1. Check the following limits: No. of clock ticks, Max. no. of child processes, Max. path length, Max. no. Of characters in a file name, Max. no. of open files/ process

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

int main()
{
printf("Runtime values\n");
printf("The max number of clock ticks : %ld\
n",sysconf(_SC_CLK_TCK));

 printf("The max runtime child processes : %ld\
n",sysconf(_SC_CHILD_MAX));

 printf("The max runtime path length : %ld\
n",pathconf("prg1.c",_PC_PATH_MAX));

 printf("The max characters in a file name : %ld\
n",pathconf("prg1.c",_PC_NAME_MAX));

 printf("The max number of opened files : %ld\
n",sysconf(_SC_OPEN_MAX));

 return 0;

}
```

2. a. Copy of a file using system calls.

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

```
#include <fcntl.h>
#include <stdlib.h>

void typefile (char *filename)
{
int fd, nread;
char buf[1024];
fd = open (filename, O_RDONLY);
if (fd == -1) {
perror (filename);
return;
}
while ((nread = read (fd, buf, sizeof (buf))) > 0)
write (1, buf, nread);
close (fd);
}
int main (int argc, char **argv)
{
int argno;

for (argno = 1; argno < argc; argno++)
{
typefile (argv[argno]);
}
exit (0);
}
```

b. Output the contents of its Environment list

```
#include<stdio.h>
int main(int argc, char* argv[ ])
{
int i;
char **ptr;
```

```
extern char **environ;
for( ptr = environ; *ptr != 0; ptr++ )
printf("%s\n", *ptr);
return 0;
}
```

3. a. Emulate the UNIX ln command

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<string.h>
int main(int argc, char * argv[])
{

if(argc < 3 || argc > 4 || (argc == 4 && strcmp(argv[1],"-s")))
{
printf("Usage: ./a.out [-s] <org_file> <new_link>\n");
return 1;
}
if(argc == 4)
{
     if((symlink(argv[2], argv[3])) == -1)
          printf("Cannot create symbolic link\n") ;
     else
          printf("Symbolic link created\n");
}
else
{
     if((link(argv[1], argv[2])) == -1)
          printf("Cannot create hard link\n");
     else
          printf("Hard link created\n");
}
```

```
    return 0;
}
```

b. Create a child from parent process using fork() and counter counts till 5 in both processes and displays.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
for(int i=0;i<5;i++)
{
if(fork() == 0)
{
printf("[son] pid %d from [parent] pid %d\n",getpid(),getppid());
exit(0);
}
}
for(int i=0;i<5;i++)
wait(NULL);
}
```

4. Illustrate two processes communicating using shared memory.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include<stdio.h>

int main(void) {
pid_t pid;
```

```
int *shared;
int shmid;

shmid = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0666);
printf("Shared Memory ID=%u",shmid);
if (fork() == 0) {

shared = shmat(shmid, (void *) 0, 0);
printf("Child pointer %p\n", shared);
*shared=1;
printf("Child value=%d\n", *shared);
sleep(2);
printf("Child value=%d\n", *shared);
}
else {

shared = shmat(shmid, (void *) 0, 0);
printf("Parent pointer %p\n", shared);
printf("Parent value=%d\n", *shared);
sleep(1);
*shared=42;
printf("Parent value=%d\n", *shared);
sleep(5);
shmctl(shmid, IPC_RMID, 0);

}
}
```

5. Demonstrate producer and consumer problem using semaphores.

Producer:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>

#define MAXSIZE 10
#define FIFO_NAME "myfifo"

int main()
{
int fifoid; int fd, n;
char *w;
int open_mode;
system("clear");
w=(char*)malloc(sizeof(char)*MAXSIZE);
open_mode=O_WRONLY;
fifoid=mkfifo(FIFO_NAME, 0755);
if(fifoid==-1)
{
printf("\nError: Named pipe cannot be Created\n");
exit(0);
}
if((fd=open(FIFO_NAME, open_mode)) < 0 )
{
printf("\nError: Named pipe cannot be opened\n");
exit(0);
}
while(1)
{
printf("\nProducer :");
fflush(stdin);
read(0, w, MAXSIZE);
n=write(fd, w, MAXSIZE);
if(n > 0)
```

