct? " CORRECT
done
```

## Positional Parameters

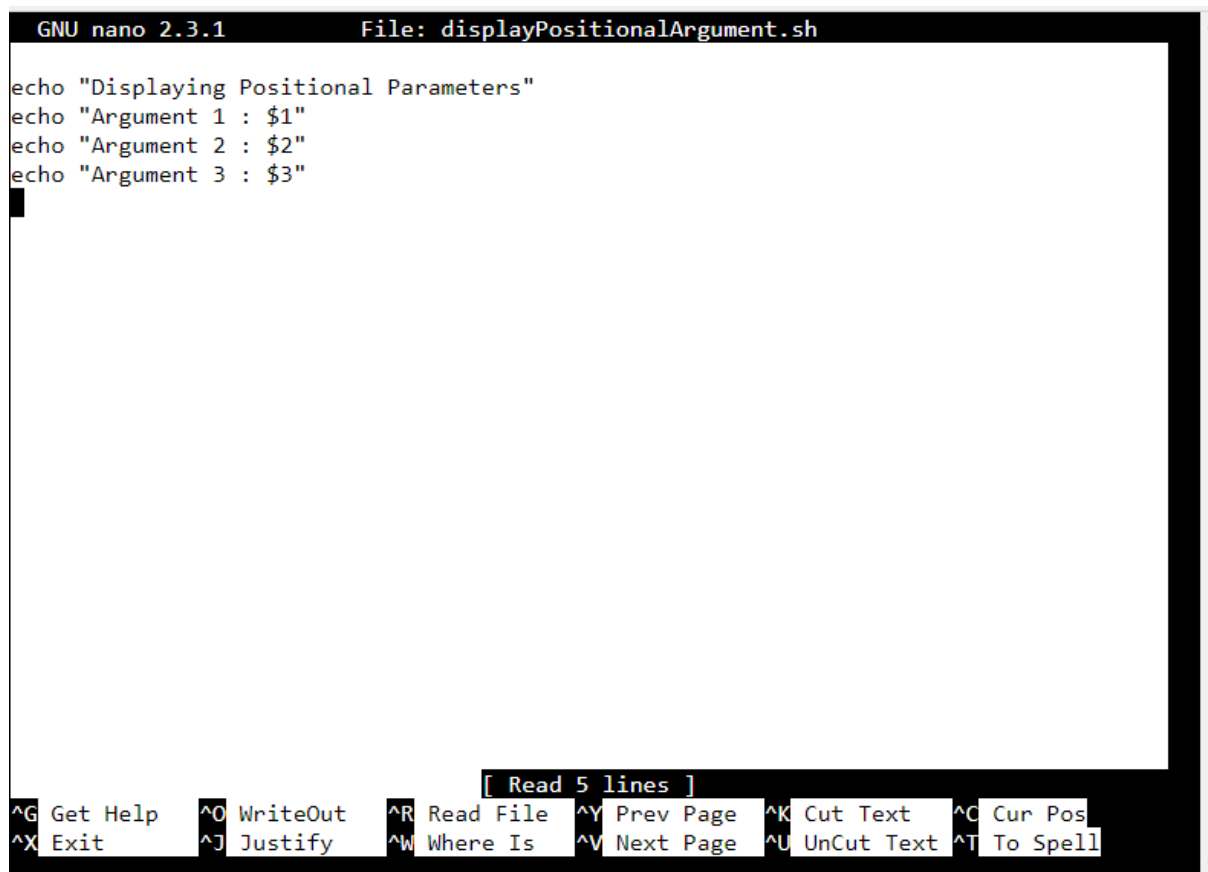
Command-line arguments are passed in the positional way i.e. in the same way how they are given in the program execution. Let us see with an example.

Create a shell program that can display the command line arguments in a positional way. “Nano” editor is used to create the shell program”

```
GNU nano 2.3.1 File: displayPositionalArgument.sh

echo "Displaying Positional Parameters"
echo "Argument 1 : $1"
echo "Argument 2 : $2"
echo "Argument 3 : $3"

```



*Nano editor is used to coding the shell script*

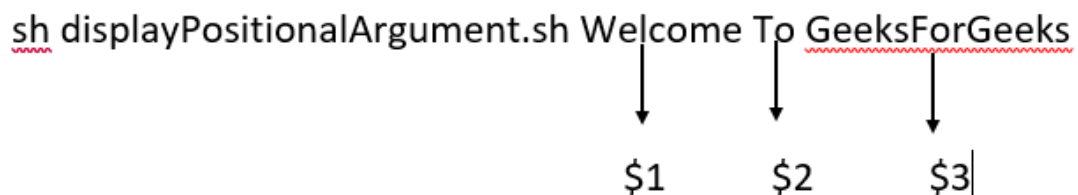
The program can be executed by using the “sh” command.

```
sh <script filename> arg1 arg2 arg3
```

So, our code of execution will become

```
sh displayPositionalArgument.sh Welcome To GeeksForGeeks
```

Diagrammatic representation of the above code:



*Diagrammatic representation*

Always the first argument starts after the <script filename>. <script filename> will be in the 0th location, We need to take positional arguments after the <script filename>. Hence in the above example

\$1-> "Welcome"

\$2-> "To"

\$3-> "GeeksForGeeks"

**Note:** We can pass n number of arguments and they are identified by means of their position.

The output of the above code :

```
[rajraj@139-162-5-218 ~]$ sh displayPositionalArgument.sh Welcome To GeeksForGeeks
Displaying Positional Parameters
Argument 1 : Welcome
Argument 2 : To
Argument 3 : GeeksForGeeks
[rajraj@139-162-5-218 ~]$
```

*Output*

Some special variables are also to be noted while handling command-line arguments.

Special Variable	Special Variable's details
\$1 ... \$n	Positional argument indicating from 1 .. n. If the argument is like 10, 11 onwards, it has to be indicated as \${10},\${11}
\$0	This is not taken into the argument list as this indicates the "name" of the shell program. In the above example, <b>\$0</b> is <b>"displayPositionalArgument.sh"</b>

\$@	Values of the arguments that are passed in the program. This will be much helpful if we are not sure about the number of arguments that got passed.
\$#	Total number of arguments and it is a good approach for loop concepts.
\$*	In order to get all the arguments as double-quoted, we can follow this way
\$\$	To know about the process id of the current shell
\$? and \$!	Exit status id and Process id of the last command

## Using Flags

Arguments can be passed along with the flags. The arguments can be identified with a single letter having – before that. A single letter can be meaningful and here let us take -1, -2, and -3.

We need to use **getopts** function to read the flags in the input, and **OPTARG** refers to the corresponding values:

```
GNU nano 2.3.1 File: usingFlags.sh

while getopts 1:2:3: flag
do
 case "${flag}" in
 1) websitename=${OPTARG};;
 2) postname=${OPTARG};;
 3) shares=${OPTARG};;
 esac
done
echo "WebsiteName : $websitename";
echo "PostName : $postname";
echo "Shares : $shares";
```

*Program2 using flags*

The above program can be executed as

`sh usingFlags.sh -1 'GeeksForGeeks' -2 'JavaProgramming' -3 100`

## Output

```
[rajraj@139-162-5-218 ~]$ sh usingFlags.sh -1 'GeeksForGeeks' -2 'JavaProgramming' -3 100
WebsiteName : GeeksForGeeks
PostName : JavaProgramming
Shares : 100
```

*Output*

## Using Loops with \$@ – Loop Constructs

Though positional parameters help to send as command-line arguments, if there are more arguments, we cannot count and use them. Instead by using \$@, we can achieve that. It is nothing but the array of all the parameters passed. Iterating over a for loop, we can get that.



```

GNU nano 2.9.1 File: loopArgument.sh

i=1;
$@ represent as the array of all the parameters passed
for program in "$@"
do
 echo "$i : Programming In $program";
 i=$((i + 1));
done

```

*Program3 that contains \$@*

## Output :

```

[rajraj@139-162-5-218 ~]$ sh loopArgument.sh Java Python Ruby R C++
1 : Programming In Java
2 : Programming In Python
3 : Programming In Ruby
4 : Programming In R
5 : Programming In C++
[rajraj@139-162-5-218 ~]$

```

Another way is by using the Shift operator instead of \$@ – Shift operator:

The \$# variable is used to return the input size. By using that and with the shift operator we can achieve instead of \$@

```

i=1;
totalArguments=$#
$# returns the input size
while [$i -le $totalArguments]
do
 echo "$i: Programming is fun with $1"; #1 refers to the next element
 i=$((i + 1));
 shift 1; #Shifting the positional argument
done

```

*Program4*

## Output:

```

[rajraj@139-162-5-218 ~]$ sh shiftArgument.sh Java Python Ruby R C++
1: Programming is fun with Java
2: Programming is fun with Python
3: Programming is fun with Ruby
4: Programming is fun with R
5: Programming is fun with C++
[rajraj@139-162-5-218 ~]$

```

## Test Command

A test command is a command that is used to test the validity of a command. It checks whether the command/expression is true or false. It is used to check the type of file and the permissions related to a file. Test command returns 0 as a successful exit status if the command/expression is true, and returns 1 if the command/expression is false.

### Syntax:

```
test [expression]
```

### Example:

