1. Who is responsible for linking with libraries? **Linker**
2. File extension after pre-processor? **.i**
3. File extension after translator? **.s**
4. File extension after assembler? **.o**
5. Library is collection of? **predefined functions/pre-compiled code/object files**
6. Types of dynamic linking? **dynamic load time and dynamic run time**
7. Another name for static linking? **Archive file**
8. Another name for dynamic linking? **shared object**
9. Dynamic libraries in windows are called? **dll files**
10. Extension for static library? **.a**
11. Extension for dynamic linking? **.so**
12. Extension for dynamic linking in windows? **.dll**
13. Command for separate executable file? **cc filename.c –o filename**
14. Command for static linking? **cc -static filename.c**
15. Command to check the size of file? **size filename**
16. Linker by default linker links? **dynamic library**
17. What is the command to display table of symbol names and addresses? **nm**
18. What is the command to check the dependencies of the executable file? **ldd filename**
19. What is the command to know the details of the file? **file filename**
20. Abbreviation for ELF? **Executable linkable file**
21. Abbreviation for LSB? **Linux system base**
22. Previous versions of the Linux executable file format? **COFF**
23. Abbreviation for COFF? **Common object file format**
24. Command to create static library? **ar options archivefilename.a objectfiles. . .**
25. rcs? r- **replace or insert** c- **create** s- **indexing**
26. Command to know list of object files in static library? **ar -t filename.a**
27. Command to delete the object file from the static library? **ar d filename.a filename.o**
28. Command to insert object file? **ar r filename .a filename.o**
29. Command to create dynamic library? **cc -shared -o filename.so objectfiles . . .**
30. Command to link with own static library? **cc filename.c filename.a**
31. Command to link with own dynamic library? **cc filename.c (path)filename.so**
32. Name of dynamic c library? **libc.so**
33. If calling function knows where exactly the called function is (at compile time)? **Static binding**
34. If calling function knows where exactly the called function is (at load time)? **Dynamic load time binding**
35. If calling function knows where exactly the called function is (at run time)? **Dynamic run time binding**
36. Mechanism used in dynamic run time linking? **Plug and play**
37. Prototype for dlerror()? **char* dlerror(void);**
38. Prototype for dlopen()? **void* dlopen(const char* filename ,int flag);**
39. Prototype for dlsym()? **void* dlsym(void* handle,const char* symbol);**

40. Prototype for dlclose()? **int dlclose(void *handle);**
41. Header for all the above prototypes is? **dlfcn.h**
42. What is an operating system? **Resource manager/ resource allocator**
43. Components of operating system? **Applications and services**
44. All the services of operating system are called **kernel services.**
45. Kernel is the **core** of the operating system.
46. **Applications** are optional **services** are mandatory.
47. **General purpose** operating system contains all the services necessary and unnecessary
48. **Embedded** operating system contains only the required services
49. After switching on the system the program that executes is **BIOS**
50. BIOS- **basic input output system**
51. Program that checks all the basic input and output peripheral connections **BIOS**
52. MBR- **master boot record**
53. MBR contains **512 bytes** of records related to booting.
54. Who initiates GRUB program – **MBR**
55. GRUB- **grand unified boot loader**
56. Who is boot loader for Linux- **GRUB**
57. Who is boot loader for windows- **NT loader**
58. **LILO** (Linux Loader) is the boot loader for older versions of the Linux.
59. Who checks the number of kernels present? **GRUB**
60. Who starts the INIT program? **Kernel**
61. Which program undergoes execution first? **INIT**
62. Program under execution is called **process**
63. Who is responsible to manage the processes? **Process manager**
64. Command to run the process in background? **(Path)Filename &**
65. Command to display the running processes? **ps**
66. Process ids are assigned by? **process manager**
67. Job ids are assigned by? **Terminal**
68. Command for entire list of all the processes? **ps –e**
69. Command to bring background process to foreground? **fg**
70. fg follows **last in first out** mechanism
71. Prototype of getpid? **pid_t getpid(void);**
72. Prototype of getppid?**pid_t getppid(void);**
73. Header file for the above prototypes? **unistd.h, sys/types.h**
74. Process id of **INIT** is **1**
75. Process id of **scheduler** is **0**
76. Shell is a **command interpreter**
77. Limit of the process ids is **32767**
78. After reaching limit of process ids the counter will reset to **300**
79. Prototype for rand? **int rand(void);**
80. Prototype for srand? **void srand(unsigned int seed);**
81. Header for above prototypes? **stdlib.h.**
82. rand will generate random numbers between **0 to 2147483647**

