1.**Develop a C program to implement the Process system calls [fork (), exec(), wait(), create process, terminate process.**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
   pid_t child_pid;
   int status;

   // Create a child process using fork()
   child_pid = fork();

   if (child_pid < 0)
   {
      perror("Fork failed");
      exit(1);
   }
   else if (child_pid == 0)
   {
      // This code is executed by the child process
      printf("Child process (PID: %d) is running.\n", getpid());

      // Replace the child process with a new program using exec()
      char *args[] = {"/bin/ls", NULL}; // Replace " " with the path to your child program
      execvp(args[0], args);

      // If exec() fails, this code will be executed
      perror("Exec failed");
      exit(1);
   }
   else
   {
      // This code is executed by the parent process
      printf("Parent process (PID: %d) is waiting for the child to complete.\n", getpid());

      // Wait for the child process to complete using wait()
      wait(&status);

      if (WIFEXITED(status))
      {
         printf("Child process (PID: %d) has completed with status %d.\n", child_pid,
WEXITSTATUS(status));
      }
   }
```

```
    return 0;
}
```

OUTPUT:

```
ashwini@ashwini-HP-ProBook-440-G3:~$ gedit p1.c
ashwini@ashwini-HP-ProBook-440-G3:~$ gcc p1.c
ashwini@ashwini-HP-ProBook-440-G3:~$ ./a.out
Parent process (PID: 3925) is waiting for the child to complete.
Child process (PID: 3926) is running.
 18CS52-CN-20231108T072300Z-001.zip              Pictures
 18CS53-DBMS-20231108T072212Z-001.zip                    pr
 18CS71-AIML-20231108T072139Z-001.zip                    pr.c
'18CSL76-AIML LAB-20231108T072148Z-001.zip'              pri
 d                                              program1.c
 d.c                                            program.c
 deadlock.c                                     prpr
 deadlocknew.c                                  prpr2
 Desktop                                        prpr2.c
 dl.c                                           Public
 Documents                                      q.c
 Downloads                                      r
Child process (PID: 3926) has completed with status 0.
```

2.Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a) **FCFS:**

AIM:

To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process name and the burst time
Step 4: Set the waiting of the first process as _0'and its burst time as its turnaround time
Step 5: for each process in the Ready Q calculate
    a). Waiting time (n) = waiting time (n-1) + Burst time (n-1)
    b). Turnaround time (n) = waiting time(n)+Burst time(n)
Step 6: Calculate
    a) Average waiting time = Total waiting Time / Number of process \
    b) Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop the process

SOURCE CODE:

```
#include<stdio.h>
void main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
```

```
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
}
```

OUTPUT:

```
Enter the number of processes -- 3
Enter Burst Time for Process 0 -- 24
Enter Burst Time for Process 1 -- 3
Enter Burst Time for Process 2 – 3

PROCESS      BURST TIME          WAITING TIME       TURNAROUND TIME
     P0              24                    0                  24
     P1               3                   24                  27
     P2               3                   27                  30
Average Waiting Time -- 17.000000
Average Turnaround Time -- 27.000000
```

b) SJF SHORTEST JOB FIRST:

AIM:
To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Pre-emption)
DESCRIPTION:
To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.
ALGORITHM:
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as _0' and its turnaround time as its burst time.
Step 6: Sort the processes names based on their Burt time
Step 7: For each process in the ready queue, calculate
    a. Waiting time(n)= waiting time (n-1) + Burst time (n-1)
    b. Turnaround time (n)= waiting time(n)+Burst time(n)
Step 8: Calculate \
    c. Average waiting time = Total waiting Time / Number of process
    d. Average Turnaround time = Total Turnaround Time / Number of process
Step 9: Stop the process

SOURCE CODE:
```
#include<stdio.h>
int main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg,tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
```

```
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
return 0;
}
```

OUTPUT:

```
Enter the number of processes -- 4
Enter Burst Time for Process 0 -- 6
Enter Burst Time for Process 1 -- 8
Enter Burst Time for Process 2 -- 7
Enter Burst Time for Process 3 – 3
```

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24

