LECTURE #1

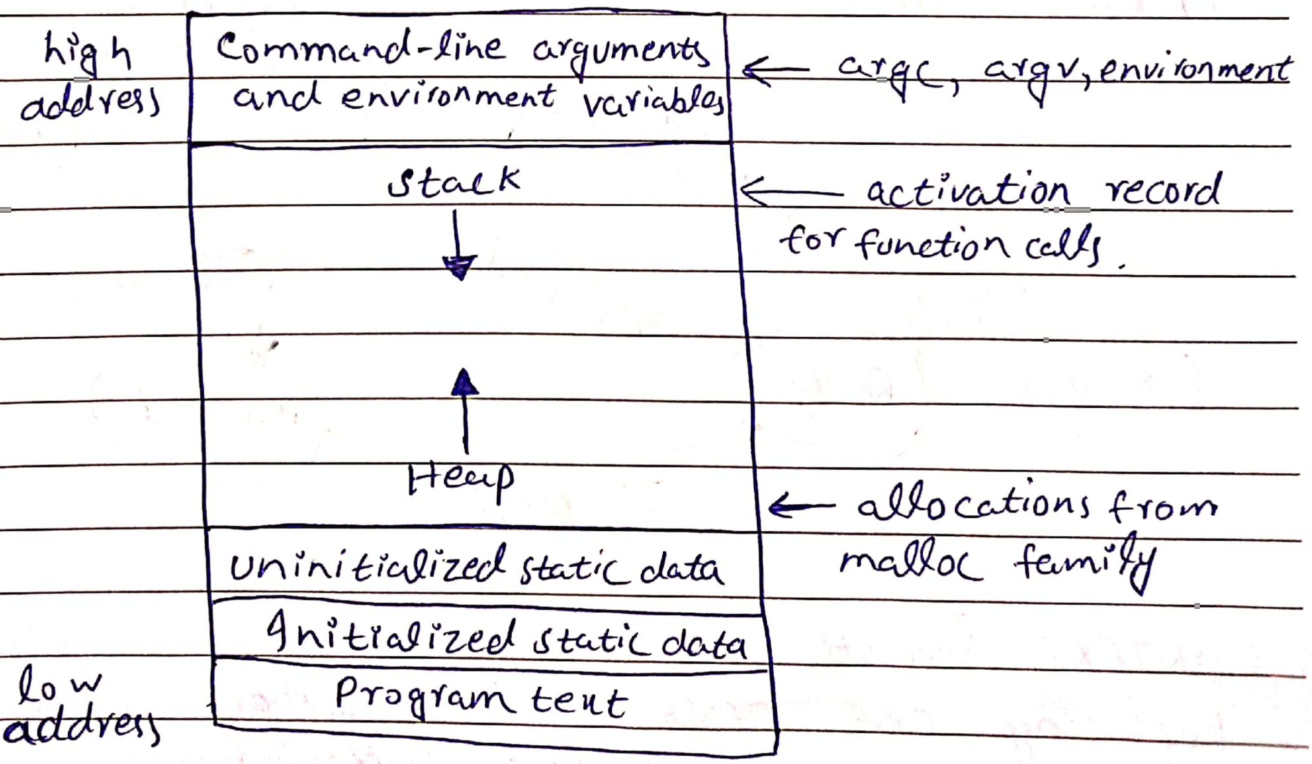
PARTS OF OPERATING SYSTEM:

1) Utilities
2) User Interface
3) Core kernel

How Program becomes a Process:

A program is a set of instructions to perform a certain task. A program in execution is called a process. when we compile a program, an object/executable file is created for that program and in order to run that program we double click on the executable file which basically calls the fork() system call,

LAYOUT OF PROGRAM IMAGE:


high address
Command-line arguments and environment variables
argc, argv, environment
stack
activation record for function calls.
Heap
allocations from malloc family
uninitialized static data
Initialized static data
Program text
low address

Note: Each thread gets its own stack.

LECTURE #2

PROCESSES IN UNIX

PCB of a process is under kernel, so we can not directly access the information stored in it.