83. value of **EXIT_FAILURE** macro **1**
84. value of **EXIT_SUCCESS** macro **0**
85. **srand** is useful for generating new sequence of random numbers
86. Time gap between process creation and process 1st instruction execution is **response time**
87. process in its life time ,the amount of time not being executed by CPU is **starvation time**
88. Time gap between creation of process to completion of process is **turnaround time**
89. No. of process completed per unit time is **throughput.**
90. **Less** starvation is good, **less** turnaround is good, **less** response time is good, **more** throughput is good
91. process is ready for execution but not being executed by CPU – **ready state**
92. process waiting for external event- **wait state**
93. process intentionally goes to delay- **delayed state**
94. process suspended by using some signals – **suspend state**
95. Command for list of running process in particular terminal? **ps –e | grep pts/terminalno.**
96. Loading and unloading of PCBs is called **context switching**.
97. Prototype of system function? **int system(const char* command);**
98. Header file for the above prototype is? **stdlib.h**
99. **Library functions** are compiler supported.
    System calls are kernel supported
100. Which function is useful to create a child process- **fork()**
101. Prototype of fork()- **pid_t fork(void);**
102. Header fie for the above prototype? **unistd.h**
103. Duplicate process is **child process**
104. In parent ,child process returns its **id**
105. In child, child returns **zero.**
106. How many process are created if fork () is called 2 times ? **2^(no of times fork called)**
107. printf prints data on monitor? **No**
108. printf sends data to std_out buffer? **Yes**
109. Reasons for std_out buffer flushing? **\n, fflush(), scanf(),terminated process, buffer full.**
110. Whenever executable file is loaded into RAM, names of the file streams opened by default are? **STD_IN, STD_OUT, STD_ERROR**
111. Either parent or child modifies common data then mechanism followed by the fork () is **copy on write** mechanism.
112. Parent runs on **foreground**
113. Child runs on **background**
114. If parent completes its process before child process then child becomes **ORPHAN**
115. If child completes its process before parent process then child becomes **ZOMBIE(dead process)**
116. Prototype of perror() ? **void perror(const char* s);**

117. Which headers are required for above prototype? **stdio.h , errno.h**
118. Set of error messages along with their unique macros are described in **errno.h** header file
119. Child can intimate parent that he completed its task by using ? **exit() or _exit()**
120. Parent can receive child's intimation that he is completed by using? **wait() or waitpid()**
121. **exit()** is a library function.
122. **_exit()** is a system call.
123. Prototype of exit()? **void exit (int status);**
124. Prototype of _exit()? **void _exit(int status);**
125. exit() is a **normal process termination**
126. _exit() is a **immediate process termination**
127. All functions registered with atexit() will be called in **reverse** order.
128. Prototype of atexit() ? **int atexit(void (*function) (void));**
129. Header required for above prototype? **stdlib.h**
130. Return, break , exit() are similar in behaviour? **False**
131. Break and return are **control statements**
132. exit() is a **function call**.
133. Registered functions using atexit() are called by **exit() function.**
134. atexit() function flushes stdio buffer? **True**
135. prototype of wait()? **pid_t wait(int *status);**
136. prototype of waitpid()? **pid_t waitpid(pid_t pid,int *status,int options);**
137. header files required for the above prototypes? **sys/types.h , sys/wait.h**
138. wait system call is a blocking function? **true**
139. another name for zombie process is **defunct process**
140. upon normal termination status is received is 1 what will be the value if status is printed? **256** because of storage into 2nd byte.
141. WIFEXITED (status) - **true if child terminates normally**
142. WEXITSTATUS (status) - **returns exit status of the child**.
143. WIFSIGNALED (status) – **True if child terminated by signal.**
144. WTERMSIG (status) – **returns signal no.**
145. **wait(&status); = waitpid(-1,&status,0);**
146. WNOHANG- **returns immediately if no child has exited**.
147. WUNTRACED – **returns if child has stopped**.
148. WCONTINUED – **returns if child has resumed**.
149. For a process priorities will be set from **0 to 99**
150. Nice – for modification of **scheduling priority.**
151. Renice –for modification of **scheduling priority of running process.**
152. Linux default scheduling policy- **ROUND ROBIN.**
153. Prototype of execl()? **int execl(const char *path,const char *arg, . . .);**
154. Prototype of execlp()? **int execlp(const char *file, const char *arg, . . .);**
155. header file for the above prototypes? **unistd.h**