```
test "variable1" operator "variable2"
```

Here, expression can be any command or expression that can be evaluated by the shell. And it is recommended to always enclose out test variables into double-quotes.

Here, are some of the operator flags that can be used with **test** command, along with their meaning:

### Flags for files and directories:

- **test -e filename:** Checks whether the file exists or not. And return 1 if file exists and returns 0 if file does not exist.
- **test -d filename:** Checks whether the file is a directory or not. And returns 0 if the file is a directory and returns 1 if the file is not a directory.
- **test -f filename:** Checks whether the file is a regular file or not. And returns 0 if the file is a regular file and returns 1 if the file is not a regular file.
- **test -s filename:** Checks whether the file is empty or not. And returns 0 if the file is not empty and returns 1 if the file is empty.

- **test -r filename:** Checks whether the file is readable or not. And returns 0 if the file is readable and returns 1 if the file is not readable.
- **test -w filename:** Checks whether the file is writable or not. And returns 0 if the file is writable and returns 1 if the file is not writable.
- **test -x filename:** Checks whether the file is executable or not. And returns 0 if the file is executable and returns 1 if the file is not executable.

### Flags for text strings

- **string1 = string2:** Checks whether the two strings are equal or not. And returns 0 if the two strings are equal and returns 1 if the two strings are not equal.
- **string1 != string2:** Checks whether the two strings are not equal or not. And returns 0 if the two strings are not equal and returns 1 if the two strings are equal.
- **-n string:** Checks whether the string is empty or not. And returns 1 if the string is empty and returns 0 if the string is not empty.
- **-z string:** Checks whether the string is empty or not. And returns 0 if the string is empty and returns 1 if the string is not empty.

### Flags for comparison of numbers

- **num1 -eq num2:** Checks whether the two numbers are equal or not. And returns 0 if the two numbers are equal and returns 1 if the two numbers are not equal.

- **num1 -ne num2:** Checks whether the two numbers are not equal or not. And returns 0 if the two numbers are not equal and returns 1 if the two numbers are equal.
- **num1 -gt num2:** Checks whether the first number is greater than the second number or not. And returns 0 if the first number is greater than the second number and returns 1 if the first number is not greater than the second number.
- **num1 -ge num2:** Checks whether the first number is greater than or equal to the second number or not. And returns 0 if the first number is greater than or equal to the second number and returns 1 if the first number is not greater than or equal to the second number.
- **num1 -lt num2:** Checks whether the first number is less than the second number or not. And returns 0 if the first number is less than the second number and returns 1 if the first number is not less than the second number.
- **num1 -le num2:** Checks whether the first number is less than or equal to the second number or not. And returns 0 if the first number is less than or equal to the second number and returns 1 if the first number is not less than or equal to the second number.

### Conditional flags

- **condition1 -a condition2:** Checks whether the two conditions are true or not. And returns 0 if both the conditions are true and returns 1 if either of the conditions are false.

- **condition1 -o condition2:** Checks whether the two conditions are true or not. And returns 0 if either of the conditions are true and returns 1 if both the conditions are false.
- **!expression:** Checks whether the expression is true or not. And returns 0 if the expression is false and returns 1 if the expression is true.

So let's take a few examples to understand the test command better.

### 1. Numeric comparisons

Below is an example to check if two numbers are equal or not.

```
#!/bin/bash
Example to check if two numbers are equal
or not

first number
a=20

second number
b=20

using test command to check if numbers
are equal
if test "$a" -eq "$b"
then
 echo "a is equal to b"
else
 echo "a is not equal to b"
fi
```

**Output:**

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE fish + v

amninder@amninder ~/D/G/New Folder> bash main.sh
a is equal to b
amninder@amninder ~/D/G/New Folder>
```

*Numeric comparison*

## 2. String comparisons

Test command allows us to compare strings as well. We can check if two strings are equal or not, if one string is greater than the other if one string is less than the other, etc. Based on the string size, we can perform the comparison.

## 3. String equality

Below is a simple example to check the equality of two strings using the test command

```
#!/bin/bash
Example to check if two strings are equal or not
first string
a="Hello"
b="Hello"
if test "$a" = "$b"
then
 echo "a is equal to b"
else
 echo "a is not equal to b"
fi
```

**Output:**

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE fish + v

amninder@amninder ~/D/G/New Folder> bash main.sh
a is equal to b
amninder@amninder ~/D/G/New Folder>
```

*String equality comparison*

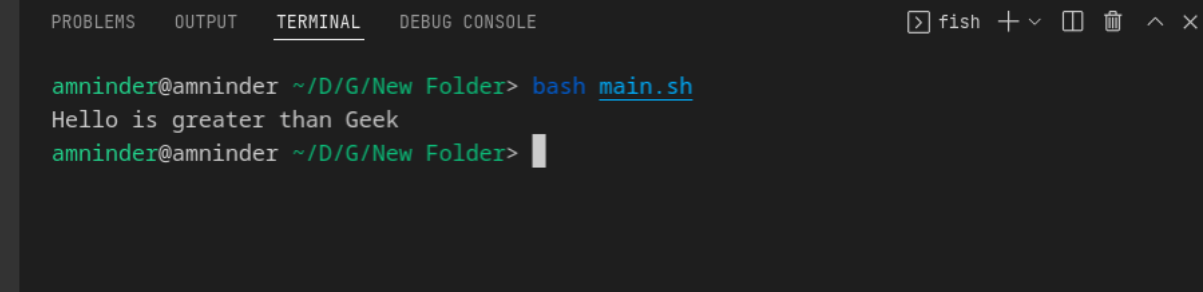
## 4. String order

String order basically means checking if one string is greater than the other or not. Below is the example to check if one string is greater than the other.

### Script:

```
#!/bin/bash
Example to check if one string is greater than
the other
first string
a="Hello"
b="Geek"
if test "$a" \> "$b"
then
 echo "$a is greater than $b"
else
 echo "$a is not greater than $b"
fi
```

### Output:

A screenshot of a terminal window with a dark background. The terminal has tabs at the top labeled 'PROBLEMS', 'OUTPUT', 'TERMINAL' (which is active), and 'DEBUG CONSOLE'. On the right side of the terminal bar, there are icons for fish shell, a plus sign, a dropdown arrow, a window icon, a trash icon, and up/down arrows. The terminal content shows a user prompt 'amninder@amninder' followed by a directory path '~/D/G/New Folder' and a command 'bash main.sh'. The output of the script is 'Hello is greater than Geek'. The prompt is followed by a cursor.

*String order comparison*

We have to use '`\>`' instead of '`>`' because '`>`' is a redirection operator. And to make it work, we have to use '`\>`' instead of '`>`'. '`\`' is an escape character.

## 5. String size

We can also compare strings based on their size using the test command. Below are some examples to perform the comparisons.

```
#!/bin/bash
Example to check if string contains some data
```

```
or not

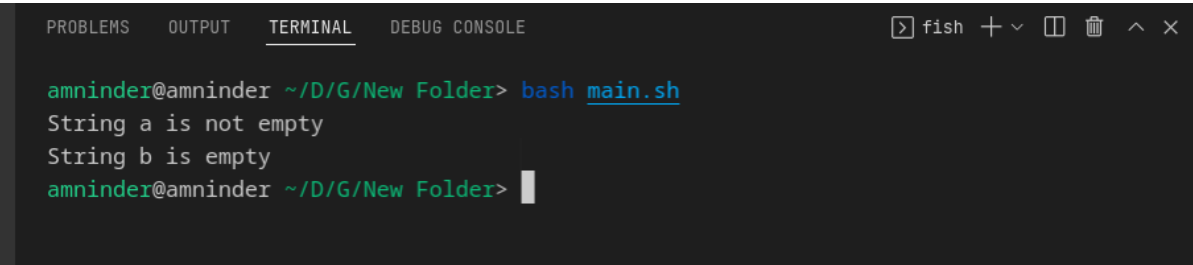
first string
a="Hello"

second string
b=""

using test command on first string to check if
string is empty or not
-n will return true if string is not empty
if test -n "$a"
then
 echo "String a is not empty"
else
 echo "String a is empty"
fi

-z will return true if string is empty
if test -z "$b"
then
 echo "String b is empty"
else
 echo "String b is not empty"
fi
```

### Output:

A screenshot of a terminal window with a dark background. The window has tabs at the top labeled 'PROBLEMS', 'OUTPUT', 'TERMINAL' (which is active), and 'DEBUG CONSOLE'. On the right side of the terminal bar are icons for fish shell, a plus sign, a window icon, a trash icon, and window control buttons (up, down, close). The terminal content shows the execution of a script: 'amninder@amninder ~/D/G/New Folder> bash main.sh', followed by the output 'String a is not empty' and 'String b is empty'. The prompt 'amninder@amninder ~/D/G/New Folder>' is shown again at the bottom with a cursor.

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE fish + [] [] ^ x

amninder@amninder ~/D/G/New Folder> bash main.sh
String a is not empty
String b is empty
amninder@amninder ~/D/G/New Folder>
```

*String size comparison*



## 6. File comparisons

Test command also aids us to test the status of files and folders on the Linux file system. Below is an example to check if a file is empty or not. For this purpose, we can use the `-s` flag which will return true if the file size is greater than 0 means if the file is not empty and it returns false if the file is empty.

```
#!/bin/bash
Example to check if file is empty or not
test command with -s flag to check if file
is empty or not
creating new file with the given name
filename="I_LOVE_GEEKSFORGEEKS.txt"

touch command is used to create empty file
touch $filename

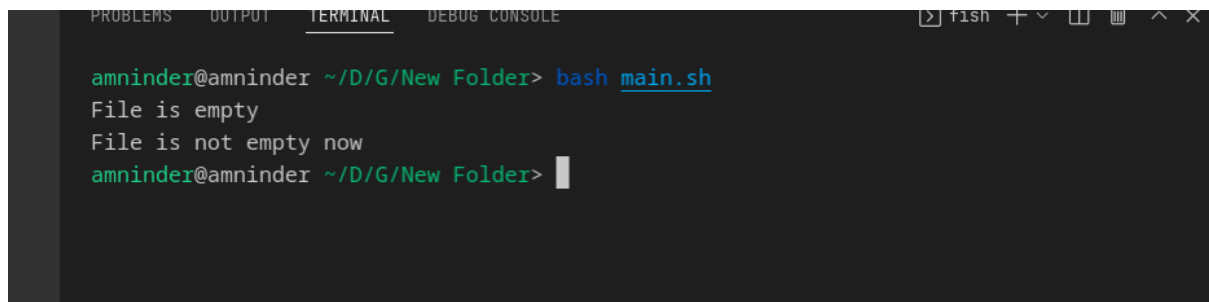
checking if file is empty or not
if test -s $filename
then
 echo "File is not empty"
else
 echo "File is empty"
fi

adding to the newly created text file
echo "I love GeeksforGeeks" >> $filename

checking again if file is empty or not
if test -s $filename
then
 echo "File is not empty now"
else
 echo "File is empty"
```

fi

### Output:

A terminal window with tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE. The terminal shows a user running a script. The output indicates the file was initially empty and then not empty after a second run.

```
amninder@amninder ~/D/G/New Folder> bash main.sh
File is empty
File is not empty now
amninder@amninder ~/D/G/New Folder>
```