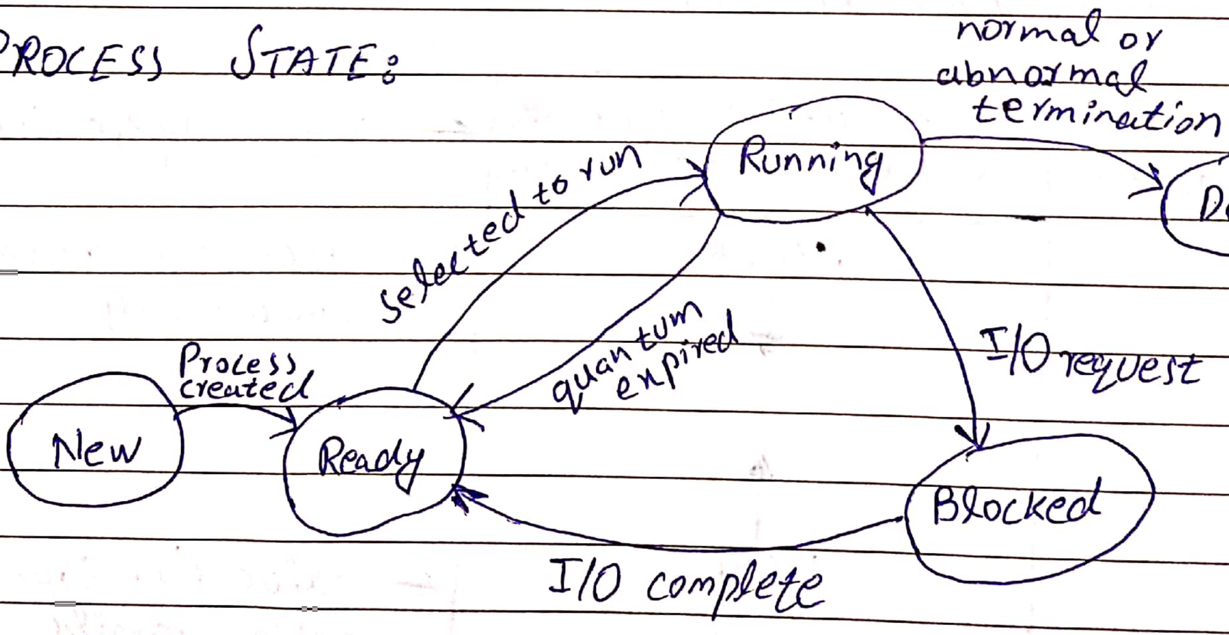
SYSTEM CALLS TO GET THE PID AND PPID:

```
pid_t getpid(void);
pid_t getppid(void);
```

SOME MORE INTERESTING SYSTEM CALLS:

1. gid_t getegid(void); (Effective Group id)
2. uid_t geteuid(void); (Effective User id)
3. git_t getgid(void); (Real group id)
4. Uid_t getuid(void); (Real user id)

PROCESS STATE:



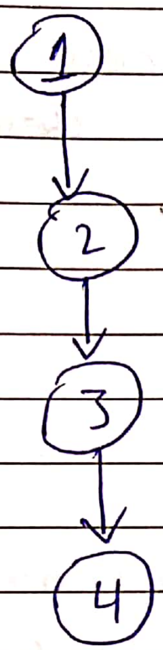
CONTEXT SWITCH:

Replacing one process with another in the CPU is known as context switching.

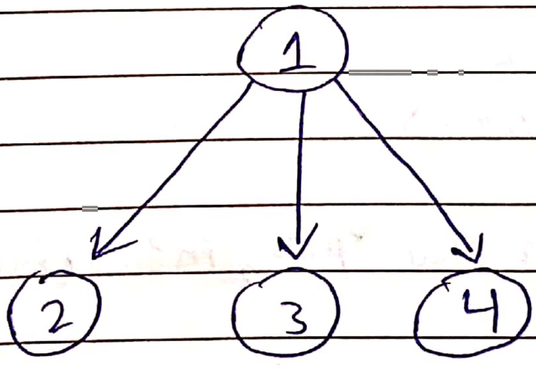
SYSTEM CALL FOR PROCESS CREATION:

pid_t fork(void);

PROCESS CHAIN:



PROCESS FAN:



BACKGROUND PROCESS:

A process detacked from its parent process.

DAEMON PROCESS:

A process which terminateg but does not know which process to merge with.

No: Date: LECTURE #3

CHAPTER 4: UNIX I/O

A process can communicate with a device through the operating system.

A process can get access to most of the devices by using these five system calls:

1. open
2. close
3. read
4. write
5. ioctl

All the devices are represented by files called special files, that are located in the /dev directory.

OPEN SYSTEM CALL:

```
#include <fcntl.h>
#include <sys/stat.h>

int open(const char *path, int oflag, ...);
```

path: Path of the file/device you want ~~to~~ open.
Oflag: Read mode, write mode etc.
The third argument is also for flags (permissions)

RETURN VALUES:

Failure: -1
Success: non-negative

CLOSE SYSTEM CALL:

```
#include <unistd.h>
 int close(int fildes);
```

fildes: File descripter number returned by the open system call.