```
printf("\nProducer sent: %s", w);
}
}
```

Consumer:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>

#define MAXSIZE 10
#define FIFO_NAME "myfifo"

int main()
{
int fifoid;
int fd, n;
char *r;
system("clear");
r=(char *)malloc(sizeof(char)*MAXSIZE);
int open_mode=O_RDONLY;
if( (fd=open(FIFO_NAME, open_mode)) < 0 )
{
printf("\nError: Named pipe cannot be opened\n");
exit(0);
}
while(1)
{
n=read(fd, r, MAXSIZE);
if(n > 0)
printf("\nConsumer read: %s", r);
}
}
```

6. Demonstrate round robin scheduling algorithm and calculates average waiting time and average
turnaround time.

```
#include<stdio.h>

int main()
{
int i, limit, total = 0, x, counter = 0, time_quantum;
int wait_time = 0, turnaround_time = 0, arrival_time[10],
burst_time[10], temp[10];

float average_wait_time, average_turnaround_time;

printf("\nEnter Total Number of Processes:\t");
scanf("%d", &limit);
x = limit;
for(i = 0; i < limit; i++)
{
printf("\nEnter Details of Process[%d]\n", i + 1);
printf("Arrival Time:\t");
scanf("%d", &arrival_time[i]);
printf("Burst Time:\t");
scanf("%d", &burst_time[i]);
temp[i] = burst_time[i];
}
printf("\nEnter Time Quantum:\t");
scanf("%d", &time_quantum);
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting
Time\n");
for(total = 0, i = 0; x != 0;)
{
if(temp[i] <= time_quantum && temp[i] > 0)
```

```
{
total = total + temp[i];
temp[i] = 0;
counter = 1;
}
else if(temp[i] > 0)
{
temp[i] = temp[i] - time_quantum;
total = total + time_quantum;
}
if(temp[i] == 0 && counter == 1)
{
x--;
printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1,
burst_time[i], total - arrival_time[i],
total - arrival_time[i] - burst_time[i]);
wait_time = wait_time + total - arrival_time[i] - burst_time[i];
turnaround_time = turnaround_time + total - arrival_time[i];
counter = 0;
}
if(i == limit - 1)
{
i = 0;
}
else if(arrival_time[i + 1] <= total)
{
i++;
}
else
{
i = 0;
}
}
average_wait_time = wait_time * 1.0 / limit;
```

```
average_turnaround_time = turnaround_time * 1.0 / limit;
printf("\n\nAverage Waiting Time:\t%f", average_wait_time);
printf("\n\nAvg Turnaround Time:\t%f\n",
average_turnaround_time);
return 0;
}
```

7. Implement priority-based scheduling algorithm and calculates average waiting time and average
turnaround time.

```
#include<stdio.h>
int main()
{
int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n;
int total=0,pos,temp,avg_wt,avg_tat;
printf("Enter Total Number of Process:");
scanf("%d",&n);
printf("\nEnter Burst Time and Priority\n");
for(i=0;i<n;i++)
{
printf("\nP[%d]\n",i+1);
printf("Burst Time:");
scanf("%d",&bt[i]);
printf("Priority:");
scanf("%d",&pr[i]);
p[i]=i+1;
}

for(i=0;i<n;i++)
{
pos=i;
for(j=i+1;j<n;j++)
{
```

```
if(pr[j]<pr[pos])
pos=j;
} temp=pr[i];
pr[i]=pr[pos];
pr[pos]=temp;
temp=bt[i];
bt[i]=bt[pos];
bt[pos]=temp;
temp=p[i];
p[i]=p[pos];
p[pos]=temp;
}
wt[0]=0;

for(i=1;i<n;i++)
{
wt[i]=0;
for(j=0;j<i;j++)
wt[i]+=bt[j];
total+=wt[i];
}
avg_wt=total/n;
total=0;
printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround
Time");
for(i=0;i<n;i++)
{
tat[i]=bt[i]+wt[i];
total+=tat[i];
printf("\nP[%d]\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
}
avg_tat=total/n;
printf("\n\nAverage Waiting Time=%d",avg_wt);
printf("\nAverage Turnaround Time=%d\n",avg_tat);
```

```
return 0;
}
```