*file comparisons*

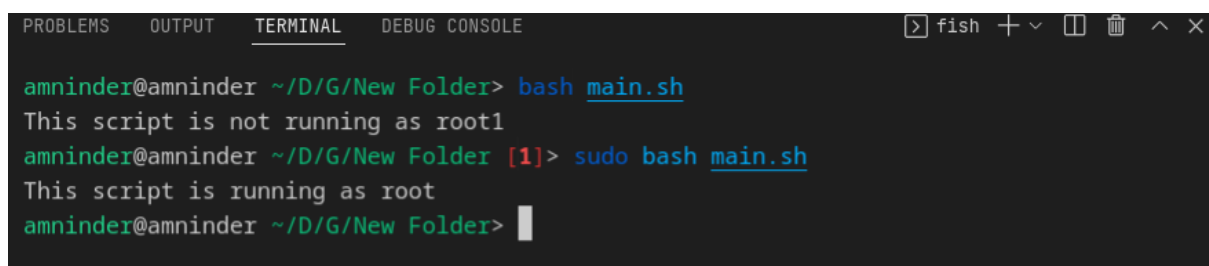
### Example :

Script to check if the script is running as root or not, using the test command.

```
check if the script is run as root or not
if test "$(id -u)" == "0"
then
 echo "This script is running as root"
else
 echo "This script is not running as root"1>&2
 exit 1
fi
```

fi

### Output:

A terminal window showing the script being run twice. The first run shows it is not running as root, and the second run, executed with sudo, shows it is running as root.

```
amninder@amninder ~/D/G/New Folder> bash main.sh
This script is not running as root1
amninder@amninder ~/D/G/New Folder [1]> sudo bash main.sh
This script is running as root
amninder@amninder ~/D/G/New Folder>
```

*Script to check if the script is running as root or not using the test command*

Here, the **-id u** flag is used to check the user id of the user who is running the script. And if the user id is not 0 that means the user is not root and the script will print the else statement. The **1>&2** is used to redirect the output to

stderr. Exit 1 will exit the script with a status code of 1 i.e., failure and if we do not use exit 1 the script will exit with a status code of 0 i.e., success.

**Note:** Here, the highlighted 1 in red color is used to indicate that the script that we run earlier was exit with status code 1. Because it was not running as root.

The **expr** command in Unix evaluates a given expression and displays its corresponding output. It is used for:

- Basic operations like addition, subtraction, multiplication, division, and modulus on integers.
- Evaluating regular expressions, string operations like substring, length of strings etc.

### Syntax:

```
$expr expression
```

### Options:

**Option --version :** It is used to show the version information.

### Syntax:

```
$expr --version
```

- **Example:**

```
anshul@anshul-VirtualBox:~/Desktop$ expr --version
expr (GNU coreutils) 8.28
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Mike Parker, James Youngman, and Paul Eggert.
anshul@anshul-VirtualBox:~/Desktop$
```

**Option –help :** It is used to show the help message and exit.

**Syntax:**

`$expr --help`

- **Example:**

```
anshul@anshul-VirtualBox:~/Desktop$ expr --help
Usage: expr EXPRESSION
or: expr OPTION

 --help display this help and exit
 --version output version information and exit

Print the value of EXPRESSION to standard output. A blank line below
separates increasing precedence groups. EXPRESSION may be:

ARG1 | ARG2 ARG1 if it is neither null nor 0, otherwise ARG2

ARG1 & ARG2 ARG1 if neither argument is null or 0, otherwise 0

ARG1 < ARG2 ARG1 is less than ARG2
ARG1 <= ARG2 ARG1 is less than or equal to ARG2
ARG1 = ARG2 ARG1 is equal to ARG2
ARG1 != ARG2 ARG1 is unequal to ARG2
ARG1 >= ARG2 ARG1 is greater than or equal to ARG2
ARG1 > ARG2 ARG1 is greater than ARG2

ARG1 + ARG2 arithmetic sum of ARG1 and ARG2
ARG1 - ARG2 arithmetic difference of ARG1 and ARG2

ARG1 * ARG2 arithmetic product of ARG1 and ARG2
ARG1 / ARG2 arithmetic quotient of ARG1 divided by ARG2
ARG1 % ARG2 arithmetic remainder of ARG1 divided by ARG2

STRING : REGEXP anchored pattern match of REGEXP in STRING

match STRING REGEXP same as STRING : REGEXP
substr STRING POS LENGTH substring of STRING, POS counted from 1
index STRING CHARS index in STRING where any CHARS is found, or 0
length STRING length of STRING
+ TOKEN interpret TOKEN as a string, even if it is a
 keyword like 'match' or an operator like '/'

(EXPRESSION) value of EXPRESSION

Beware that many operators need to be escaped or quoted for shells.
Comparisons are arithmetic if both ARGs are numbers, else lexicographical.
Pattern matches return the string matched between \(and \) or null; if
\(and \) are not used, they return the number of characters matched or 0.

Exit status is 0 if EXPRESSION is neither null nor 0, 1 if EXPRESSION is null
or 0, 2 if EXPRESSION is syntactically invalid, and 3 if an error occurred.
```

*Below are some examples to demonstrate the use of “expr” command:*

**1. Using expr for basic arithmetic operations :**

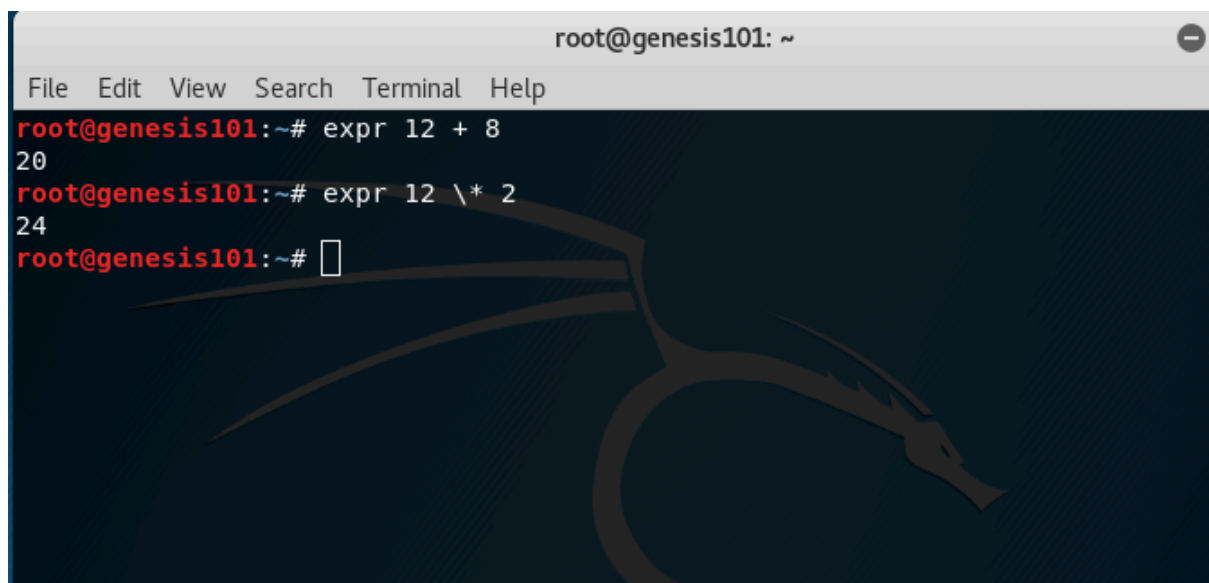
### Example: Addition

```
$expr 12 + 8
```

### Example: Multiplication

```
$expr 12 * 2
```

### Output

A screenshot of a terminal window titled 'root@genesis101: ~'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The background is dark blue with a faint dragon logo. The terminal shows the following commands and outputs:

```
root@genesis101:~# expr 12 + 8
20
root@genesis101:~# expr 12 * 2
24
root@genesis101:~#
```

**Note:**The multiplication operator `*` must be escaped when used in an arithmetic expression with `expr`.

## 2. Performing operations on variables inside a shell script

### Example: Adding two numbers in a script

```
echo "Enter two numbers"
```

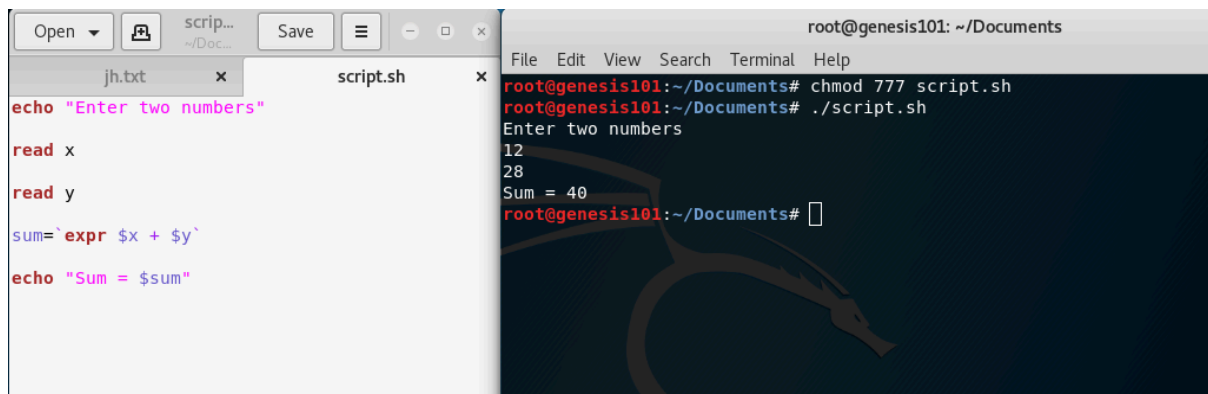
```
read x
```

```
read y

sum=`expr $x + $y`

echo "Sum = $sum"
```

## Output:



```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# chmod 777 script.sh
root@genesis101:~/Documents# ./script.sh
Enter two numbers
12
28
Sum = 40
root@genesis101:~/Documents#
```

**Note:** `expr` is an external program used by Bourne shell. It uses `expr` external program with the help of backtick. The *backtick*(```) is actually called command substitution.