```
Average Waiting Time -- 7.000000
Average Turnaround Time -- 13.000000
```

c) ROUND ROBIN:

AIM: To simulate the CPU scheduling algorithm round - robin.

DESCRIPTION: To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time - slot and the loop continues until all the processes are completed.

ALGORITHM:
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time
process (n)/time slice
Step 5: If the burst time is less than the time slice then the no. of time slices =1.
Step 6: Consider the ready queue is a circular Q, Calculate

a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]);
ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
```

```
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t) {
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
}
else {
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++){
wa[i]=tat[i]-
ct[i]; att+=tat[i];
awt+=wa[i];}
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
return 0;
}
```

OUTPUT:

```
Enter the no of processes -- 6
Enter Burst Time for process 1 -- 5
Enter Burst Time for process 2 -- 6
Enter Burst Time for process 3 -- 7
Enter Burst Time for process 4 -- 9
Enter Burst Time for process 5 -- 2
Enter Burst Time for process 6 -- 3
Enter the size of time slice – 3
```

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	5	14	19
2	6	16	22
3	7	22	29
4	9	23	32
5	2	12	14
6	3	14	17

```
The Average Turnaround time is -- 22.166666
The Average Waiting time is -- 16.833334
```

d). PRIORITY:

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

DESCRIPTION: To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround time
Step 6: Arrange the processes based on process priority
Step 7: For each process in the Ready Queue calculate
Step 8: For each process in the Ready Queue calculate
a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process
e) Print the results in an order.
Step 9: Stop

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg,tatavg;
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i); scanf("%d%d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k])
{
temp=p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
```

```
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n);
printf("\nAverage Turnaround Time is --- %f",tatavg/n);
return 0;
}
```

OUTPUT:

```
Enter the number of processes --- 5
Enter the Burst Time & Priority of Process 0 ---        10        3
Enter the Burst Time & Priority of Process 1 ---        1         1
Enter the Burst Time & Priority of Process 2 ---        2         4
Enter the Burst Time & Priority of Process 3 ---        1         5
Enter the Burst Time & Priority of Process 4 ---        5         2

PROCESS         PRIORITY        BURST TIME      WAITING TIME    TURNAROUND TIME
        1               1               1               0               1
        4               2               5               1               6
        0               3               10              6               16
        2               4               2               16              18
        3               5               1               18              19

Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.000000
```

3)Develop a C program to simulate producer-consumer problem using semaphores.

AIM: To Write a C program to simulate producer-consumer problem using semaphores.

DESCRIPTION: Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

```
#include <stdio.h>
 #include <stdlib.h>
 int mutex = 1;
 int full = 0;
 int empty = 3, x = 0;
 void producer()
 {
 --mutex;
 ++full;
 --empty;
 x++;
 printf("\nProducer produces item %d",x);
 ++mutex;
 }
 void consumer()
 {
 --mutex;
 --full;
 ++empty;
 printf("\nConsumer consumes item %d",x);
 x--;
 ++mutex;
 }
 int main()
 {
 int n, i;
 printf("\n1. Press 1 for Producer\n2. Press 2 for Consumer\n3. Press 3 for Exit");
 #pragma omp critical
 for (i = 1; i > 0; i++)
 {
printf("\nEnter your choice:");
 scanf("%d", &n);
  switch (n)
 {
 case 1:
 if ((mutex == 1) && (empty != 0))
```

```
 {
 producer();
 }
 else
 {
 printf("Buffer is ful");
  }
 break;

 case 2:
 if ((mutex == 1) && (full != 0))
  {
 consumer();
 }
 else
 {
 printf("Buffer is empty!");
 }
 break;
 case 3:
 exit(0);
 break;
 }
 }
  }
```

OUTPUT:

```
ashwini@ashwini-HP-ProBook-440-G3:~$ ./a.out
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer produces item 1
Enter your choice:1

Producer produces item 2
Enter your choice:1

Producer produces item 3
Enter your choice:1
Buffer is ful
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2
```

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:3

4.**Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program**.

Reader Process:

```
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#define MAX_BUF 1024
int main()
{
  int fd;
  /* A temporary FIFO file is not created in reader */
  char *myfifo = "/tmp/myfifo";
  char buf[MAX_BUF];
  /* Open the named pipe for reading */
  fd = open(myfifo, O_RDONLY);
  /* Read data from the FIFO */
  read(fd, buf, MAX_BUF);
  printf("Writer: %s\n", buf);
  /* Close the FIFO */
  close(fd);
  return 0;
}
```

Writer Process:

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
  int fd;
  char buf[1024];
  /* Create the named pipe (FIFO) */
  char *myfifo = "/tmp/myfifo";
  mkfifo(myfifo, 0666);
  printf("Run Reader process to read the FIFO File\n");
  /* Open the named pipe for writing */
  fd = open(myfifo, O_WRONLY);
  /* Write data to the FIFO */
  strcpy(buf, "Hello from Writer Process");
  write(fd, buf, sizeof(buf));
```

```
  /* Close the FIFO */
  close(fd);
  /* Remove the FIFO */
  unlink(myfifo);
  return 0;
}
```

To execute the program, first compile the writer process and the reader process separately:

```
gcc -o writer writer.c
gcc -o reader reader.c
```

5.Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.

```
#include <stdio.h>

// Function to check if the system is in a safe state
int isSafe(int processes, int resources, int max[][resources], int allocated[][resources], int
available[]) {
  int need[processes][resources];
  int finish[processes];
  for (int i = 0; i < processes; i++)
{
    finish[i] = 0;
    for (int j = 0; j < resources; j++)
{
      need[i][j] = max[i][j] - allocated[i][j];
    }
  }
  int work[resources];
  for (int i = 0; i < resources; i++)
{
    work[i] = available[i];
  }
  int safe = 0;
  while (1)
{
    int found = 0;
    for (int i = 0; i < processes; i++)
{
      if (!finish[i])
{
        int canAllocate = 1;
        for (int j = 0; j < resources; j++)
{
          if (need[i][j] > work[j])
{
            canAllocate = 0;
            break;
          }
        }
        if (canAllocate)
{
          for (int j = 0; j < resources; j++)
{
            work[j] += allocated[i][j];
          }
          finish[i] = 1;
          found = 1;
          break;
        }
      }
```

```
    }
    if (!found)
{
      break;
    }
  }
  for (int i = 0; i < processes; i++)
{
    if (!finish[i])
{
      return 0;
    }
  }
  return 1;
}
int main()
{
  int processes, resources;
  printf("Enter the number of processes: ");
  scanf("%d", &processes);
  printf("Enter the number of resources: ");
  scanf("%d", &resources);
  int max[processes][resources];
  int allocated[processes][resources];
  int available[resources];
  printf("\nEnter the maximum resource matrix:\n");
  for (int i = 0; i < processes; i++)
{
    for (int j = 0; j < resources; j++)
{
      scanf("%d", &max[i][j]);
    }
  }
  printf("\nEnter the allocated resource matrix:\n");
  for (int i = 0; i < processes; i++)
{
    for (int j = 0; j < resources; j++)
{
      scanf("%d", &allocated[i][j]);
    }
  }
  printf("\nEnter the available resources:\n");
  for (int i = 0; i < resources; i++)
{
    scanf("%d", &available[i]);
  }

  if (isSafe(processes, resources, max, allocated, available))
{
```

```
        printf("\nThe system is in a safe state.\n");
      }
    else
    {
       printf("\nThe system is not in a safe state.\n");
      }

     return 0;
    }
```

OUTPUT:

Enter the number of processes: 2
Enter the number of resources: 2
Enter the maximum resource matrix:
1
2
34
5
Enter the allocated resource matrix:
8
9
5
6
Enter the available resources:
4
8
The system is not in a safe state.

Enter the number of processes: 2
Enter the number of resources: 2

Enter the maximum resource matrix:
1
2
3
4
Enter the allocated resource matrix:
9
8
7
6
Enter the available resources:
10
20
The system is in a safe state.

6.Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.