8. Act as sender to send data in message queues and receiver that reads data from message queue.

Reciever:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>


struct mesg_buffer {

    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    key = ftok("progfile", 65);

    msgid = msgget(key, 0666 | IPC_CREAT);

    msgrcv(msgid, &message, sizeof(message), 1, 0);

    printf("Data Received is : %s \n",  message.mesg_text);

    msgctl(msgid, IPC_RMID, NULL);
```

```
    return 0;
}
```

Writer:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10


struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main() {
    key_t key;
    int msgid;

    key = ftok("progfile", 65);

    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;
    printf("Write Data : ");
    fgets(message.mesg_text,MAX,stdin);

    msgsnd(msgid, &message, sizeof(message), 0);

    printf("Data send is : %s \n", message.mesg_text);
    return 0;
}
```

9. Where a parent writes a message to pipe and child reads message from pipe.
Producer:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>

#define MAXSIZE 10
#define FIFO_NAME "myfifo"

int main()
{
int fifoid;
int fd, n;
char *w;

system("clear");
w=(char *)malloc(sizeof(char)*MAXSIZE);
int open_mode=O_WRONLY;
fifoid=mkfifo(FIFO_NAME, 0755);
if(fifoid==-1)
{
printf("\nError: Named pipe cannot be Created\n");
exit(0);
}
if( (fd=open(FIFO_NAME, open_mode)) < 0 )
{
printf("\nError: Named pipe cannot be opened\n");
exit(0);
}
while(1)
{
```

```
printf("\nProducer :");
fflush(stdin);
read(0, w, MAXSIZE);
n=write(fd, w, MAXSIZE);
if(n > 0)
printf("\nProducer sent: %s", w);
}
}
```

Consumer:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>

#define MAXSIZE 10
#define FIFO_NAME "myfifo"

int main()
{
int fifoid;
int fd, n;
char *r;

system("clear");
r = (char *)malloc(sizeof(char)*MAXSIZE);
int open_mode = O_RDONLY;
if( (fd=open(FIFO_NAME, open_mode)) < 0 )
{
printf("\nError: Named pipe cannot be opened\n");
exit(0);
}
while(1)
```

```
{
n=read(fd, r, MAXSIZE);
if(n > 0)
printf("\nConsumer read: %s", r);
}
}
```

10. Demonstrate setting up a simple web server and host website on your own Linux computer.

11. a. Create two threads using pthread, where both thread counts until 100 and joins later.

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
#include<stdlib.h>

void* myturn(void *arg)
{
   for(int i=1;i<=20;i++)
   {
      sleep(1);
      printf("process 1: i=%d\n",i);
   }
   return NULL;
}

void yourturn()
{
```

```
    for(int i=1;i<=10;i++)
    {
        sleep(2);
        printf("process 2: j=%d\n",i);
    }
}

int main()
{
    pthread_t newthread;

    pthread_create(&newthread,NULL,myturn,NULL);
    yourturn();
    pthread_join(newthread,NULL);
    return 0;
}
```

b. Create two threads using pthreads. Here, main thread creates 5 other threads for 5 times and each new thread print "Hello World" message with its thread number.

12. Using Socket APIs establish communication between remote and local processes.