## 3. Comparing two expressions

### Example:

```
x=10

y=20

matching numbers with '='
res=`expr $x = $y`

echo $res
```

```
displays 1 when arg1 is less than arg2
res=`expr $x \< $y`
echo $res
```

```
display 1 when arg1 is not equal to arg2
res=`expr $x \!= $y`
echo $res
```

## Output:

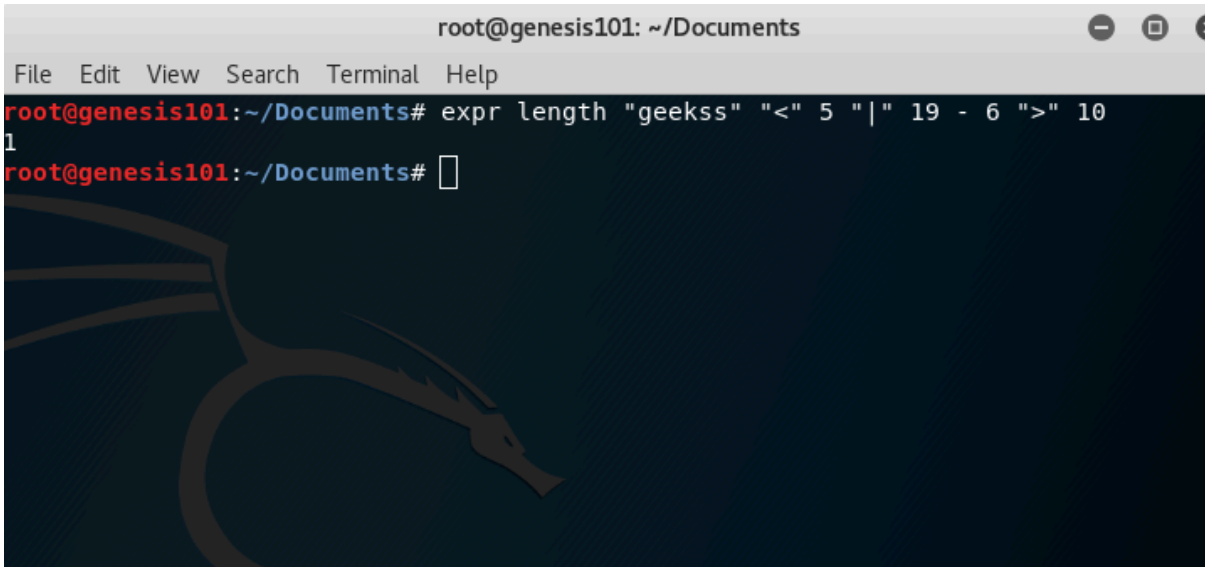
```
Open scrip... Save jh.txt script.sh
x=10
y=20
matching numbers with '='
res=`expr $x = $y`
echo $res
displays 1 when arg1 is less than arg2
res=`expr $x \< $y`
echo $res
display 1 when arg1 is not equal to arg2
res=`expr $x \!= $y`
echo $res
```

```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
0
1
1
root@genesis101:~/Documents#
```

## Example: Evaluating boolean expressions

```
OR operation
$expr length "geekss" "<" 5 "|" 19 - 6 ">" 10
```

Output:

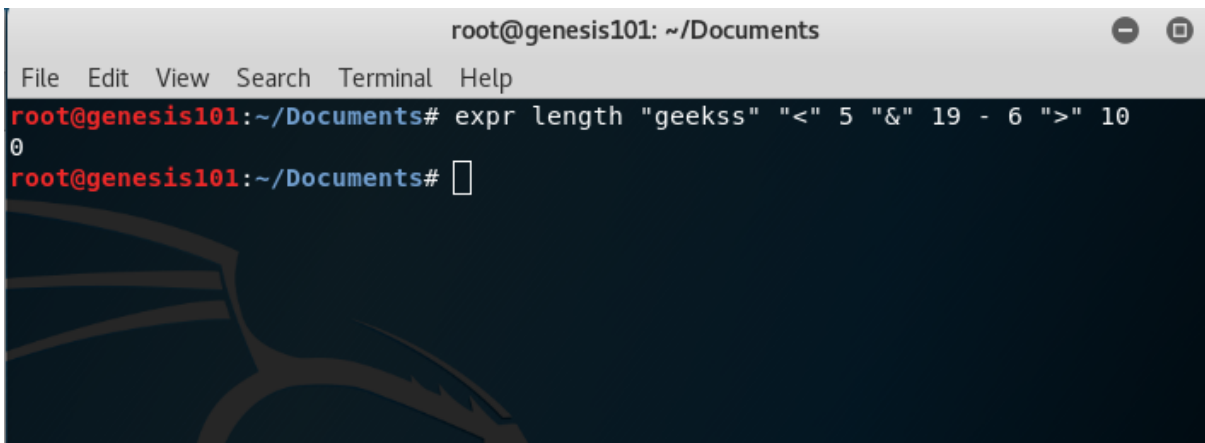
A terminal window titled 'root@genesis101: ~/Documents' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'root@genesis101:~/Documents#'. The command entered is 'expr length "geekss" "<" 5 "|" 19 - 6 ">" 10'. The output is '1'. The prompt is then 'root@genesis101:~/Documents#' with a cursor.

```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# expr length "geekss" "<" 5 "|" 19 - 6 ">" 10
1
root@genesis101:~/Documents#
```

# AND operation

```
$expr length "geekss" "<" 5 "&" 19 - 6 ">" 10
```

Output:

A terminal window titled 'root@genesis101: ~/Documents' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'root@genesis101:~/Documents#'. The command entered is 'expr length "geekss" "<" 5 "&" 19 - 6 ">" 10'. The output is '0'. The prompt is then 'root@genesis101:~/Documents#' with a cursor.

```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# expr length "geekss" "<" 5 "&" 19 - 6 ">" 10
0
root@genesis101:~/Documents#
```

#### 4. For String operations

**Example:** Finding length of a string

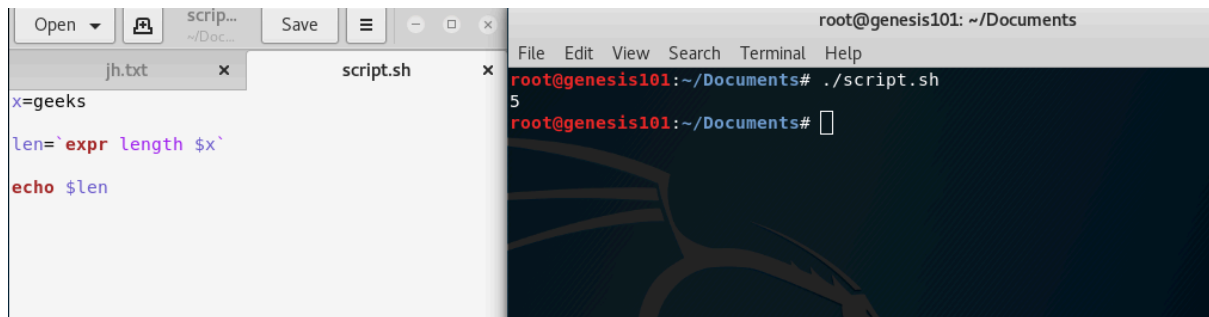
```
x=geeks
```



```
len=`expr length $x`
```

```
echo $len
```

### Output:



```
Open scrip... Save jh.txt script.sh
x=geeks
len=`expr length $x`
echo $len

root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
5
root@genesis101:~/Documents#
```

### Example: Finding substring of a string

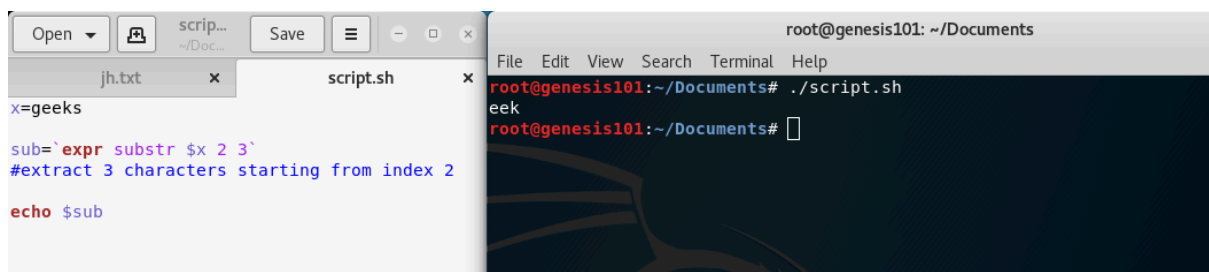
```
x=geeks
```

```
sub=`expr substr $x 2 3`
```

```
#extract 3 characters starting from index 2
```

```
echo $sub
```

