UNIT - I

1. Describe history on Linux.

Linux began in 1991 as a personal project by Finnish student Linus Torvalds: to create a new free operating system kernel. The resulting Linux kernel has been marked by constant growth throughout its history. Since the initial release of its source code in 1991, it has grown from a small number of C files under a license prohibiting commercial distribution to the 4.15 version in 2018 with more than 23.3 million lines of source code,

n 1991, while studying computer science at University of Helsinki, Linus Torvalds began a project that later became the Linux kernel. He wrote the program specifically for the hardware he was using and independent of an operating system because he wanted to use the functions of his new PC with an 80386 processor. Development was done on MINIX using the GNU C Compiler. The GNU C Compiler is still the main choice for compiling Linux today, but can be built with other compilers, such as the Intel C Compiler.

The designation "Linux" was initially used by Torvalds only for the Linux kernel. The kernel was, however, frequently used together with other software, especially that of the GNU project. This quickly became the most popular adoption of GNU software. In June 1994 in GNU's bulletin, Linux was referred to as a "free UNIX clone", and the Debian project began calling its product *Debian GNU/Linux*. In May 1996, Richard Stallman published the editor Emacs 19.31, in which the type of system was renamed from Linux to Lignux. This spelling was intended to refer specifically to the combination of GNU and Linux, but this was soon abandoned in favor of "GNU/Linux".

2. Describe the FILE I/O commands (open, create,close,lseek,read,write).

System Calls

Remember that everything is a file in Unix. Because there are so many types of files, several of the system calls below will behave slightly differently for files that are not regular. I have not attempted to include the exact behavior, or all the errors that can arise for files which are not regular.

Open

Open	
Purpose	open or create a file for reading or writing
Include	#include<fcntl.h> If the optional third argument is used also include: #include<sys/types.h> #include<sys/stat.h>
Useage	int open(const char *path, int flags[, mode_t mode]); (The third argument is optional.)
Arguments	path: the (relative) path to the file flags: the file status flags mode: file permissions, used when creating a new file

The value for *mode* **must** be included when O_CREAT is set. It is simply the permissions, and can be written using C's *octal* representation (this is base eight and starts with a leading zero.) For example, to request that you (the creator) have read and write privileges and everyone else have read privileges only, you would specify

```
open("pathtofile",O_WRONLY | O_CREAT, 0644);
```

This is only a request on your part, and this request will be compared with the umask to determine the final permissions of the file. See especially [Molay, pp. 84-8], or [Stevens, p.78-81]for further details.

Notice how open allows a process to have different file descriptors to the same file. These may have different file status flags, and may even have different offsets within the file.

close

close is used to detach the use of the file descriptor for a process. When a process terminates any open file descriptors are automatically closed by the kernel.

Close	
Purpose	delete a file descriptor
Include	#include<unistd.h>
Useage	int close(int d);
Arguments	d: a file descriptor
Returns	-1 on error 0 on success (the file descriptor deleted)
Errors	EBADF: d is not an active descriptor. EINTR: An interrupt was received.

read

read starts at the file's current offset, which is then offset by the number of bytes read (for regular files.)

Read	
Purpose	read input from file
Include	#include<unistd.h>
Useage	ssize_t read(int d, void *buf, size_t nbytes);
Arguments	d: a file descriptor buf: buffer for storing bytes read nbytes: maximum number of bytes to read
Returns	-1 on error number of bytes read and placed in buf or 0 if end of file
Errors	EBADF: d is not an active descriptor.. EFAULT: buf points outside the allocated address space. EAGAIN: The file was marked for non-blocking I/O, and no data were ready to

	be read. EINVAL: The pointer associated with d was negative. EIO: An I/O error occurred while reading from the file system.

The main reason the number of bytes read may be less than the number of bytes requested in nbytes is that the end of the file was reached before the requested number of bytes has been read. See [Stevens, p. 54] for several other reasons involving other types of files.

write

write starts at the file's current offset, which is then offset by the number of bytes written to the file (for regular files.)

