NAME : SANTHOSH S
REGISTER NUMBER : 192324200
SUBJECT :OPERATING SYSTEM
SUBJECT CODE:CSA0477

1.Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h> int
main()
{ printf("Process ID: %d\n", getpid() );
  printf("Parent Process ID: %d\n", getpid() ); return
  0;
}
```

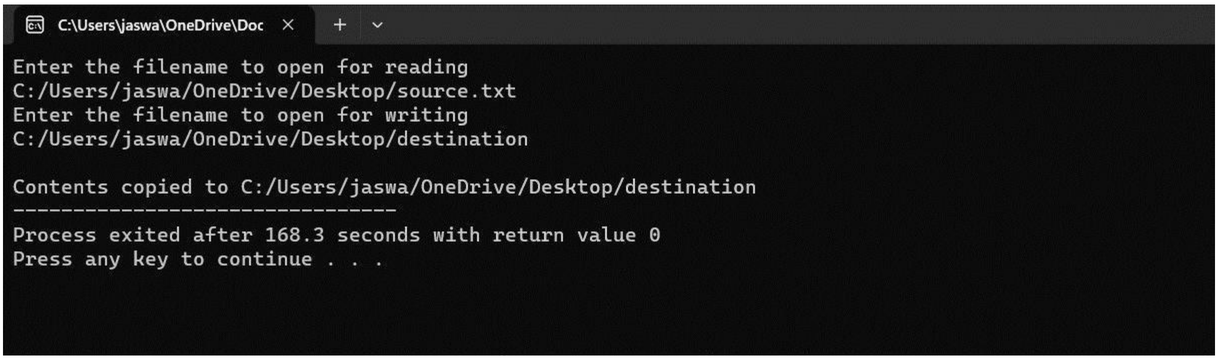
OUTPUT

2. Identify the system calls to copy the content of one file to another and illustrate the same using a C program

PROGRAM:

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

int main()
{
    FILE *fptr1, *fptr2; char
    filename[100], c;

    printf("Enter the filename to open for reading \n");
    scanf("%s", filename);
    fptr1 = fopen(filename, "r"); if
    (fptr1 == NULL)
    { printf("Cannot open file %s \n", filename);
        exit(0);
    }


    printf("Enter the filename to open for writing \n");
    scanf("%s", filename);
    fptr2 = fopen(filename, "w"); if
    (fptr2 == NULL)
    { printf("Cannot open file %s \n", filename);
        exit(0);
    }       c       =
    fgetc(fptr1);
    while (c != EOF)
    { fputc(c, fptr2); c
        = fgetc(fptr1);
    }


    printf("\nContents copied to %s", filename);


    fclose(fptr1);
    fclose(fptr2);
    return 0;

    }
```

OUTPUT



3. Design a CPU scheduling program with C using First Come First Served technique with the following considerations.

a. All processes are activated at time 0.

b. Assume that no process waits on I/O devices.

PROGRAM:

```
#include <stdio.h>

int main() {
    int A[100][4]; // A[][0]: Process ID, A[][1]: Burst Time, A[][2]:
Waiting Time, A[][3]: Turnaround Time
    int i, j, n, total = 0, index, temp;
    float avg_wt, avg_tat;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    printf("Enter Burst Time:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &A[i][1]);
        A[i][0] = i + 1; // Process ID
    }
```

```
// Sorting processes based on Burst Time
for (i = 0; i < n; i++) {
    index = i;
    for (j = i + 1; j < n; j++) {
        if (A[j][1] < A[index][1]) {
            index = j;
        }
    }

    // Swap Burst Time
    temp = A[i][1];
    A[i][1] = A[index][1];
    A[index][1] = temp;

    // Swap Process ID
    temp = A[i][0];
    A[i][0] = A[index][0];
    A[index][0] = temp;
}

// Calculating Waiting Time
A[0][2] = 0; // First process has no waiting time
for (i = 1; i < n; i++) {
    A[i][2] = 0;
    for (j = 0; j < i; j++) {
        A[i][2] += A[j][1];
    }
    total += A[i][2];
}
avg_wt = (float)total / n; // Average Waiting Time
total = 0;
```

```
    // Calculating Turnaround Time
    printf("\nP\tBT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        A[i][3] = A[i][1] + A[i][2]; // Turnaround Time = Burst Time + Waiting Time
        total += A[i][3];
        printf("P%d\t%d\t%d\t%d\n", A[i][0], A[i][1], A[i][2], A[i][3]);
    }
    avg_tat = (float)total / n; // Average Turnaround Time

    printf("Average Waiting Time = %.2f\n", avg_wt);
    printf("Average Turnaround Time = %.2f\n", avg_tat);

    return 0;
  }
}
```