RETURN VALUES:

Failure: -1
success: 0

READ SYSTEM CALL:

```
#include <unistd.h>
 ssize_t read(int fildes, void *buf, size_t nbyte);
```

fildes: File descripter number
*buf: Pointer to a buffer(Array).
nbyte: Number of bytes you want to read.

RETURN VALUES:

Failure: -1
Success: number of bytes read.

WRITE SYSTEM CALL:

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

fildes: File descriptor number
buf: Pointer to a buffer
nbyte: Number of bytes you want to write.

No: Date:

LECTURE #4

CHAPTER #4

If a process wants to communicate with multiple devices and the first device it checks is not available then it will get blocked and it will wait for that device to become available, instead of moving on and use the other device which is available. To avoid this problem we can use the select function call.

The select Function Call:

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

int select(int nfds, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set
           *restrict errorfds, struct timeval
           *restrict timeout);
```

int nfds: max fd + 1

readfds: devices you want to monitor for reading

writefds: devices you want to monitor for writing

errorfds: devices you want to monitor for error

timeout: time in sec/micro seconds you want to wait for any of the device to become available.

On successful return, select clears all the descriptors in each of readfds, writefds and errorfds except those descriptors that are ready.

No: Date:

RETURN VALUES OF select:

Success: No of file descriptors that are ready.
Failure : -1

FUNCTIONS USED WITH SELECT:

1. Void FD_CLR(int fd, fd_set *fdset);
   It clears the file descriptor passed as 1st argument from the available file descriptors set passed as the 2nd argument. It is used when the process uses ~~a~~ certain device and it no longer need~~s~~ that device.

2. int FD_ISSET(int fd, fd_set *fdset);
   It checks if the file descriptor passed as 1st argument is available in the list of available file descriptors passed as second argument.

3. void FD_SET(int fd, fd_set *fdset);
   It sets the file descriptor passed as 1st argument in the list of file descriptors passed as 2nd argument. It is used when we want to add a device to be monitored for the current process.

4. Void FD_ZERO(fd_set *fdset);
   It clears all the file descriptors passed to this function.

No: Date:

Example: A function that blocks until one of two fds is ready

```
#include <string.h>
#include <sys/select.h>

int whichisready( int fd1, int fd2){
      int     maxfd;
      int     nfds;
      fd_set    readset;
       maxfd = ( fd1 > fd2)? fd1 : fd2;
       FD_ZERO ( &readset); //Clear garbage
       FD_SET ( fd1, &readset); // add fd1 to readset
       FD_SET(fd2, &readset); //add fd2 to readset
       nfds = select ( maxfd +1, &readset, NULL, NULL, NULL);
       if(FD_ISSET(fd1, &readset))
                 return fd1;
      if (FD_ISSET(fd2, &readset))
                 return fd2;
}
```

No:

Date:

LECTURE #5

THE POLL FUNCTION CALL:

The poll function is similar to select function but it organizes the information by file descriptor rather than by the type of condition. The possible events are stored in a struct pollfd. In contrast, select function organizes information by the type of event and has seperate descriptor masks for read, write and error conditions.

SYNOPSIS:

```
#include <poll.h>
int poll(struct pollfd fds[], nfds_t nfds,
                int timeout);
```

fds: Array of struct pollfd, representing the monitoring information for the file descriptors.

nfds: Number of descriptors to be monitored.

timeout: Time in milliseconds that the poll should wait without recieving an event before returning.

NOTE:

If the timeout value is -1, the poll never times out. Maximum timeout period is 2^{32} milliseconds ≈ 30 mins.

RETURN VALUES:

0 : timeout

-1 : Failure

No of descriptors that have events : Success

The struct pollfd structure includes the following members:

```
int    fds;     // file descriptor
short  events;  // requested events (read, write, error).
short  revents; // returned events
```

VALUES OF EVENT FLAGS FOR THE POLL FUNCTION

Event Flag	Meaning
POLLIN	read other than high priority data without blocking.
POLLRDNORM	read normal data without blocking.
POLLPRI	read high priority data without blocking.
POLLRDBAND	read priority data without blocking.
POLLOUT	write normal data without blocking.
POLLWRNORM	same as POLLOUT
POLLERR	error occured on the descriptor
POLLHUP	device has been disconnected
POLLINVAL	file descriptor invalid

Set events to contain the events to monitor; poll fills the revents with the events that have occured.

Example on book page #153.