Write	
Purpose	write output to file
Include	#include<unistd.h>
Useage	ssize_t write(int d, void *buf, size_t nbytes);
Arguments	d: a file descriptor buf: buffer for storing bytes to be written nbytes: maximum number of bytes to read
Returns	-1 on error number of bytes written
Errors	Too numerous to list all: see **man 2 write** EBADF: d is not an active descriptor. EFAULT: Data to be written to the file points outside the allocated address space. EINVAL: The pointer associated with d was negative. EFBIG: An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. ENOSPC: There is no free space remaining on the file system containing the file.. EAGAIN: The file was marked for non-blocking I/O, and no data were ready to be read. EINTR: A signal interrupted the write before it could be completed. EIO: An I/O error occurred while reading from the file system.

lseek

Every file descriptor has an associated *current file offset*, a number of bytes from the beginning of the file. Read and write operations normally start at the current offset and cause the offset to be incremented the number of bytes read or written. lseek explicitly repositions this offset value.

Lseek	
Purpose	reposition read/write file offset
Include	#include<unistd.h>
Useage	off_t lseek(int d, off_t offset, int base);

Arguments	d: a file descriptor offset: the number of bytes to be offset base: the position from which the bytes will be offset: SEEK_SET: offset bytes from beginning of the file. SEEK_CUR: offset bytes from current value of offset. SEEK_END: offset bytes from end of the file.
Returns	-1 on error The resulting offset location as measured in bytes from the beginning of the file.
Errors	EBADF: d is not an active descriptor.. EINVAL: basenot a proper value. ESPIPE: base associated with a non-regular file (pipe, socket or FIFO.)

A file's offset can be greater than its current size. In this case, if the file is then written to it creates holes in the file, whose value is '\0'. A regular file may not be offset before the beginning of the file

lseek can be used to determine the current offset

```
off_t currpos;
ccurrpos = lseek(fd, 0, SEEK_CUR);
```

lseek can also be used to test a file descripter if it is a pipe, FIFO, or socket, since these are not capable of seeking: they force a return of -1 and set errno to ESPIPE.

create

create opens a file for writing, creating a new file if one did not exist, or truncating the current file, discarding its contents, if a file does exist. Actually, creat is now implemented by open. Its prototype, together with the necessary header file is:

```
#include<fcntl.h>
int creat(const char *path, mode_t mode);
```

but is implemented as

```
open(path, O_CREAT | O_TRUNC | O_WRONLY, mode);
```

dup and dup2

dup and dup2 duplicate the contents of an existing file descriptor. Remember, a file descriptor is the index of an array which contains a pointer to the file table. These functions allow a second file descriptor to index a pointer to the same file table. The difference is that dup takes a single argument, the file descriptor you want to duplicate, and returns a new file descriptor which is guaranteed to be the lowest available. dup2 gives you more control over the new file descriptor: it takes two arguments, an already opened file descriptor and a new file descriptor, and directs the new file descriptor to point to the same file table. This is especially valuable when we want to create pipes between programs. If the new file descriptor is actually being used, dup2 closes the file descriptor first, then reassigns it; if the two file descriptors are the same, nothing occurs.

dup	

dup2	
Purpose	duplicate an existing file descriptor
Include	#include<unistd.h>
Useage	int dup(int oldd); int dup2(int oldd, int newd);
Arguments	oldd: an existing file descriptor newd: the value of the new descriptor newd
Returns	-1 on error the value of newd
Errors	EBADF: oldd or newd is not a valid active descriptor EMFILE: Too many descriptors are active.

3. Briefly describe the Process control commands (fork,vfork,exit,wait,waitpid,exec).

It is found that in any Linux/Unix based Operating Systems it is good to understand **fork** and **vfork** system calls, how they behave, how we can use them and differences between them. Along with these **wait** and **exec** system calls are used for process spawning and various other related tasks.

Most of these concepts are explained using programming examples. In this article, I will be covering what are fork, vfork, exec and wait system calls, their distinguishing characters and how they can be better used.

fork()

fork(): System call to create a child process.

```
shashi@linuxtechi ~}$ man fork
```

This will yield output mentioning what is fork used for, syntax and along with all the required details.