OUTPUT

```
Enter number of process: 4
Enter Burst Time:
P1: 12
P2: 14
P3: 15
P4: 16
P       BT      WT      TAT
P1      12      0       12
P2      14      12      26
P3      15      26      41
P4      16      41      57
Average Waiting Time= 19.750000
Average Turnaround Time= 34.000000
--------------------------------
Process exited after 17.9 seconds with return value 0
Press any key to continue . . .
```

4. Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

PROGRAM:

```
#include<stdio.h>
int main()
{ int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
  float avg_wt,avg_tat;
  printf("Enter    number    of    process:");
  scanf("%d",&n);    printf("nEnter    Burst
  Time:\n"); for(i=0;i<n;i++)
  { printf("p%d:",i+1);
    scanf("%d",&bt[i])
    ; p[i]=i+1; }
    for(i=0;i<n;i++){
    pos=i;
    for(j=i+1;j<n;j++)
    { if(bt[j]<bt[pos])
        pos=j;
    } 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=(float)total/n;
total=0;
printf("nProcesst   Burst Time       tWaiting TimetTurnaround Time\n");
for(i=0;i<n;i++)
{ tat[i]=bt[i]+wt[i];
    total+=tat[i];
    printf("np%dtt
    %dtt
     %dttt%d",p[i],bt
    [i],wt[i],tat[i]);
} avg_tat=(float)total/n; printf("nnAverage Waiting
Time=%f",avg_wt); printf("nAverage Turnaround
Time=%fn",avg_tat);
}
```

OUTPUT

```
Enter number of process:3
nEnter Burst Time:
p1:45
p2:32
p3:18
nProcesst    Burst Time    tWaiting TimetTurnaround Time
np3tt  18tt    0ttt18np2tt  32tt    18ttt50np1tt  45tt    50ttt95nnAverage Waiting Time=22.666666nAverage Turnaround Time=54.333332n
--------------------------------
Process exited after 8.49 seconds with return value 0
Press any key to continue . . .
```

5. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

Program:-

```
#include<stdio.h> struct
priority_scheduling { char
  process_name; int
burst_time; int
waiting_time; int
turn_around_time; int
priority; }; int main() {
  int number_of_process; int
  total = 0;
  struct priority_scheduling temp_process; int
  ASCII_number = 65;
  int          position;          float
  average_waiting_time;       float
  average_turnaround_time;
  printf("Enter the total number of Processes: "); scanf("%d",
  & number_of_process);
  struct priority_scheduling process[number_of_process]; printf("\nPlease Enter the
  Burst Time and Priority of each process:\n"); for (int i = 0; i < number_of_process;
  i++) {
    process[i].process_name = (char) ASCII_number;
    printf("\nEnter   the    details of    the    process       %c    \n",
        process[i].process_name);
    printf("Enter the burst time: "); scanf("%d", &
    process[i].burst_time); printf("Enter the
    priority: "); scanf("%d", &
    process[i].priority); ASCII_number++; }
  for (int i = 0; i < number_of_process; i++) {
    position = i;
    for (int j = i + 1; j < number_of_process; j++) {
      if (process[j].priority > process[position].priority) position
```

```
= j; } temp_process = process[i];
process[i] =
process[position]; process[position] = temp_process;
} process[0].waiting_time =
0; for (int i = 1; i < number_of_process;
i++) {
process[i].waiting_time = 0;
for (int j = 0; j < i; j++) {
process[i].waiting_time += process[j].burst_time;           }
total += process[i].waiting_time; } average_waiting_time =
(float) total / (float) number_of_process; total = 0;

printf("\n\nProcess_name \t Burst Time \t Waiting Time \t           Turnaround
Time\n");
printf("_______________________________________________\n");
for (int i = 0; i < number_of_process; i++) {
process[i].turn_around_time = process[i].burst_time +
process[i].waiting_time;
printf("\t %c \t\t %d \t\t %d \t\t %d", process[i].process_name, process[i].burst_time,
process[i].waiting_time, process[i].turn_around_time);
printf("\n_______________________________________________\n"); }
average_turnaround_time = (float) total / (float) number_of_process; printf("\n\n
Average Waiting Time : %f", average_waiting_time); printf("\n Average
Turnaround Time: %f\n", average_turnaround_time); return 0;
```