Worst fit

```
#include<stdio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
static int bf[max],ff[max];
printf("\n\tMemory Management Scheme - Worst Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
```

```
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
}
```

O/P

Memory Management Scheme - Worst Fit

Enter the number of blocks:4

Enter the number of files:3

Enter the size of the blocks:-

Block 1:300

Block 2:200

Block 3:400

Block 4:500

Enter the size of the files :-

File 1:260

File 2:450

File 3:350

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	260	4	500	240
2	450	0	0	0
3	350	3	400	50

b)Best fit

```
#include<stdio.h>
void main()
{
   int fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
   static int barray[20],parray[20];

   printf("\n\t\t\tMemory Management Scheme - Best Fit");
   printf("\nEnter the number of blocks:");
   scanf("%d",&nb);
   printf("Enter the number of processes:");
   scanf("%d",&np);

   printf("\nEnter the size of the blocks:-\n");
   for(i=1;i<=nb;i++)
   {
      printf("Block no.%d:",i);
      scanf("%d",&b[i]);
   }
    printf("\nEnter the size of the processes :-\n");
   for(i=1;i<=np;i++)
   {
      printf("Process no.%d:",i);
      scanf("%d",&p[i]);
   }
    for(i=1;i<=np;i++)
   {
      for(j=1;j<=nb;j++)
      {
         if(barray[j]!=1)
         {
            temp=b[j]-p[i];
            if(temp>=0)
```

```
            if(lowest>temp)
            {
               parray[i]=j;
               lowest=temp;
            }
         }
      }

      fragment[i]=lowest;
      barray[parray[i]]=1;
      lowest=10000;
   }
     printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
   for(i=1;i<=np && parray[i]!=0;i++)
      printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

OUTPUT:

Memory Management Scheme - Best Fit

Enter the number of blocks:4

Enter the number of processes:3

Enter the size of the blocks:-

Block no.1:300

Block no.2:200

Block no.3:400

Block no.4:500

Enter the size of the processes :-

Process no.1:260

Process no.2:450

Process no.3:350

Process_no	Process_size	Block_no	Block_size	Fragment
1	260	1	300	40
2	450	4	500	50
3	350	3	400	50

c)First fit

```
#include <stdio.h>
 #include <stdlib.h>
 #define max 25
 void main()
 {

int frag[max], b[max], f[max], i, j, nb, nf, temp;
static int bf[max], ff[max];
printf("\n\tMemory Management Scheme - First Fit");
printf("\nEnter the number of blocks:");
scanf("%d", &nb);
printf("Enter the number of files:");
scanf("%d", &nf);
printf("\nEnter the size of the blocks:-\n");
for (i = 1; i <= nb; i++)
{
printf("Block %d:", i);
scanf("%d", &b[i]);
}
printf("Enter the size of the files :-\n");
for (i = 1; i <= nf; i++)
{
printf("File %d:", i);
scanf("%d", &f[i]);
}
for (i = 1; i <= nf; i++)
{
for (j = 1; j <= nb; j++)
{
if (bf[j] != 1)
{
temp = b[j] - f[i];
```

```
if (temp >= 0)
{
ff[i] = j;
break;
}
}
}
frag[i] = temp;
bf[ff[i]] = 1;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for (i = 1; i <= nf; i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
exit(0);
 }
```

OUTPUT:

Memory Management Scheme - First Fit

Enter the number of blocks:4

Enter the number of files:3

Enter the size of the blocks:-

Block 1:300

Block 2:200

Block 3:400

Block 4:500

Enter the size of the files :-

File 1:260

File 2:450

File 3:350

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	260	1	300	40
2	450	4	500	50
3	350	3	400	50

7)Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

a)FIFO

```
#include <stdio.h>
 #include <stdlib.h>
int fr[3];
void main()
{
void display();
int i, j, page[12] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};
int flag1 = 0, flag2 = 0, pf = 0, frsize = 3, top = 0;
for (i = 0; i < 3; i++)
{
fr[i] = -1;
}
for (j = 0; j < 12; j++)
{
flag1 = 0;
5.
flag2 = 0;

for (i = 0; i < 12; i++)
{
if (fr[i] == page[j])
{
flag1 = 1;
flag2 = 1;
break;
}
}
if (flag1 == 0)
{
for (i = 0; i < frsize; i++)
{
```