The syntax used for the fork system call is as below,

Fork system call creates a child that differs from its parent process only in **pid(process ID)** and **ppid(parent process ID)**. Resource utilization is set to zero. File locks and pending signals are not inherited. (In Linux "fork" is implemented as "**copy-on-write()**").

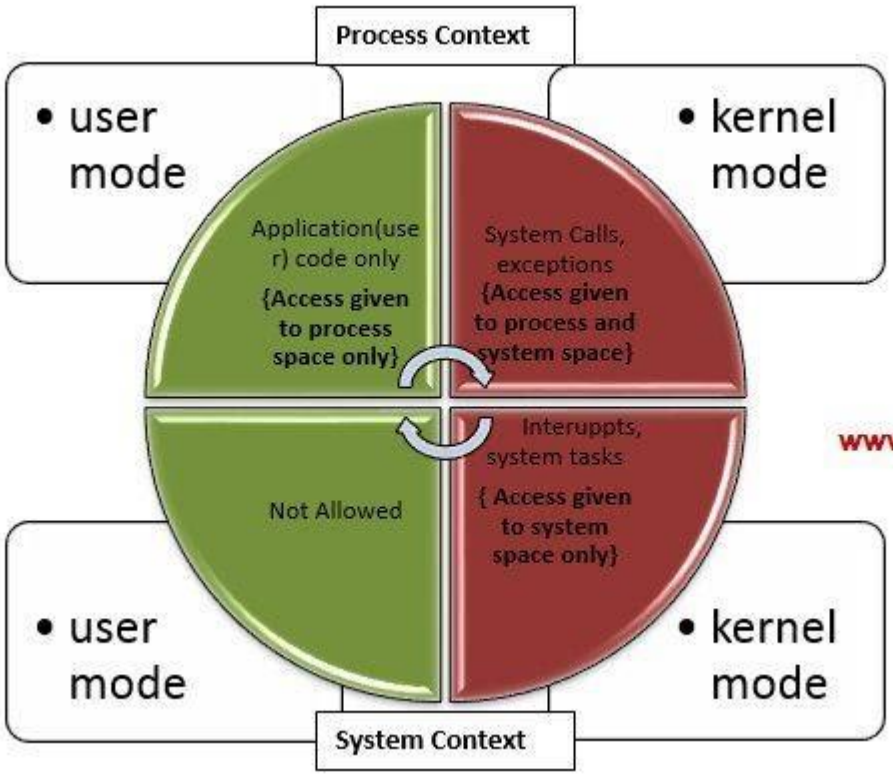
Note:- "**Copy on write**" -> Whenever a fork() system call is called, a copy of all the pages(memory) related to the parent process is created and loaded into a separate memory location by the Operating System for the child process. But this is not needed in all cases and may be required only when some process writes to this address space or memory area, then only separate copy is created/provided.

Return values:- PID (process ID) of the child process is returned in parents thread of execution and "**zero**" is returned in child's thread of execution. Following is the c-programming example which explains how fork system call works.

```
shashi@linuxtechi ~}$ vim 1_fork.c
#include<stdio.h>
#include<unistd.h>
Int main(void)
```

```
{
printf("Before fork\n");
fork();
printf("after fork\n");
}
shashi@linuxtechi ~}$
shashi@linuxtechi ~}$ cc 1_fork.c
shashi@linuxtechi ~}$ ./a.out
Before fork
After fork
shashi@linuxtechi ~}$
```

Whenever any system call is made there are plenty of things that take place behind the scene in any unix/linux machines.
First of all context switch happens from user mode to kernel(system) mode. This is based on the process priority and unix/linux operating system that we are using. In the above C example code we are using "{" opening curly brace which is the entry of the context and "}" closing curly brace is for exiting the context. The following table explains context switching very clearly.



vfork()
vfork –> create a child process and block parent process.
Note:- In vfork, signal handlers are inherited but not shared.

```
shashi@linuxtechi ~}$ man vfork
```