1. Signals are also called as **software interrupts.**
2. Some of the important signal numbers
   1-SIGHUP
   2-SIGINT
   3-SIGQUIT
   9-SIGKILL
   10-SIGUSR1
   11-SIGSEGV
   12-SIGUSR2
   14-SIGALRM
   17-SIGCHLD
   18-SIGCONT
   19-SIGSTOP
3. Process sends signal to process directly **FALSE**
4. Process sends signal to process via signal manager **TRUE**
5. **Raise function** sends signal to the **caller.**
6. Raise function equivalent kill statement is – **kill(getpid(),signal no.);**
7. **Pause** function **waits (suspends)** the process.
8. Prototype of raise- **int raise(int sig);**
9. Header required for above prototype is **signal.h**
10. Prototype for pause is – **int pause(void);**
11. Header required for the above prototype is **unistd.h**
12. Prototype of signal function – **sighandler_t signal(int signum,sighandler_t handler);**
13. Signals that cannot be caught are **9-SIGKILL,19-SIGSTOP**
14. Signal() will generate the signal **FALSE**
15. Signal () will not block the process **TRUE**
16. Signal () modifies the **signal table** of the process.
17. Prototype of alarm() function – **unsigned int alarm(unsigned int seconds);**
18. Header file required for above prototype is **unistd.h**
19. SIGALRM is a blocking signal **FALSE**
20. SIGALRM terminates the process **TRUE.**
21. It is possible to set multiple alarm() functions at a time **FALSE**
22. Daemon process runs in **background.**
23. Signal function returns **old action** and sets **new action.**
24. Signal function returns function pointer of prototype **-void (*sighandler) (int);**
25. Prototype of sigaction – **int sigaction (int signum, const struct sigaction *act, struct sigaction *oldaction) ;**
26. Header file required for the above prototype is **signal.h**
27. Sigaction can be used in 3 ways
    Sigaction(num,&s,&s1); - set new action and get old action
    Sigaction (num,&s,NULL)- set new action
    Sigaction (num,NULL,&s)- get old action
28. Who sets new action and who returns old action – **signal manager**