```
if (fr[i] == -1)
{
fr[i] = page[j];
flag2 = 1;
break;
}
}
}
if (flag2 == 0)
{
fr[top] = page[j];
top++;
pf++;
if (top >= frsize)
top = 0;
}
display();
}
printf("Number of page faults : %d ", pf + frsize);
exit(0);
 }
 void display()
 {
int i;
printf("\n");
for (i = 0; i < 3; i++)
printf("%d\t", fr[i]);
 }
```

OUTPUT:

shwini@ashwini-HP-ProBook-440-G3:~$ ./a.out

```
2       -1      -1
2       3 -1
2       3 -1
2       3 1
5       3 1
5       2 1
5       2 4
5       2 4
3       2 4
3       2 4
3       5 4
3       5 2     Number of page faults : 9
```

b)LRU

```
#include <stdio.h>
 #include <stdlib.h>
 int fr[3];
 void main()
{
void display();
int p[12] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2}, i, j, fs[3];
int index, k, l, flag1 = 0, flag2 = 0, pf = 0, frsize = 3;
for (i = 0; i < 3; i++)
{
fr[i] = -1;
}
for (j = 0; j < 12; j++)
{
flag1 = 0, flag2 = 0;
for (i = 0; i < 3; i++)
{
if (fr[i] == p[j])
{
flag1 = 1;
flag2 = 1;
break;
}
}
if (flag1 == 0)
{
for (i = 0; i < 3; i++)
{
if (fr[i] == -1)
{
fr[i] = p[j];
flag2 = 1;
```

```
break;
}
}
}
if (flag2 == 0)
{
for (i = 0; i < 3; i++)
fs[i] = 0;
for (k = j - 1, l = 1; l <= frsize - 1; l++, k--)
{
for (i = 0; i < 3; i++)
{
if (fr[i] == p[k])
fs[i] = 1;
}
}
for (i = 0; i < 3; i++)
{
if (fs[i] == 0)
index = i;
}
fr[index] = p[j];
pf++;
}
display();
}
printf("\n no of page faults :%d", pf + frsize);
exit(0);
 }
void display()
 {
int i;
printf("\n");
```