OUTPUT:

```
Enter the total number of Processes: 3
Please Enter the  Burst Time and Priority of each process:
Enter the details of the process A
Enter the burst time: 2
Enter the priority: 1
Enter the details of the process B
Enter the burst time: 10
Enter the priority: 3
Enter the details of the process C
Enter the burst time: 6
Enter the priority: 2

Process_name     Burst Time      Waiting Time     Turnaround Time
------------------------------------------------------------
          B               10               0               10
------------------------------------------------------------
          C               6                10              16
------------------------------------------------------------
          A               2                16              18
------------------------------------------------------------

 Average Waiting Time : 8.666667
 Average Turnaround Time: 14.666667
```

6. Construct a C program to simulate Round Robin scheduling algorithm with C.

Program:-

```
#include<stdio.h>
#include<conio.h> int
main()
{ int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10]; float
   avg_wt, avg_tat; printf(" Total number of process in the system: ");
   scanf("%d", &NOP); y = NOP;
for(i=0; i<NOP; i++)
{ printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1); printf("
Arrival time is: \t"); scanf("%d", &at[i]);
printf(" \nBurst time is: \t");
scanf("%d", &bt[i]); temp[i] =
bt[i];
}
printf("Enter the Time Quantum for the process: \t"); scanf("%d", &quant);
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time "); for(sum=0, i =
0; y!=0; ) { if(temp[i] <= quant && temp[i] > 0)
{ sum = sum + temp[i];
   temp[i] = 0; count=1;
   } else if(temp[i] > 0)
   { temp[i] = temp[i] - quant; sum
      = sum + quant;
   }
   if(temp[i]==0 && count==1)
   { y--;
         printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum- at[i], sum-
at[i]-bt[i]); wt = wt+sum-at[i]-
      bt[i]; tat = tat+sum-at[i];
      count =0;
   } if(i==NOP-
   1)
```

```
{ i=0; } else
if(at[i+1]<=sum)
{ i++;
} else
{ i=0;
}
} avg_wt = wt *
1.0/NOP; avg_tat = tat
* 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt); printf("\n
Average Waiting Time: \t%f", avg_tat); getch();
}
```

OUTPUT

```
 Total number of process in the system: 3

 Enter the Arrival and Burst time of the Process[1]
 Arrival time is:       2

Burst time is:  33334

 Enter the Arrival and Burst time of the Process[2]
 Arrival time is:       23

Burst time is:  45

 Enter the Arrival and Burst time of the Process[3]
 Arrival time is:       27

Burst time is:  67
Enter the Time Quantum for the process:         9

 Process No              Burst Time              TAT             Waiting Time
Process No[2]            45                              121                     76
Process No[3]            67                              175                     108
Process No[1]            33334                           33444                   110
 Average Turn Around Time:      98.000000
 Average Waiting Time:  11246.666992
```