### Output:



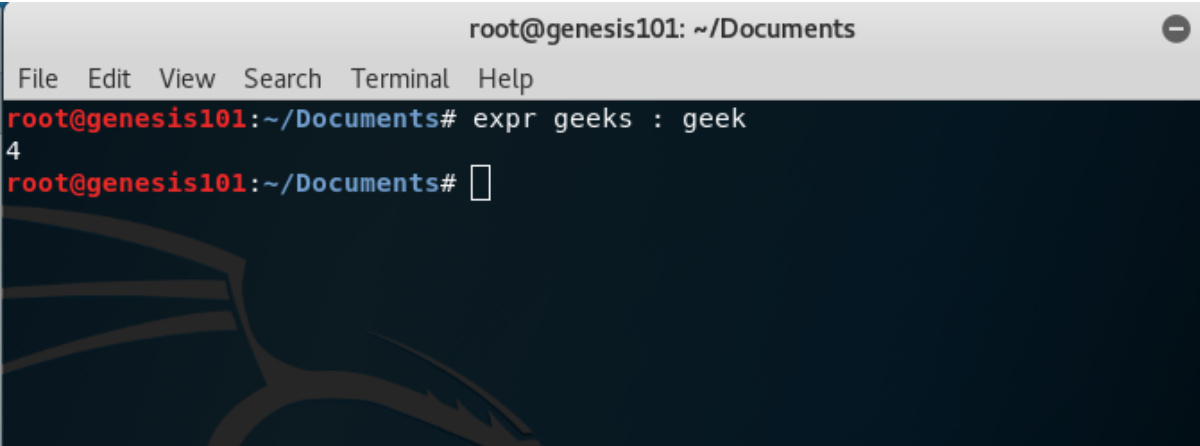
```
Open scrip... Save jh.txt script.sh
x=geeks
sub=`expr substr $x 2 3`
#extract 3 characters starting from index 2
echo $sub

root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
eek
root@genesis101:~/Documents#
```

### Example: Matching number of characters in two strings

```
$ expr geeks : geek
```

## Output:

A terminal window titled 'root@genesis101: ~/Documents' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'root@genesis101:~/Documents#'. The command 'expr geeks : geek' is entered, and the output '4' is displayed. The prompt is then shown again with a cursor.

```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# expr geeks : geek
4
root@genesis101:~/Documents#
```

### Providing command line options to scripts

Providing command-line options to shell scripts allows you to customize the behavior of your script without modifying its code. These options are often specified when invoking the script and are accessed within the script using special variables. The most common way to handle command-line options in shell scripts is by using the ``getopts`` command.

Here's how you can provide and handle command-line options in a shell script:

**\*\*Using ``getopts``:**

The ``getopts`` command allows you to define and parse command-line options in a structured manner. It supports both short options (single letters) and long options (full words).

```
#!/bin/bash
```

```
while getopts ":a:b:" opt; do
```

```
 case $opt in
```

```
 a)
```

```
 echo "Option a has been specified with value: $OPTARG"
```

```
 ;;
```

```
 b)
```

```
 echo "Option b has been specified with value: $OPTARG"
```

```
 ;;
```

```
 \?)
```

```
 echo "Invalid option: -$OPTARG"
```

```

 ;;
 :)
 echo "Option -$OPTARG requires an argument."
 ;;
 esac
done

```

In this example, the script is designed to accept two options: ``-a`` and ``-b``, each followed by an argument. To run the script and provide options:

```
./script.sh -a value1 -b value2
```

**\*\*Using ``$1``, ``$2``, ...:\*\***

You can also directly access command-line arguments using positional parameters ``$1``, ``$2``, etc. However, this approach becomes less readable and manageable as the number of options increases.

```

#!/bin/bash
if ["$1" == "-a"]; then
 echo "Option a has been specified with value: $2"
fi
if ["$3" == "-b"]; then
 echo "Option b has been specified with value: $4"
fi

```

To run the script with this approach:

```
./script.sh -a value1 -b value2
```

**\*\*Using ``shift``:\*\***

If you want to process both options and non-option arguments (e.g., filenames), you can use ``shift`` to iterate through the command-line arguments while processing options.

```

#!/bin/bash
while [$# -gt 0]; do
 case "$1" in
 -a)
 echo "Option a has been specified with value: $2"
 shift 2
 ;;
 -b)
 echo "Option b has been specified with value: $2"
 shift 2

```

```

 ;;
 *)
 echo "Non-option argument: $1"
 shift
 ;;
 esac
done

```

**Note:** It's a good practice to enclose variables in double quotes ("\$variable") to handle cases with spaces or special characters correctly.

Using `getopts` is generally recommended, as it provides a more standardized and user-friendly way to handle command-line options in shell scripts.

## exporting variables

### Introduction

The `export` command is a built-in [Bash](#) shell command that exports [environmental variables](#) as child processes without affecting the existing environment variables. Local shell variables are known only to the shell that created them, and starting a new shell session means that previously created variables are unknown to it.

Use the `export` command to export the variables from a shell, making them global and available in each new shell session.

## Linux export Syntax

The syntax for the `export` command is:

```
export [-f] [-n] [name[=value] ...]
```

or

```
export -p
```

### Linux export Options

The options allow users to remove, add, or see previously exported variables. The following table shows the available export options:

Option	Description
--------	-------------

- f Exports [name]s as functions.
- n Allows users to remove [name]s from the list of exported variables.
- p Displays a list of all exported variables and functions in the current shell.

#### 1. Using the -p option:

The -p option lists all variable names used in the current shell. Run:

```
export -p
```

#### 2. Using the -f option:

The -f option exports the variable names as functions. To export a name as a function, create a function in the command line with a unique name. After exporting it, call the function using its name in the command line.

### What Does Exporting Mean in This Context?

In Linux, the "export" command is used to make environment variables available to other programs and even sub-shells (child processes) launched from the shell session where the variable was exported. If we don't "export" a variable, it stays confined to the shell where it was created. This means other programs can't read it.

#### How to Do It

1. Setting a Variable: Before we export it, we first have to set a variable. we can do this like so:

```
``bash
my_variable="Hello, World!"
...`
```

2. Exporting the Variable: Once it's set, we can then export it:

```
``bash
export my_variable`
```

...

After this, any new program or shell session started from the original session will know about `my\_variable` and its value "Hello, World!"

### Real-world Example

Let's say we're developing a Python app and we want to keep our database password a secret. we could set an environment variable to hold that password.

1. Open the terminal and type:

```
```bash
DATABASE_PASSWORD="mysecretpassword"
```
```

2. Then we export it:

```
```bash
export DATABASE_PASSWORD
```
```

3. Now, in our Python app, we could read that password like this:

```
```python
import os
password = os.environ.get('DATABASE_PASSWORD')
```
```

This way, we keep the password out of our code, making it more secure. Exporting variables is a smart way to share configuration settings or other data between different parts of a system or among different applications. It can make system more modular and easier to manage. Plus, it can help keep sensitive information secure.

### Arrays

Arrays are a way to store multiple values in a single variable. In Linux, specifically in the Bash shell, we can define an array and manipulate its elements for various tasks like automating operations, data transformations, and more.

## What Are Arrays in Linux?

Arrays are like lists that we can use to store multiple values under a single name. Imagine we have a shopping list. Instead of writing each item on a different piece of paper, we write them all on one sheet. That's what arrays help us do in programming.

### How to Create an Array

1. Defining an Array: The simplest way to create an array in Bash is to use the `declare` command or just assign values to an array directly.

```
bash
Using declare
declare -a my_array=("apple" "banana" "cherry")

Direct assignment
my_array=("apple" "banana" "cherry")
```

2. Implicit Declaration: we can also implicitly declare an array by just assigning values.

```
bash
my_array[0]="apple"
my_array[1]="banana"
my_array[2]="cherry"
```

### Accessing Array Elements

- To access an individual item (known as an 'element'):

```
bash
echo ${my_array[0]} # Outputs "apple"
```

- To access all elements:

```
bash
echo ${my_array[@]} # Outputs "apple banana cherry"
```

## Useful Operations

1. Add Elements: To add an element to the array.

```
bash
my_array+=("date")
```

Now, `my\_array` will be `("apple" "banana" "cherry" "date")`.

2. Remove Elements: To remove an element, we can use `unset`.

```
bash
unset my_array[1]
```

Now, `my\_array` will be `("apple" "" "cherry" "date")`.

3. Find Length: To find out how many elements are in the array.

```
bash
echo ${my_array[#]} # Outputs the length
```

4. Loop Through Array: Looping through each element in the array can be done with a `for` loop.

```
bash
for fruit in "${my_array[@]}"; do
 echo $fruit
done
```

## Remote shell execution,

## 2. SSH

**SSH** stands for Secure Shell, is a cryptographic network protocol that runs at layer 7 of the OSI model for secure network services over the insecure network. It runs over the TCP port 22 with SSHv2 as its latest version.



It has many interesting features like running a command on the remote servers, port forwarding, tunneling, and more. Initially, the client starts the negotiation with the server. Then, the server further sends its public exchange key to invoke the channel and negotiate the other parameters. After successful negotiations, it displays the login prompt of the server host operating system. **Further on, the end-to-end data transfer takes place in encrypted form.**

### Using SSH for Remote Shell Execution

SSH is a secure protocol used to connect to a remote server. Once connected, we can execute commands as if we were sitting in front of that machine.

1. Basic Syntax: To SSH into a machine, the basic syntax is:

```
bash
ssh username@remote_host
```

For example, if our username is `john` and the remote host's IP address is `192.168.1.2`, we would type:

```
bash
ssh john@192.168.1.2
```

2. Executing a Single Command: To execute a single command on the remote machine and then disconnect, we can do:

```
bash
ssh username@remote_host 'command_to_execute'
```

Like, to check the free disk space:

```
bash
ssh john@192.168.1.2 'df -h'
```

### Using SCP for File Transfer

Sometimes we may need to transfer files rather than execute commands. SCP can help with that.