29. **SA_NOCLDWAIT** is the flag useful to avoid child becoming zombie.
30. **SA_NOCLDSTOP** – will not receive child stop and resume status but receives child termination.
31. **SA_NODEFER** – signal (same) will be serviced instantly.
32. **SA_RESETHAND**- after one time occurrence action will be set to default.
33. **sigemptyset**-allow all other signals
34. **sigaddset**- allow all except x(signal)
35. **sigfillset**-block all other signals
36. **sigdelset**- block all except x(signal)
37. Who decides the process stack limit- **resource manager**
38. **Soft limit**: it is a limit whenever process gets created the resource manager will assign it.
39. **Hard limit**: it is the limit where process can request upto that limit. Hard limit acts a ceiling for soft limit.
40. Prototype of getrlimit – **int getrlimit(int resource,sturct rlimit *rlim);**
41. Prototype of setrlimit- **int setrlimit(int resource,const struct rlimit *rlim);**
42. Headers required- **sys/time.h , sys/resource.h**
43. **Proc** is a user interface directory
44. **RLIMIT_CPU**- setting time limit for process , receives **SIGXCPU** after specified time limit.
45. **RLIMIT_FSIZE** – setting limit for file size, receives **SIGXFZ** signal if file size exceeds.
46. Prototype for time- **time_t time(time_t *t);**
47. Header file required – **time.h**
48. time() returns number of **seconds** since 1970-01-01
49. Prototype of ctime – **char* ctime(const time_t * timep);** required header- **time.h**
50. **Boot**: contains booting related files.
51. **Super**: file system information(NTFS,FAT etc).
52. **Inode**: information related to file.
53. **Data**: content of the file.
54. **chmod 0NNN filename** for all new permissions
55. **chmod g+w filename** for specific permissions (u- user , g-group, o-others ,+give,-remove)(r-read w-write x-execute)
56. size of one file block – **1024 bytes**
57. prototype of stat- **int stat(const char *path,struct stat * buf);**
58. headers required for above prototype- **sys/types.h, sys/stat.h, unistd.h**
59. **soft link** is also called as symbolic link.
60. Command to hard link files **ln filename filename.**
61. Command to **ln –s filename filename**
62. Prototype of opendir- **DIR* opendir(const char *name);**
63. Header file required for above prototype-**sys/types.h,dirent.h**
64. Prototype of readdir – **struct dirent* readdir(DIR* dirp);** header required – **dirent.h**
65. In linux if filename starts with **dot(.)** then it will be considered as **hidden files.**

66. Prototype of open- **int open(const char *pathname, int flag, mode_t mode);**
67. Headers required are – **sys/types.h,sys/stat.h,fcntl.h**
68. fopen equivalent open calls
    fopen("data","r"); = = open("data",O_RDONLY);
    fopen("data","r+") ; = = open("data",O_RDWR);
    fopen("data","w"); = = open("data",O_WRONLY|O_CREAT|O_TRUNC);
    fopen("data","w+"); = =open("data",O_RDWR|O_CREAT|O_TRUNC);
    fopen("data","a"); = = open("data",O_WRONLY|O_APPEND|O_CREAT);
    fopen("data","a+");= =open("data",O_RDWR|O_APPEND|O_CREAT);
69. prototype for read- **ssize_t read(int fd,void *buf,size_t count**); header – **unistd.h**
70. read() will place '\0' at the end of string. **FALSE**
71. prototype of bzero()- **void bzero(void *s,size_t n);** header – **strings.h**
72. prototype of memset() – **void* memset(void *s, int c, size_t n**); header – **string.h**
73. prototype of write() – **ssize_t write(int fd, const void *buf, size_t count);**
    header – **unistd.h**
74. write and read converts data to **unformatted type** like fwrite and fread does . **TRUE**
75. open() assigns least available file descriptor – **TRUE**
76. **Output redirection:** manipulating the destination of actual output (dest) to other (dest).
77. **Input redirection:** manipulating the access of input from one (dest) to other (dest)
78. Command that acts as output redirection **ls >data**
79. Prototype for pipe- **int pipe(int pipefd[2]);**
80. Prototype for pipe2- **int pipe2(int pipefd[2],int flags);**
81. Header required for above prototypes- **unistd.h**
82. **Pipefd[0]-** read end of pipe **pipefd[1]-** write end of pipe.
83. Pipe is a **unidirectional** form of communication between processes.
84. Pipe is used between two unrelated processes – **FALSE**
85. Communication channel provided by the pipe is **byte stream**
86. Size of pipe – buffer capacity- **65536 bytes.**
87. lseek is not applicable in **pipes.**
88. Another name for FIFO is **named pipe.**
89. FIFO must be open on both ends. **TRUE**
90. Command for creating FIFO- **mkfifo pipename**
91. Prototype for mkfifo – **int mkfifo (const char* pathname,mode_t mode);**
92. Header file required for above prototype – **sys/types.h,sys/stat.h**
93. Size of FIFO – **Zero**
94. Command to create special file/block file- **mknod filename type**