7. Construct a C program to implement non-preemptive SJF algorithm

PROGRAM:

```
#include <stdio.h>

int main() {
    int at[10], bt[10], pr[10];
    int n, i, j, temp, time = 0, count, over = 0, sum_wait = 0, sum_turnaround = 0, start;
    float avgwait, avgturn;

    printf("Enter the number of processes\n");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("Enter the arrival time and execution time for process %d\n", i + 1);
        scanf("%d%d", &at[i], &bt[i]);
        pr[i] = i + 1;
    }

    // Sorting based on arrival time
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (at[i] > at[j]) {
                temp = at[i]; at[i] = at[j]; at[j] = temp;
                temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
                temp = pr[i]; pr[i] = pr[j]; pr[j] = temp;
            }
        }
    }

    printf("\n\nProcess\t| Arrival time\t| Execution time\t| Start time\t| End time\t|
Waiting time\t| Turnaround time\n\n");

    // Main scheduling logic
    while (over < n) {
        count = 0;

        // Count processes that have arrived
        for (i = over; i < n; i++) {
            if (at[i] <= time)
                count++;
            else
                break;
        }
```

```
        // Sorting based on execution time (burst time) if multiple processes are available
        if (count > 1) {
            for (i = over; i < over + count - 1; i++) {
                for (j = i + 1; j < over + count; j++) {
                    if (bt[i] > bt[j]) {
                        temp = at[i]; at[i] = at[j]; at[j] = temp;
                        temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
                        temp = pr[i]; pr[i] = pr[j]; pr[j] = temp;
                    }
                }
            }
        }

        // Process execution
        start = time;
        time += bt[over];

        printf("p[%d]\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\n",
               pr[over], at[over], bt[over], start, time,
               time - at[over] - bt[over], time - at[over]);

        sum_wait += time - at[over] - bt[over];
        sum_turnaround += time - at[over];
        over++;
    }

    avgwait = (float)sum_wait / (float)n;
    avgturn = (float)sum_turnaround / (float)n;

    printf("Average waiting time is %.2f\n", avgwait);
    printf("Average turnaround time is %.2f\n", avgturn);

    return 0;
}}
```

OUTPUT

```
Enter the number of processes
3
Enter the arrival time and execution time for process 1
 1 3
Enter the arrival time and execution time for process 2
2 6
Enter the arrival time and execution time for process 3
3 8


Process |Arrival time   |Execution time |Start time     |End time       |waiting

p[1]    |       1       |       3       |       0       |       3       |       -1      |       2
p[2]    |       2       |       6       |       3       |       9       |       1       |       7
p[3]    |       3       |       8       |       9       |       17      |       6       |       14
Average waiting time is 2.000000
Average turnaround time is 7.666667

--------------------------------
Process exited after 20.18 seconds with return value 0
Press any key to continue . . .
```

8. Construct a C program to simulate Round Robin scheduling algorithm with C.

PROGRAM:

```
#include<stdio.h> #include<conio.h> int main() { int i, NOP, sum=0,count=0,
y, quant, wt=0, tat=0, at[10], bt[10], temp[10]; float avg_wt, avg_tat; printf("
Total number of process in the system: "); scanf("%d", &NOP); y = NOP;
for(i=0; i<NOP; i++) { printf("\n Enter the Arrival and Burst time of the
Process[%d]\n", i+1); printf(" Arrival time is: \t"); scanf("%d", &at[i]);

scanf("%d", &bt[i]); temp[i] = bt[i]; } printf("Enter
the Time Quantum for the process: \t");
scanf("%d", &quant);
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time "); for(sum=0, i
= 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0)
{ sum = sum + temp[i];
    temp[i] = 0; count=1;
    }
    else if(temp[i] > 0)
    { temp[i] = temp[i] - quant; sum
        = sum + quant;
    }
    if(temp[i]==0 && count==1)
    { y--;
        printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum- at[i], sum-
at[i]-bt[i]); wt = wt+sum-at[i]-
        bt[i]; tat
```

```
        = tat+sum-at[i]; count =0;
    }
    if(i==NOP-1)
    { i=0;
    }
    else if(at[i+1]<=sum)          {
        i++;      }
    else        {
        i=0; }
}
avg_wt = wt * 1.0/NOP; avg_tat
= tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt); printf("\n
Average Waiting Time: \t%f", avg_tat); getch();
}
```

OUTPUT

```
 Total number of process in the system: 4

 Enter the Arrival and Burst time of the Process[1]
 Arrival time is:       1

Burst time is:  23

 Enter the Arrival and Burst time of the Process[2]
 Arrival time is:       2

Burst time is:  32

 Enter the Arrival and Burst time of the Process[3]
 Arrival time is:       3

Burst time is:  2

 Enter the Arrival and Burst time of the Process[4]
 Arrival time is:       4

Burst time is:  45
Enter the Time Quantum for the process:         5

 Process No             Burst Time              TAT             Waiting Time
Process No[3]           2                               9                               7
Process No[1]           23                              64                              41
Process No[2]           32                              85                              53
Process No[4]           45                              98                              53
 Average Turn Around Time:      38.500000
 Average Waiting Time:  64.000000
```

9. Illustrate the concept of inter-process communication using shared memory with a C program

AIM:

To implement the concept of inter-process communication using shared memory using C programming.