1. Sending Files: To send a file from wer local machine to a remote machine:

```
bash
scp local_file_path username@remote_host:remote_file_path
```

For example:

```
bash
scp my_file.txt john@192.168.1.2:/home/john/
```

2. Receiving Files: To get a file from a remote machine to wer local machine:

```
bash
scp username@remote_host:remote_file_path local_file_path
```

For example:

```
bash
scp john@192.168.1.2:/home/john/my_file.txt ./my_file.txt
```

## Safety Measures

1. Public Key Authentication: It's more secure to use public key authentication rather than passwords. This involves creating a pair of cryptographic keys that can authenticate we.
2. Firewall Rules: Make sure only trusted IP addresses can connect to wer remote machine.
3. Sudo Usage: Be cautious when executing commands with `sudo` as they can make system-level changes.
4. Double-check Commands: Always double-check wer commands to avoid unwanted changes or data loss.

## Why Remote Shell Execution is Important

1. Automation: Once we know how to execute remote commands, we can automate many tasks.
2. Management: Easier management of multiple systems or servers from a single machine.
3. Data Backup: Easy transfer of data for backup or migration purposes.

Remote shell execution is a crucial skill for anyone managing networks or systems. It simplifies a lot of tasks and opens the door to effective automation and management.

## 2.1. Using User Interactive SSH

For the sake of illustration, we use the below BASH script throughout this article. It extracts the basic details of the remote machine like hostname, IP address, date, and current user id:

```
#!/bin/bash
echo "CURRENT TIME = "`date`
echo "HOSTNAME = "`hostname`
echo "USER id = "`whoami`
echo "IP ADDRESS = "`ip a s enp0s3 | grep "inet " | cut -f6 -d" "`
```

Copy

First, the command will log in into the remote box using SSH, and then it executes the bash command in the Shell. The “s” option of bash helps to read the executable command from the standard input:

```
local-machine# ssh tools@192.168.56.103 'bash -s' < get_host_info.sh
tools@192.168.56.103's password:
CURRENT TIME = Sun Oct 3 18:29:38 IST 2021
HOSTNAME = REMOTE-SERVER
USER id = tools
IP ADDRESS = 192.168.56.103/24
```

Copy

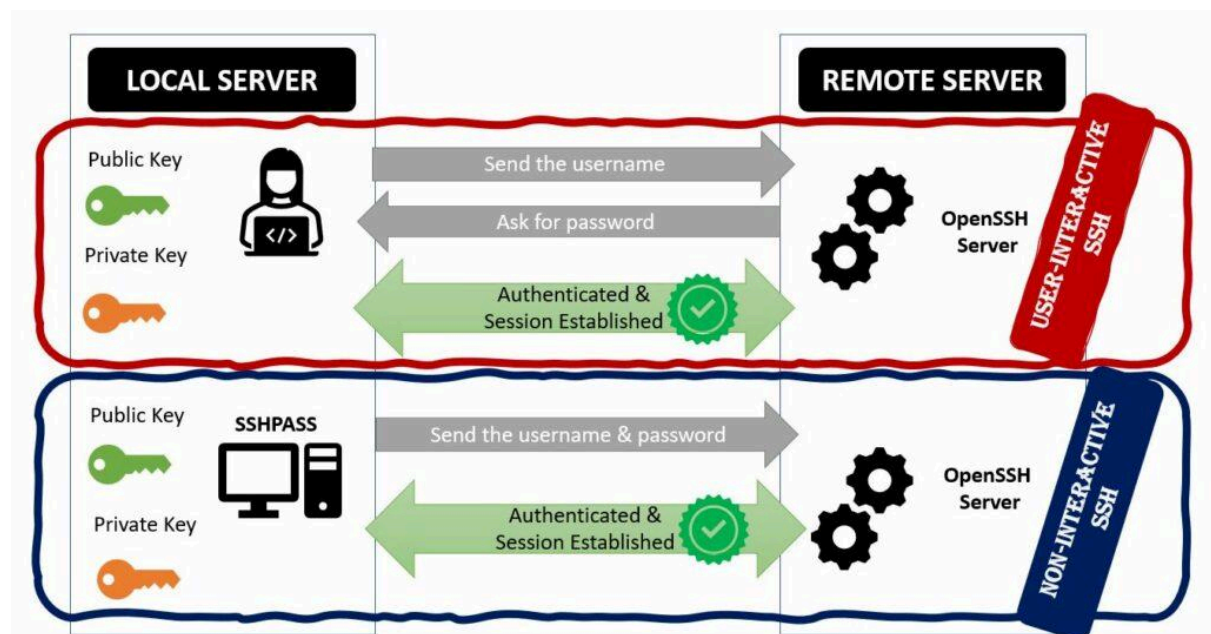
## 2.2. Using Non-interactive SSH

Needless to mention, it would be burdensome to type the password every time, in such a case, use the *sshpas* package and pass the credentials in line with the command. It is a flying hack that proffers an easy way of accessing the remote box:

```
local-machine# sshpass -vvv -p Baels@123 ssh tools@192.168.56.103 'bash
-s' < get_host_info.sh
SSHPASS searching for password prompt using match "assword"
SSHPASS read: tools@192.168.56.103's password:
SSHPASS detected prompt. Sending password.
SSHPASS read:
CURRENT TIME = Sun Oct 3 18:42:23 IST 2021
HOSTNAME = REMOTE-SERVER
USER id = tools
IP ADDRESS = 192.168.56.103/24
local-machine#
```

[Copy](#)

In the above verbose snippet, the *sshpas* is searching for the password pattern (without P/p – “assword”) where it supplies the password to the *ssh* prompt. To provide the password from the file, we can use the “-f” option of *sshpas*. **It supports *ssh* by providing the password but nevertheless, it cannot supplant the *ssh* command:**



## 3. Plink

*plink* is the open-source freely available SSH client for Windows platforms. It is the most secure, clean, and convenient method for automating SSH actions on remote machines. By default, it uses SSH protocol to manage the UNIX machines remotely. But besides, it also has other options like telnet, rlogin, raw and serial. Most importantly, *plink* is flexible enough to integrate easily with Windows batch scripts that execute the scripts remotely through SSH.

Without any further ado, let's get into action.

Here, for the sake of explanation, we have deployed a *plink* 0.62 release:

```
C:\Users\baeldung.01\Downloads>.\plink -V
plink: Release 0.62
```

Copy

The *get\_host\_info* script is placed in the local Windows machine, as shown below:

```
C:\Users\baeldung.01\Downloads>dir | findstr get_host_info
03-10-2021 03:49 166 get_host_info.sh
```

Copy

### 3.1. Using User Interactive Plink

For better elucidation, let's get the verbose output of the command using the option *-v*. Here, we force the communication protocol as SSH and give the script name under the option *-m*:

```
C:\Users\baeldung.01\Downloads>.\plink.exe -v -ssh tools@192.168.56.103
-m get_host_info.sh
Looking up host "192.168.56.103"
Connecting to 192.168.56.103 port 22
Server version: SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.5
Using SSH protocol version 2
We claim version: SSH-2.0-PuTTY_Release_0.62
...
output truncated
...
```

Copy

Firstly, it begins with the layer 3 IP reachability checks and establishes the session using port number 22. Secondly, SSH version 2 is the preferred version after successful version negotiations:

```
...
output truncated
...
Using Diffie-Hellman with standard group "group14"
Doing Diffie-Hellman key exchange with hash SHA-1
Host key fingerprint is:
ssh-rsa 2048 1e:77:e1:d2:eb:32:6b:b1:f9:cb:72:fe:ef:e6:a6:24
Initialised AES-256 SDCTR client->server encryption
Initialised HMAC-SHA1 client->server MAC algorithm
Initialised AES-256 SDCTR server->client encryption
Initialised HMAC-SHA1 server->client MAC algorithm
```

Copy

Thirdly the Diffie-Hellman key exchange algorithm is used for exchanging the cryptographic keys between two machines. Further on, communication is fully encrypted using *AES-256* and *HMAC-SHA1* algorithms.

```
...
output truncated
...
Using username "tools".
tools@192.168.56.103's password:
Sent password
Access granted
```

Copy

Lastly, we have to provide the password to authenticate the session. Now, *plink* launches the subshell in the remote machine for executing the *get\_host\_info.sh* in our local Windows machine:

```
...
output truncated
...
Opened channel for session
Started a shell/command

CURRENT TIME = Sun Oct 3 15:32:40 IST 2021
HOSTNAME = REMOTE-SERVER
USER id = tools
Server sent command exit status 0
```

```
IP ADDRESS = 192.168.56.103/24
Disconnected: All channels closed
```

Copy

Obviously, it will be tedious to type the password at all times. Hence, plink supports the inline password feature using the option `-pw`. The **option `-pw` in unison with the option `-m` bolsters its faster integration with Windows batch scripts for better management of remote servers:**

```
C:\Users\baeldung.01\Downloads>.\plink.exe tools@192.168.56.103 -m
get_host_info.sh -pw Baels@123
```

```
CURRENT TIME = Sun Oct 3 03:49:18 IST 2021
HOSTNAME = REMOTE-SERVER
USER id = tools
IP ADDRESS = 192.168.56.103/24
```

Copy

## 4. Using Expect Script

Usually, ***expect* scripting in Linux enables the automation of multiple CLI terminal-based processes**. To better assimilate, let's write an *expect* the program to run a script in the remote machine from the localhost.