1. **msgget()** is useful for creating as well as opening the message queue.
2. **msgsnd()** is useful for transferring the data into message queue.
3. **msgrcv()** is useful for receiving the data from the message queue.
4. **msgctl()** is useful for doing some control operation on the message queue.
5. Prototype of msgget- **int msgget(key_t key,int msgflg);**
6. Prototype of msgsnd- **int msgsnd(int msqid,const void *msgp, size_t msgz,int msgflg);**
7. Prototype of msgrcv –**ssize_t msgrcv(int msqid,void* msgp,size_t msgsz,long msgtyp, int msgflg);**
8. Prototype of msgctl- **int msgctl(int msgid,int cmd,struct msqid_ds* buf);**
9. Header files required for the above prototypes – <**sys/types.h**>, <**sys/ipc.h**>, <**sys/msg.h**>
10. What is the value of **IPC_PRIVATE** macro? **0**
11. What is the command to display the message queues available? **ipcs –q**
12. Message queue is a **data structure.**
13. Life of message queue is till **system shutdown.**
14. Commands to forcefully remove message queues **ipcrm –Q key / ipcrm –q id.**
15. If message type is **zero** then mechanism followed is **first in first out.**
16. If message type is **greater than zero** then **first in** with respect to **message type** will **first out**.
17. If message type is **less than zero** then message types which are **less than or equal** to **absolute value** of **message type** will be first out.
18. **Message queue** is the slowest ipc.
19. Shared memory ipc is **simplex** form of communication.
20. **shmget()**- for creating as well as sharing memory segment.
21. **shmat()** for attaching the shared memory segment.
22. **shmdt()** for detaching the shared memory segment.
23. **shmctl()** for doing control operations on shared memory.
24. Prototype of shmget()- **int shmget (key_t key, size_t size, int shmflg);**
25. Prototype of shmat()- **void* shmat(int shmid, const void * shmaddr, int shmflg);**
26. Prototype of shmdt()- **int shmdt(const void* shmaddr);**
27. Headers required for the above prototypes – <**sys/ipc.h**>, <**sys/shm.h**>
28. **ipcs –m** is the command for shared memory information.
29. Prototype of dup()-**int dup(int oldfd);**
30. Prototype of dup2()-**int dup2(int oldfd,int newfd);**
31. Header required for the above prototype – <**unistd.h**>
32. Dup() assigns **least** available file descriptor.
33. Dup2() assigns **specified** file descriptor.
34. Dup() family of functions are useful for implementing **pipelining** mechanism.
35. Prototype for fcntl()- **int fcntl(int fd, int cmd, ...);**
36. Headers required for the above prototypes – **unistd.h , fcntl.h**
37. **Dup(fd)** equivalent to **fcntl(fd,F_DUPED,0);**
38. Values of **F_GETLK,F_SETLCK,F_SETLKW** macros. **5 , 6 ,7**
39. Values of **F_RDLCK,F_WRLCK,F_UNLCK** macros. **0 , 1, 2**