ALGORITHM:

1. Create a shared memory segment:

- Use shmget() function to create a new shared memory segment or get the identifier of an existing one.
- Ensure to handle errors if the shared memory creation fails.

2. Attach shared memory to processes:

- Use shmat() function to attach the shared memory segment to the process address space.
- This allows processes to read and write data to the shared memory.

3. Read/Write data in shared memory:

- Processes can read and write data directly to the shared memory location.
- Ensure proper synchronization mechanisms (like semaphores) are used to avoid race conditions and maintain data consistency.

4. Detach shared memory and clean up:

- Use shmdt() function to detach the shared memory segment from the process when done.
- Optionally, remove the shared memory segment using shmctl() with the IPC_RMID command.

PROGRAM:

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

#define SHM_SIZE 1024 // Size of the shared memory segment int main() {
    key_t key = ftok("shmfile", 65); // Generate a unique key for the shared
memory segment
```

```
    // Create a new shared memory segment (or get the identifier of an existing
one) int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666); if (shmid
    == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }

    // Attach the shared memory segment to the process address space
    char *shm_ptr = (char*)shmat(shmid, NULL, 0); if
    (shm_ptr == (char*)(-1)) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }

    // Write data to the shared memory strcpy(shm_ptr,
    "Hello, shared memory!");

    // Detach the shared memory segment from the process if
    (shmdt(shm_ptr) == -1) {
        perror("shmdt");
        exit(EXIT_FAILURE);
    }
printf("Data written to shared memory: %s\n", shm_ptr);

    // Optional: Remove the shared memory segment if
    (shmctl(shmid, IPC_RMID, NULL) == -1)
        { perror("shmctl");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

OUTPUT:

```vbnet
Data written to shared memory: Hello, shared memory!
```

10. Illustrate the concept of inter-process communication using message queue with a c program

PROGRAM :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct message {
    long msg_type;
    char msg_text[100];
};

int main() { key_t key = ftok("msgqfile", 65); // Generate a unique key for
    the message
queue

    // Create a new message queue (or get the identifier of an existing one) int msgid
    = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }

    struct message msg;
    msg.msg_type = 1; // Message type (can be any positive number)

    // Producer: Send a message to the message queue
    strcpy(msg.msg_text, "Hello, message queue!"); if
    (msgsnd(msgid, (void*)&msg, sizeof(msg.msg_text),
IPC_NOWAIT) == -1) { perror("msgsnd");
        exit(EXIT_FAILURE);
    }
printf("Producer: Data sent to message queue: %s\n", msg.msg_text);

    // Consumer: Receive a message from the message queue if
    (msgrcv(msgid, (void*)&msg, sizeof(msg.msg_text), 1, 0) == -1) {
    perror("msgrcv"); exit(EXIT_FAILURE);
    }

    printf("Consumer: Data received from message queue: %s\n",
msg.msg_text);
```

```
    // Remove the message queue if
    (msgctl(msgid, IPC_RMID, NULL) == -1) {
    perror("msgctl"); exit(EXIT_FAILURE);
    }