A simple *expect* script is explained below that login to the remote box first, copies the script from the source to the remote machine, and subsequently executes it:

```
#!/usr/bin/expect
set timeout 60
spawn ssh [lindex $argv 1]@[lindex $argv 0]
expect "*?assword" {
 send "[lindex $argv 2]\r"
}
expect ":~$ " {
 send "
 mkdir -p /home/tools/baeldung/auto-test;
 cd /home/tools/baeldung/auto-test;
 tree
 sshpass -p 'Baels@123' scp -r
tools@10.45.67.11:/home/tools/cw/baeldung/get_host_info.sh ./;
```

```

 tree
 bash get_host_info.sh\r"
 }
expect ":\~$ " {
 send "exit\r"
}
expect eof

```

Copy

Apply the executable permissions on the created *expect* program using the *chmod* command. The credentials and IP address of the remote box are the input arguments to the program:

```

local-machine# chmod 755 get_host.exp
local-machine# ./get_host.exp "localhost" "tools" "Baels@123"
local-machine#

```

Copy

In the first step, it spawns a new *ssh* session with the remote server. Depending on the CLI pattern, it sends the password for successful login. The program creates the test environment by creating a directory and copying the main script from the source machine. Here, *sshpass* is used to send the password inline to the *scp* command for copying the main script. The program executes the copied *get\_host\_info.sh* using the BASH command in the remote box. Finally, it safely terminates the communication channel using the *exit* command:

```

local-machine# ./get_host.exp "192.168.56.103" "tools" "Baels@123"
spawn ssh tools@192.168.56.103
tools@192.168.56.103's password:
..
..
Last login: Mon Oct 4 12:19:02 2021 from 10.45.67.11
remote-machine# mkdir -p /home/tools/baeldung/auto-test;
remote-machine# cd /home/tools/baeldung/auto-test;
remote-machine# ls -ltrh
total 0
remote-machine# sshpass -p 'Baels@123' scp -r
tools@10.45.67.11:/home/tools/cw/baeldung/get_host_info.sh ./;
remote-machine# ls -ltrh
total 4.0K
-rw-rw-r-- 1 tools tools 168 Oct 4 12:20 get_host_info.sh
remote-machine# bash get_host_info.sh
CURRENT TIME = Mon Oct 4 12:20:42 IST 2021

```



```
HOSTNAME = REMOTE-SERVER
USER id = tools
IP ADDRESS = 192.168.56.103/24
remote-machine# exit
logout
Connection to localhost closed.
local-machine#
```

Copy

As a useful tip, we can also add a couple of clean-up commands like *rm*, *mv* to make the destination clean after proper exiting.

### Chmod Command

## Chmod Command in Linux/Unix with Examples

Linux `chmod` command is used to change the access permissions of files and directories. It stands for **change mode**. It can not change the permission of symbolic links. Even, it ignores the symbolic links come across recursive directory traversal.

In the **Linux** file system, each file is associated with a particular owner and have permission access for different users. The user classes may be:

- owner
- group member
- Others (Everybody else)

The **file permissions** in Linux are the following three types:

- read (r)
- write (w)
- execute (x)

## Brief History of Chmod

First, the chmod command is represented in AT&T UNIX version 1 with the chmod system call. The access-control lists were included in several file systems in inclusion to these most common modes to enhance flexibility because systems grew in types and a number of users.

The chmod version arranged in GNU coreutils was specified by Jim Meyering and David MacKenzie. This command is present as an isolated package for Microsoft Windows as an element of the UnxUtils native Win32 port collection of basic GNU Unix-like utilities. Also, the chmod command has been shipped to the IBM i OS.

Let's see how to change the file permission using the chmod command.

### **Syntax:**

The basic syntax of chmod command is as follows:

1. `chmod <options> <permissions> <file name>`

Generally implemented options are:

- **-R:** It stands for recursive, i.e., add objects to subdirectories.
- **-V:** It stands for verbose, display objects modified (unmodified objects are not displayed).

The target object is influenced if a symbolic link is mentioned. File modes related to symbolic links themselves directly are not used typically.

The primary component of the chmod permission:

***For instance,*** `rwxr-x---`

All groups of three characters specify permissions for all classes:

- **rwX:** The leftmost three characters specify permissions for the file owner (i.e., the User class).
- **r-X:** The three middle characters specify permissions for the group owning the file (i.e., the Group class).

- **---**: The three rightmost characters specify permissions for the Other class. Users who aren't the file owner and group members can't access the file.

### Options:

The chmod command supports the following command-line options:

**-c, --changes**: It is similar to the verbose option, but the difference is that it is reported if a change has been made.

**-f, --silent, --quiet**: It is used to suppress the error messages.

**-v, --verbose**: It is used to display a diagnostic for every processed file.

**--no-preserve-root**: It is used for not treating the backslash symbol ('/'), especially (the default).

**--preserve-root**: If this option is used, it will fail to operate recursively on backslash ('/').

**--reference=RFILE**: It is used to specify the RFILE's mode alternatively MODE values.

**-R, --recursive**: It is used to change files and directories recursively.

**--help**: It is used to display the help manual having a brief description of usage and support options.

**--version**: It is used to display the version information.

## File Permission Syntax

If you are a new user, you may get confused with the different types of letters used to set the file permission. So, Before proceeding further with the chmod command, let's understand the file permission syntax.

To set the permission of a file or directory, we have to specify the following things:

- **Who**: Who we are. (user)
- **What**: What change are we going to made ( Such as adding or removing the permission)?

- Which: Which of the permissions?

The permission statement is represented in indicators such as u+x, u-x. Where 'u' stands for 'user,' '+' stands for add, '-' stands for remove, 'x' stands for executable (which).

The user value can be:

u: the owner of the file

g: group member

o: others

a: all

The permission types can be r, w, and x.

## Setting and Updating the Permissions

To set the permission of a file, execute a permission statement with the chmod command. For example, we want to set the read and write permission for all users and groups of file 'Demo.txt.' We have to pass the "u=rw,go=rw Demo.txt" permission statement with chmod command. To display the file permission, execute the below command:

1. `ls -l Demo.txt`

The above command will display the file's current file permission of the 'Demo.txt' file.

To change the permission, execute the below command:

1. `chmod u=rw,go=rw Demo.txt`

Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~$ groups
javatpoint adm cdrom sudo dip plugdev lpadmin sambashare
```

From the above output, the access permission of 'Demo.txt' has changed.

## Setting Permissions for Multiple Files

We can set permission for multiple files at once by using the `chmod` command. To change the file permission of multiple files, specify the file pattern with the `chmod` command. For example, if we want to set read and write permission for all text files, specify the `*.txt` pattern with `chmod` command.

To view the permission of all text file from the current working directory, execute the below command:

1. `ls -l *.txt`

It will list all the text files with their permission mode. Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~$ ls -l *.txt
-rw-r--r-- 1 javatpoint javatpoint 21 Jun 3 01:47 Demo1.txt
-rw-rw-rw- 1 javatpoint javatpoint 0 Jul 8 23:43 Demo.txt
-rw-r--r-- 1 javatpoint javatpoint 37 Jun 23 01:01 lookups.txt
-rw-r--r-- 1 javatpoint javatpoint 37 Jul 2 20:50 pings.txt
-rw-r--r-- 1 javatpoint javatpoint 0 Jun 11 21:21 ref.txt
-rw-r--r-- 1 javatpoint javatpoint 131 Jun 8 22:46 time.txt
-rw-r--r-- 1 javatpoint javatpoint 132 Jun 8 22:32 timme.txt
```

From the above output, many files have only read permission for other users.

To set the read and write permission for other users, execute the below command:

1. `chmod o+w *.txt`

It will set the read and write permission for other users of the text files. Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~$ chmod o+w *.txt
javatpoint@javatpoint-Inspiron-3542:~$ ls -l *.txt
-rw-r--rw- 1 javatpoint javatpoint 21 Jun 3 01:47 Demo1.txt
-rw-rw-rw- 1 javatpoint javatpoint 0 Jul 8 23:43 Demo.txt
-rw-r--rw- 1 javatpoint javatpoint 37 Jun 23 01:01 lookups.txt
-rw-r--rw- 1 javatpoint javatpoint 37 Jul 2 20:50 pings.txt
-rw-r--rw- 1 javatpoint javatpoint 0 Jun 11 21:21 ref.txt
-rw-r--rw- 1 javatpoint javatpoint 131 Jun 8 22:46 time.txt
-rw-r--rw- 1 javatpoint javatpoint 132 Jun 8 22:32 timme.txt
```

## Numerical Shorthand

We can use the numeric values instead of letters to specify the permissions. A three-digit value is used to specify the permission. The leftmost digit represents the owner (u), and the middle digit represents the group members (g). The rightmost digit represents the others (o).

The following table represents the digits and their permissions:

| Digits | Permissions                          |
|--------|--------------------------------------|
| 000    | No permission                        |
| 001    | Execute permission                   |
| 010    | Write permission                     |
| 011    | Write and execute permissions        |
| 100    | Read permission                      |
| 101    | Read and execute permissions         |
| 110    | Read and write permissions           |
| 111    | Read, write, and execute permissions |

## Symbolic modes

Also, the `chmod` command accepts the finer-grained symbolic notation, which permits changing specific modes. The symbolic mode consists of three elements, which are merged to form a single text string:

1. `$ chmod [references] [operator] [modes] file...`

The `chmod` program applies an operator to define how the file modes should be arranged. The below operators are approved:

| Operator |                                                                                                    | Description |
|----------|----------------------------------------------------------------------------------------------------|-------------|
| +        | It adds the described to the described classes.                                                    |             |
| -        | It removes the described mode from the described classes.                                          |             |
| =        | It represents that the modes described are to be created the same modes for the described classes. |             |

The modes represent which permissions will to be removed or granted from the described classes. There are mainly three common modes that are related to the common permissions:

| Name            | Mode | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-----------------|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| read            | r    | It reads a file or lists the contents of a directory.                                                                                                                                                                                                                                                                                                                                                                                                                     |
| write           | w    | It writes to a directory or file.                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| execute         | x    | It recurses a directory tree or executes a file.                                                                                                                                                                                                                                                                                                                                                                                                                          |
| special execute | X    | It is not permission but instead can be used rather than x. It uses the execute permissions for directories despite their current permissions and uses the execute permissions for a file that has at least an execute permissions bit set. It is helpful if used with the "+" operator and without setting the execute permission which would happen if we just used <code>chmod -R a+rx .</code> , whereas we can implement <code>chmod -R a+rx .</code> with x rather. |

Multiple modifications can be described by isolating multiple symbolic modes along with commas. The `chmod` command will inspect the ***umask*** if a user isn't specified.

## Special modes

Also, the `chmod` command can change the special modes and extra permissions of a directory or file. The symbolic modes apply 's' to indicate the setgid and setuid modes and 't' to indicate the sticky mode. A mode is only used for the correct classes, despite whether other classes are mentioned or not.

Almost all operating systems numerically support the special mode specification, specifically in octal, but a few don't. Only the symbolic modes can be applied to these systems.

**Some examples of the command line:**

| Command                                          | Description                                                                                                                                               |
|--------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>chmod a+r publicComments.txt</code>        | It will add the read permission for every class (i.e., Group, Owner, and Others).                                                                         |
| <code>chmod a-x publicComments.txt</code>        | It will remove the execute permission for every class.                                                                                                    |
| <code>chmod a+rx viewer.sh</code>                | It will add the execute and read permissions for every class.                                                                                             |
| <code>chmod u=rw, g=r, o=internalPlan.txt</code> | It will set the write and read permissions for the user, ser read for Group, and reject access for Others.                                                |
| <code>chmod -R u+w, go-w docs</code>             | It will include the write permission into the directory docs and each of its content for the owner and deletes the write permission for others and group. |



|                                                         |                                                                                                                      |
|---------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <b>chmod</b> <b>ug=rw</b><br><b>groupAgreements.txt</b> | It will set the write and read permissions for Group and user.                                                       |
| <b>chmod</b> <b>664</b><br><b>global.txt</b>            | It will set the write and read permissions for Group and user and gives the read permission to Others.               |
| <b>chmod</b> <b>744</b><br><b>Show_myCV.sh</b>          | It will set the execute, write, and read permissions for the user and gives the read permission to Group and Others. |

### Concurrent TCP sockets

In networking, concurrent TCP sockets refer to the ability of a system to handle multiple TCP (Transmission Control Protocol) socket connections simultaneously. TCP is a connection-oriented protocol that provides reliable, stream-oriented communication between two devices on a network. Concurrent TCP sockets are essential for efficiently handling multiple clients or connections concurrently in server applications.

Here's a general overview of how concurrent TCP sockets work:

#### 1. **\*\*Server Setup:\*\***

- A server application creates a TCP socket and binds it to a specific IP address and port.
- The server then listens for incoming connection requests from clients.

#### 2. **\*\*Connection Establishment:\*\***

- When a client wants to communicate with the server, it initiates a connection by sending a TCP connection request (SYN) to the server's IP address and port.

#### 3. **\*\*Accepting Connections:\*\***

- The server's listening socket accepts incoming connection requests.

- For each incoming connection, the server creates a new socket to handle communication with that specific client.

#### 4. **Concurrent Handling:**

- To handle multiple clients simultaneously, the server typically employs concurrency mechanisms such as multithreading or multiprocessing.
- Each accepted connection is assigned to a separate thread or process, allowing the server to handle multiple connections concurrently.

#### 5. **Communication:**

- The server communicates with each client through its dedicated socket.
- Data is exchanged using the TCP protocol, ensuring reliable and ordered delivery of messages.

#### 6. **Closing Connections:**

- When a client or the server decides to end the communication, a TCP connection termination process takes place (FIN, FIN-ACK, ACK).

Key considerations for concurrent TCP sockets:

#### - **Concurrency Models:**

- **Multithreading:** Each connection is handled by a separate thread.
- **Multiprocessing:** Each connection is handled by a separate process.
- **Event-Driven:** Uses an event loop to manage multiple connections asynchronously.

#### - **Synchronization:**

- Proper synchronization mechanisms are required to manage shared resources (e.g., data structures, variables) among concurrent threads or processes.

#### - **Scalability:**

- Efficient handling of a large number of concurrent connections is crucial for scalable server applications.

- **Error Handling:**

- Robust error handling is essential to handle unexpected situations and maintain the stability of the server.

- **Resource Management:**

- Proper resource management is crucial to avoid resource leaks and ensure optimal system performance.

Implementing concurrent TCP sockets allows servers to efficiently handle multiple clients simultaneously, making it suitable for applications with a high level of concurrency, such as web servers, chat applications, and online multiplayer games.

## Connecting to MySQL from the Command Line

To connect to MySQL from the command line, follow these steps:

1. Log in to your A2 Hosting account using SSH.
2. At the command line, type the following command, replacing *username* with your username:
3. **Copy**
4. `mysql -u username -p`
5. At the **Enter Password** prompt, type your password. When you type the correct password, the **mysql>** prompt appears.
6. To display a list of databases, type the following command at the **mysql>** prompt:
7. **Copy**
8. `show databases;`

9. *Make sure you do not forget the semicolon at the end of the statement.*
10. To access a specific database, type the following command at the **mysql>** prompt, replacing *dbname* with the name of the database that you want to access:
  11. **Copy**
  12. `use dbname;`
  13. *Make sure you do not forget the semicolon at the end of the statement.*
14. After you access a database, you can run SQL queries, list tables, and so on. Additionally:
  - To view a list of MySQL commands, type `help` at the **mysql>** prompt.
  - To exit the *mysql* program, type `\q` at the **mysql>** prompt.
15. *When you run a command at the **mysql>** prompt, you may receive a warning message if MySQL encounters a problem. For example, you may run a query and receive a message that resembles the following:*

```
Query OK, 0 rows affected, 1 warning (0.04 sec).
```

*To view the complete warning message, type the following command:*
  16. **Copy**
  17. `SHOW WARNINGS;`

## Essential system administration

Essential system administration tasks in Linux are crucial for maintaining and managing a Linux system effectively. Here are some key tasks and concepts for Linux system administration:

### 1. **\*\*User and Group Management\*\***:

- **\*\*Creating Users\*\***: Use the `'useradd'` or `'adduser'` command to create new user accounts.

- **Modifying Users**: Use `usermod` to modify user account properties like username, home directory, or group.
- **Deleting Users**: Use `userdel` to remove user accounts.
- **Group Management**: Create and manage groups with `groupadd`, `groupmod`, and `groupdel`.

## 2. **File Permissions**:

- Understand and set file permissions using `chmod` (change mode).
- Manage file ownership with `chown` (change owner) and `chgrp` (change group).

## 3. **Package Management**:

- Use package managers like `apt` (Debian/Ubuntu), `yum` (Red Hat/CentOS), or `dnf` (Fedora) to install, update, and remove software packages.

## 4. **System Updates**:

- Regularly update the system and software packages to patch security vulnerabilities and improve performance.

## 5. **Filesystem Management**:

- Monitor disk usage with `df` (disk free) and `du` (disk usage).
- Create, format, and manage filesystems with tools like `mkfs`, `fdisk`, and `fsck`.

6. **Backup and Restore**:

- Implement backup strategies using tools like ``rsync``, ``tar``, or dedicated backup solutions.
- Practice disaster recovery to restore the system from backups.

7. **Network Configuration**:

- Configure network settings, including IP addresses, using tools like ``ifconfig``, ``ip``, and network configuration files.
- Set up and manage firewall rules with ``iptables`` or `firewalld`.

8. **Services and Daemons**:

- Start, stop, and manage system services and daemons using tools like ``systemctl``, ``service``, or ``init.d`` scripts.
- Enable services to start at boot.

9. **User and System Logs**:

- View system logs using tools like ``journalctl`` or review log files in ``/var/log``.
- Troubleshoot issues by analyzing log entries.

10. **User Authentication**:

- Configure authentication methods, including passwords, SSH keys, and PAM (Pluggable Authentication Modules).

#### 11. **\*\*System Performance Monitoring\*\***:

- Monitor system resource usage with tools like `'top'`, `'htop'`, `'vmstat'`, and `'sar'`.
- Diagnose and optimize system performance.

#### 12. **\*\*Security and Hardening\*\***:

- Implement security best practices, such as disabling unnecessary services, applying security patches, and using strong passwords.
- Configure SELinux or AppArmor for mandatory access control.

#### 13. **\*\*Shell Scripting\*\***:

- Write and maintain shell scripts to automate repetitive tasks and system maintenance.

#### 14. **\*\*Remote Access\*\***:

- Use SSH for secure remote access and administration.
- Set up remote desktop environments (e.g., VNC) if needed.

#### 15. **\*\*User and System Resource Quotas\*\***:

- Implement user and system resource quotas to control and limit resource usage.

#### 16. **\*\*Kernel Tuning\*\***:

- Adjust kernel parameters to optimize system performance for specific workloads.

#### 17. **\*\*System Monitoring Tools\*\***:

- Utilize monitoring tools like Nagios, Zabbix, or Prometheus to maintain system health and receive alerts.

#### 18. **\*\*Documentation and Logging\*\***:

- Keep detailed documentation of system configurations, changes, and procedures.

- Regularly review and archive logs for future reference.

#### 19. **\*\*User Education and Training\*\***:

- Train users and provide guidelines for best practices in using the system securely and efficiently.

#### 20. **\*\*Disaster Recovery Planning\*\***:

- Develop and regularly test disaster recovery plans to ensure the system can be restored in case of data loss or system failures.

Linux system administration is a broad and evolving field, so it's essential to stay up-to-date with the latest tools and best practices to effectively manage Linux-based systems.