40. Values of **SEEK_SET,SEEK_CUR,SEEK_END** macros. **0, 1 , 2**
41. Part of program where processes need to access as a shared resource is called **critical section** of the code.
42. fcntl() is useful to sync **file type** of shared resource **only**.
43. Semaphore is useful to sync **any-kind** of shared resource.
44. Semaphore is a **small positive integer** which is present in the **kernel space.**
45. Semaphore is useful to **sync** the multiple processes try to access the **shared resources**.
46. **semget()** is useful for creating as well as opening the semaphore set.
47. **semctl()** is useful for doing some operation on semaphore value.
48. **semop()** is useful for deciding the process to access the critical section or not.
49. Prototype of semget()- **int semget(key_t key,int nsems,int semflg);**
50. Prototype of semctl()- **int semctl(int semid,int semnum,int cmd, . . .);**
51. Prototype of semop() – **int semop (int semid,sturct sembuf *sops, unsigned int nsops) ;**
52. Headers required for the above prototypes- <**sys/types.h**>, <**sys/ipc.h**>, <**sys/sem.h**>
53. **ipcs –s** is the command to display how many semaphore sets are available.
54. value of **SEM_UNDO** macro? **0x1000**
55. value of **GETVAL** macro? **12**
56. Value of **SETVAL** macro? **16**
57. If sem_op is **positive** then sem_op value is **added** to semaphore value.
58. If sem_op is **zero** then process **waits** for semaphore value to be **zero**.
59. If sem_op is **negative** if semvalue is greater than sem_op then absolute value of sem_op is **subtracted** from semvalue.
60. Value of **IPC_NOWAIT** macro? 04000
61. **O_CREAT- 0100**
62. **O_TRUNC- 01000**
63. **O_APPEND- 02000**
64. **O_NONBLOCK- 04000**
65. Thread is a **light weight process**
66. Context switch between threads is **faster** compared to context switch between processes.
67. Creation of threads is **10times** faster than the creation of process.
68. Threads exchange/ communicate through **global variables.**
69. Prototype of pthread_create- **int pthread_create(pthread_t *thread, const pthread _attr_t* attr, void* (*start_routine) (void *), void * arg);**
70. Thread related programs must be linked with **–pthread** during compilation
71. Prototype of pthread_exit- **void pthread_exit (void* retval);**
72. Prototype of pthread_join- **int pthread_join(pthread_t thread,void ** retval);**
73. Prototype of pthread_mutex_init- **int pthread_mutex_init(pthread_mutex_t * restrict_mutex);**
74. Prototype of pthread_mutex_lock- int **pthread_mutex_lock(pthread_mutex_t* mutex);**

75. Prototype of pthread_mutex_unlock- **int pthread_mutex_unlock(pthread_mutex_t* mutex);**
76. for all the thread related functions header file required is **pthread.h**
77. semaphore is **signalling mechanism** where as mutex is **locking mechanism**
78. using semaphore critical section can be accessed by others –**TRUE**
79. using mutex one alone can access the critical section at a time.**TRUE**
80. If the binding between the opcode and the location where opcode have to occupy
    If this is done at compile time- **absolute code**
    If this is done at load time – **relocatable code**
    It this is done at run time- **executable code**
81. To achieve the binding concept of opcode and its location hardware required is **MMU**
82. Older versions of operating systems use to follow **partitioning mechanism**
83. If the process got more than required memory then the unused memory is called **internal fragmentation.**
84. Overall free memory wise required memory space for new process is available but not contiguously is called **external fragmentation.**
85. Frames in virtual ram are called **pages.**
86. Frame size in ram = page size in virtual memory = **4096**
87. **Page fault** occurs when requested page is not available in physical memory.
88. MMU takes **logical address** and gives **physical address** to cpu.
89. **Swap memory** of virtual memory is utilized when frames of ram gets full.
90. **Read and write/ write access pages** will be loaded to and unloaded from RAM.

1. UNIX system was first described in **1974** by **ken Thompson and Dennis Ritchie**.
2. The file **errno.h** defines the variable errno.
3. The value of error is never set to **0 (zero).**
4. System Calls in UNIX called **service points** for kernel
5. Programmers reference manuals
   Section **1** Commands, Precompiled utilities.
   Section **2** System Calls, For Unix Services.
   Section **3** C Library Facilities.
   Section **4** Header Files.
6. **O_APPEND** Append to the end of file.
7. **O_CREAT** Create the file .
8. **O_EXCL** Exclusively create the file.
9. **O_TRUNC** Truncate the file to 0 bytes.
10. **O_NOCTTY** Don’t make the device the control terminal.
11. **O_NONBLOCK** Non block I/O.
12. **O_SYNC** Synchronous write.
13. Process with ID **0,1 and 2** used by special processes
14. Swapper process has an **ID of 0**
15. init process has an **ID of 1**
16. page daemon process has an **ID of 2**
17. scheduler process also known as **swapper process**