```
for (i = 0; i < 3; i++)
printf("\t%d", fr[i]); }
```

O/P

```
ashwini@ashwini-HP-ProBook-440-G3:~$ ./a.out

                2 -1    -1
                2 3     -1
                2 3     -1
                2 3     1
                2 5     1
                2 5     1
                2 5     4
                2 5     4
                3 5     4
                3 5     2
                3 5     2
                3 5     2
 no of page faults :7
```

8. Simulate following File Organization Techniques: a) Single level directory b) Two level directory

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_FILES 5
#define MAX_NAME_LENGTH 20

typedef struct {
    char name[MAX_NAME_LENGTH];
    int size;
} File;

typedef struct {
    char name[MAX_NAME_LENGTH];
    int numFiles;
    File files[MAX_FILES];
} Directory;

Directory singleLevelDirectory;
Directory twoLevelDirectory[MAX_FILES];

void initializeSingleLevelDirectory() {
    strcpy(singleLevelDirectory.name, "Root");
    singleLevelDirectory.numFiles = 0;
}

void initializeTwoLevelDirectory() {
    for (int i = 0; i < MAX_FILES; ++i) {
        sprintf(twoLevelDirectory[i].name, "Directory%d", i + 1);
        twoLevelDirectory[i].numFiles = 0;
    }
```

```
}

void displaySingleLevelDirectory() {
    printf("Single Level Directory:\n");
    printf("Directory Name: %s\n", singleLevelDirectory.name);
    printf("Number of Files: %d\n", singleLevelDirectory.numFiles);
    printf("Files:\n");
    for (int i = 0; i < singleLevelDirectory.numFiles; ++i) {
        printf("File Name: %s, Size: %d KB\n", singleLevelDirectory.files[i].name,
singleLevelDirectory.files[i].size);
    }
    printf("\n");
}

void displayTwoLevelDirectory() {
    printf("Two Level Directory:\n");
    for (int i = 0; i < MAX_FILES; ++i) {
        printf("Directory Name: %s\n", twoLevelDirectory[i].name);
        printf("Number of Files: %d\n", twoLevelDirectory[i].numFiles);
        printf("Files:\n");
        for (int j = 0; j < twoLevelDirectory[i].numFiles; ++j) {
            printf("File Name: %s, Size: %d KB\n", twoLevelDirectory[i].files[j].name,
twoLevelDirectory[i].files[j].size);
        }
        printf("\n");
    }
}

void addFileSingleLevelDirectory(char name[], int size) {
    if (singleLevelDirectory.numFiles < MAX_FILES) {
        strcpy(singleLevelDirectory.files[singleLevelDirectory.numFiles].name, name);
        singleLevelDirectory.files[singleLevelDirectory.numFiles].size = size;
        singleLevelDirectory.numFiles++;
```

```
        printf("File '%s' added to Single Level Directory\n", name);
    } else {
        printf("Single Level Directory is full, cannot add file '%s'\n", name);
    }
}

void addFileTwoLevelDirectory(char name[], int size, int directoryIndex) {
    if (directoryIndex >= 0 && directoryIndex < MAX_FILES) {
        if (twoLevelDirectory[directoryIndex].numFiles < MAX_FILES) {

strcpy(twoLevelDirectory[directoryIndex].files[twoLevelDirectory[directoryIndex].numFiles].name
, name);
            twoLevelDirectory[directoryIndex].files[twoLevelDirectory[directoryIndex].numFiles].size
= size;
            twoLevelDirectory[directoryIndex].numFiles++;
            printf("File '%s' added to Directory '%s'\n", name,
twoLevelDirectory[directoryIndex].name);
        } else {
            printf("Directory '%s' is full, cannot add file '%s'\n",
twoLevelDirectory[directoryIndex].name, name);
        }
    } else {
        printf("Invalid Directory Index for Two Level Directory\n");
    }
}

int main() {
    initializeSingleLevelDirectory();
    initializeTwoLevelDirectory();

    addFileSingleLevelDirectory("file1.txt", 1024);
    addFileSingleLevelDirectory("file2.txt", 2048);
    addFileSingleLevelDirectory("file3.txt", 3072);
```

```
    displaySingleLevelDirectory();

    addFileTwoLevelDirectory("file4.txt", 4096, 0);
    addFileTwoLevelDirectory("file5.txt", 5120, 1);
    addFileTwoLevelDirectory("file6.txt", 6144, 2);
   displayTwoLevelDirectory();
   return 0;
}
```

OUTPUT:

```
File 'file1.txt' added to Single Level Directory
File 'file2.txt' added to Single Level Directory
File 'file3.txt' added to Single Level Directory
Single Level Directory:
Directory Name: Root
Number of Files: 3
Files:
File Name: file1.txt, Size: 1024 KB
File Name: file2.txt, Size: 2048 KB
File Name: file3.txt, Size: 3072 KB

File 'file4.txt' added to Directory 'Directory1'
File 'file5.txt' added to Directory 'Directory2'
File 'file6.txt' added to Directory 'Directory3'
Two Level Directory:
Directory Name: Directory1
Number of Files: 1
Files:
File Name: file4.txt, Size: 4096 KB

Directory Name: Directory2
Number of Files: 1
Files:
File Name: file5.txtFile Name: file6.txt, Size: 6144 KB

Directory Name: Directory4
Number of Files: 0
Files:

Directory Name: DirectoryNumber of Files: 0
Files:
```

9.Develop a C program to simulate the Linked file allocation strategies.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
 int main()
{
int f[50],p,i,j,k,a,st,len,n,c;
for(i=0;i<50;i++)
f[i]=0;
printf("Enter how many blocks that are already allocated");
scanf("%d",&p);
printf("\nEnter the blocks no.s that are already allocated");
for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
X:
printf("Enter the starting index block & length");
scanf("%d%d",&st,&len);
k=len;
for(j=st;j<(k+st);j++)
{
if(f[j]==0)
{
f[j]=1;
printf("\n%d->%d",j,f[j]);
}
else
{
printf("\n %d->file is already allocated",j);
k++;
}
```