This will yield output mentioning what is vfork used for, syntax and along with all the required details.

```
pid_t vfork(void);
```

vfork is as same as fork except that behavior is undefined if process created by vfork either modifies any data other than a variable of type pid_t used to store the return value p of vfork or calls any other function between calling _exit() or one of the exec() family.
Note: vfork is sometimes referred to as special case of clone.
Following is the C programming example for vfork() how it works.

```
shashi@linuxtechi ~}$ vim 1.vfork.c
#include<stdio.h>
#include<unistd.h>
Int main(void)
{
printf("Before fork\n");
vfork();
printf("after fork\n");
}
shashi@linuxtechi ~}$ vim 1.vfork.c
shashi@linuxtechi ~}$ cc 1.vfork.c
shashi@linuxtechi ~}$ ./a.out
Before vfork
after vfork
after vfork
a.out: cxa_atexit.c:100: __new_exitfn: Assertion `l != NULL' failed.
Aborted
```

Note:– As explained earlier, many a times the behaviour of the vfork system call is not predictable. As in the above case it had printed before once and after twice but aborted the call with _exit() function. It is better to use fork system call unless otherwise and avoid using vfork as much as possible.
Differences between fork() and vfork()

	fork()	vfork()
Address space	Both the child and parent process will have different address space	Both child and parent process share the same address space
Modification in address space	Any modification done by the child in its address space is not visible to parent process as both will have separate copies	Any modification by child proce visible to both parent and child both will have same copies
CoW(copy on write)	This uses copy-on-write.	Vfork doesn't use CoW
Execution summary	Both parent and child executes simultaneously	Parent process will be suspend until child execution is complet
Outcome of usage	Behaviour is predictable	Behaviour is not predictable

Vfork() behaviour explained in more details in the below program.

```
shashi@linuxtechi ~}$ cat vfork_advanced.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
   int n =10;
   pid_t pid = vfork(); //creating the child process
   if (pid == 0)          //if this is a chile process
   {
      printf("Child process started\n");
   }
   else//parent process execution
   {
      printf("Now i am coming back to parent process\n");
   }
   printf("value of n: %d \n",n); //sample printing to check "n" value
   return 0;
}
shashi@linuxtechi ~}$ cc vfork_advanced.c
shashi@linuxtechi ~}$ ./a.out
Child process started
value of n: 10
Now i am coming back to parent process
value of n: 594325573
a.out: cxa_atexit.c:100: __new_exitfn:
```

Note: Again if you observe the outcome of vfork is not defined. Value of "n" has been printed first time as 10, which is expected. But the next time in the parent process it has printed some garbage value.

wait()

wait() system call suspends execution of current process until a child has exited or until a signal has delivered whose action is to terminate the current process or call signal handler.

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

There are other system calls related to wait as below,
1) **waitpid()**: suspends execution of current process until a child as specified by pid arguments has exited or until a signal is delivered.

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

2) **wait3()**: Suspends execution of current process until a child has exited or until signal is delivered.

```
pid_t wait3(int *status, int options, struct rusage *rusage);
```

3) **wait4()**: As same as wait3() but includes pid_t pid value.

```
pid_t wait3(pid_t pid, int *status, int options, struct rusage *rusage);
```

exec()
exec() family of functions or sys calls replaces current process image with new process image.
There are functions like **execl**, **execlp**,**execle**,**execv**, **execvp** and **execvpe** are used to execute a file.
These functions are combinations of array of pointers to null terminated strings that represent the argument list , this will have path variable with some environment variable combinations.
exit()
This function is used for normal process termination. The status of the process is captured for future reference. There are other similar functions **exit(3)** and **_exit().**, which are used based on the exiting process that one is interested to use or capture.
Conclusion:-
The combinations of all these system calls/functions are used for process creation, execution and modification. Also these are called "shell" spawning set-of functions. One has to use these functions cautiously keeping in mind the outcome and behavior.

4.Compare and contrast FILE I/O commands& Process control commands.

Above 2&3 answers.