18. Process with ID 0,1 and 2 are **not** created with **fork ,**created by the **kernel** as a part of **bootstrapping**
19. The new process created is called **child process** called once but returns twice
20. File locks set in the parent are **not** inherited by the child.
21. Some versions of operating system combine fork and exec called as **spawning**
22. When a child process terminates the parent is notified by the kernel with **SIGCHLD** signal
23. System function is implemented by calling **fork, exec, waitpid**
24. user reserved signals **SIGUSR1 SIGUSR2**
25. **SIGILL** indicates process executed illegal instruction
26. **SIGKILL** a **sure way to kill** a process **cannot** be **caught or ignored**
27. **SIGSYS** signals invalid system call
28. **SIGURG** notifies a process that an urgent Condition has occurred
29. **Difference** -The kill sends a signal to a **process or group of processes** ,the raise function sends a signal to **itself**
30. pause call causes the process to **suspend** till a signal is **caught**
31. Daemon processes can be started during **system start up** by **/etc/init** ,they **do not have** a **controlling terminal ,**their parent is the **init process (ID 1)**
32. Since processes themselves cannot access the address space of other processes the kernel provides the facility for **IPC**
33. write to a FIFO with no process open for reading signal **SIGPIPE** is generated
34. **Message Queue** are a **linked list** of messages
35. **Fastest ipc** is **shared memory.**
36. **Initial thread** created when main() is invoked by the **process loader**
37. A **CPU scheduler**, running in the **dispatcher**, is responsible for selecting of the **next running process**.
38. All jobs (processes), once submitted, are in the **job queue**
39. All processes that are ready and waiting for execution are in the **ready queue**.
40. **long-term scheduler/job scheduler** selects processes from the job queue to the ready queue.
41. **CPU scheduler/short-term scheduler** selects a process from the ready queue for execution.
42. **CPU utilization**: percentage of the time that CPU is busy.
43. **Throughput**: the number of processes completed per unit time
44. **Turnaround time**: the interval from the time of submission of a process to the time of completion.
45. **Wait time:** the sum of the periods spent waiting in the ready queue
46. **Response time**: the time of submission to the time the first response is produced
47. **fairness:** It is important, but harder to define quantitatively.
48. When do we perform **non-preemptive scheduling**?
    . process switches from the running state to waiting state (e.g. I/O request)
    . A process switches from the running state to the ready state.
    . A process switches from waiting state to ready state (completion of an I/O operation)
    . A process terminates.

1. Predict the output of the following program code.

```
main()
{
  fork();
  printf("Hello World!");
}
```

Answer: Hello World!Hello World!
Explanation: The fork creates a child that is a duplicate of the parent process. The child begins from the fork(). All the statements after the call to fork() will be executed twice.(once by the parent process and other by child). The statement before fork() is executed only by the parent process.

2. Predict the output of the following program code

```
main()
{
  fork(); fork(); fork();
  printf("Hello World!");
}
```

Answer: "Hello World" will be printed 8 times.
Explanation: 2^n times where n is the number of calls to fork().

3. Difference between the fork() and vfork() system call?

During the fork() system call the Kernel makes a copy of the parent process's address space and attaches it to the child process.

But the vfork() system call do not makes any copy of the parent's address space, so it is faster than the fork() system call. The child process as a result of the vfork() system call executes exec() system call. The child process from vfork() system call executes in the parent's address space (this can overwrite the parent's data and stack) which suspends the parent process until the child process exits.

4. What is a shell?

A shell is an interactive user interface to an operating system services that allows an user to enter commands as character strings or through a graphical user interface. The shell converts them to system calls to the OS or forks off a process to execute the command. System call results and other information from the OS are presented to the user through an interactive interface. Commonly used shells are sh,csh,ks etc.

5. What is a binary semaphore? What is its use?

A binary semaphore is one, which takes only 0 and 1 as values. They are used to implement mutual exclusion and synchronize concurrent processes.

6. What are turnaround time and response time?

Turnaround time is the interval between the submission of a job and its completion. Response time is the interval between submission of a request, and the first response to that request.

7. What is process spawning?

When the OS at the explicit request of another process creates a process, this action is called process spawning.