```
}
printf("\n If u want to enter one more file ?(yes -1/no -0)");
scanf("%d",&c);
if (c == 1)
goto X;
else
return 0;
}
```

Output:

```
ashwini@ashwini-HP-ProBook-440-G3:~$ gedit linknew.c
ashwini@ashwini-HP-ProBook-440-G3:~$ gcc linknew.c
ashwini@ashwini-HP-ProBook-440-G3:~$ ./a.out
Enter how many blocks that are already allocated5
Enter the blocks no.s that are already allocated6 4 3 8 9
Enter the starting index block & length1 4
1->1
2->1
 3->file is already allocated
 4->file is already allocated
5->1
 6->file is already allocated
7->1
 If u want to enter one more file ?(yes -1/no -0)0
```

10.Develop a C program to simulate SCAN disk scheduling algorithm.

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

int main() {
 int queue[20],head,n,i,j,seekTime=0,direction,maxTrack;
 printf("Enter the number of disk requests: ");
 scanf("%d", &n);
 printf("Enter the disk request queue: ");
for (i=0;i<n;i++)
{
 scanf("%d",&queue[i]);
 }
printf("Enter the initial head position: ");
scanf("%d",&head);
 printf("Enter the maximum track number: ");
scanf("%d",&maxTrack);
 printf("Enter the direction (0 for left, 1 for right): ");
scanf("%d",&direction);
printf("\n");
int temp;
 for (i=0;i< n-1;i++) {
 for (j=i+1;j<n;j++) {
if(queue[i] > queue[j]) {
 temp = queue[i];
 queue[i] = queue[j];
 queue[j] = temp;
}
 }
 }
 int currentTrack = head;
printf("Seek Sequence: ");
if(direction == 0) {
```

```
 for(i = head;i>=0;i--) {
 printf("%d ",i);
seekTime += abs(currentTrack - i);
        currentTrack = i;
}
 printf("0");
 seekTime += currentTrack;
 for(i = 1; i<= maxTrack; i++) {
 printf("%d",i);
seekTime += abs(currentTrack-i);
 currentTrack = i;
 }
 } else {
for(i=head; i<=maxTrack; i++) {
printf("%d",i);
seekTime += abs(currentTrack - i);
currentTrack = i;
}
printf("%d", maxTrack);
seekTime += abs(currentTrack - maxTrack);
for(i = maxTrack-1;i>= 0;i--) {
 printf("%d",i);
seekTime += abs(currentTrack - i);
currentTrack=i;
}
}
 printf("\n\nTotal Seek Time: %d\n",seekTime);
exit(0);
}
```

Output:

shwini@ashwini-HP-ProBook-440-G3:~$ gcc sacannew.c

ashwini@ashwini-HP-ProBook-440-G3:~$ ./a.out

Enter the number of disk requests: 5

Enter the disk request queue: 98 183 37 122 14

Enter the initial head position: 53

Enter the maximum track number: 199

Enter the direction (0 for left, 1 for right): 1

Seek Sequence:

53545556575859606162636465666768697071727374757677787980818283848586878889909192
93949596979899100101102103104105106107108109110111112113114115116117118119120121 1
22123124125126127128129130131132133134135136137138139140141142143144145146147148
14915015115215315415515615715815916016116216316416516616716816917017117217317417
51761771781791801811821831841851861871881891901911921931941951961971981991991981
97196195194193192191190189188187186185184183182181180179178177176175174173172171
17016916816716616516416316216116015915815715615515415315215115014914814714614514
41431421411401391381371361351341331321311301291281271261251241231221211201191181
17116115114113112111110109108107106105104103102101100999897969594939291908988878 6
85848382818079787776757473727170696867666564636261605958575655545352515049484746
45444342414039383736353433323130292827262524232221201918171615141312111098765432
10

Total Seek Time: 345