    return 0;
}
```

OUTPUT :

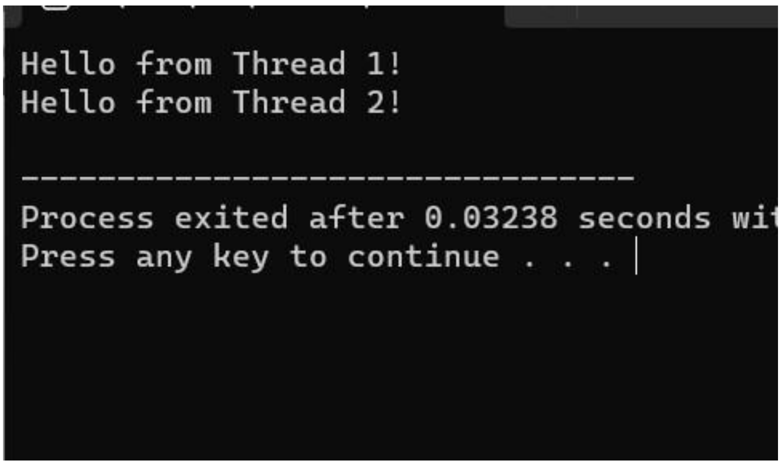
```
arduino

Producer: Data sent to message queue: Hello, message queue!
Consumer: Data received from message queue: Hello, message queue!
```

11. Illustrate the concept of multithreading using a C program

PROGRAM :

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

void* threadFunction(void* arg) { char*
    message = (char*)arg; printf("%s\n",
    message);
    return NULL;
}

int main() { pthread_t
    thread1, thread2;
    char* message1 = "Hello from Thread 1!"; char*
    message2 = "Hello from Thread 2!";

    // Create threads
    pthread_create(&thread1, NULL, threadFunction, (void*)message1);
    pthread_create(&thread2, NULL, threadFunction, (void*)message2);

    // Wait for threads to complete pthread_join(thread1,
        NULL); pthread_join(thread2, NULL);

    return 0;
}
```

OUTPUT :



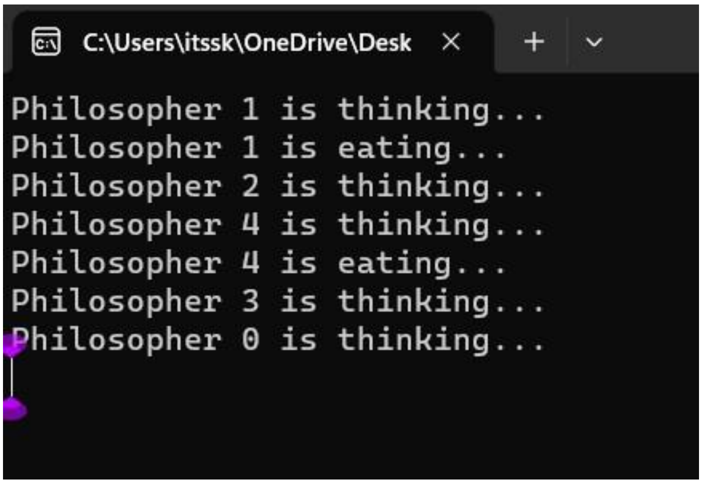
12. Design a C program to simulate the concept of Dining-Philosophers

PROGRAM :

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

#define NUM_PHILOSOPHERS 5

pthread_mutex_t chopsticks[NUM_PHILOSOPHERS];

void* philosopherLifeCycle(void* arg) { int id =
    *((int*)arg); int
    left_chopstick = id;
    int right_chopstick = (id + 1) % NUM_PHILOSOPHERS;

      while (1) {
        // Think printf("Philosopher %d is
       thinking...\n", id);

       // Pick up chopsticks pthread_mutex_lock(&chopsticks[left_chopstick]);
       pthread_mutex_lock(&chopsticks[right_chopstick]);

       // Eat printf("Philosopher %d is
       eating...\n", id);
       sleep(rand() % 3 + 1); // Eating time

       // Put down chopsticks
       pthread_mutex_unlock(&chopsticks[left_chopstick]);
       pthread_mutex_unlock(&chopsticks[right_chopstick]);
```

```
        // Repeat the cycle
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS]; int
    philosopher_ids[NUM_PHILOSOPHERS];

    // Initialize mutex locks
    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        pthread_mutex_init(&chopsticks[i], NULL);
    }

    // Create philosopher threads for (int i = 0; i <
    NUM_PHILOSOPHERS;           ++i)           {
    philosopher_ids[i]              =               i;
    pthread_create(&philosophers[i],          NULL,
    philosopherLifeCycle,
(void*)&philosopher_ids[i]);
    }

    // Wait for threads to finish (although they run indefinitely) for (int i =
    0; i < NUM_PHILOSOPHERS; ++i) {
        pthread_join(philosophers[i], NULL);
    }

    // Destroy mutex locks
    for (int i = 0; i < NUM_PHILOSOPHERS; ++i)
    { pthread_mutex_destroy(&chopsticks[i]); }


    return 0;
}
```

OUTPUT :



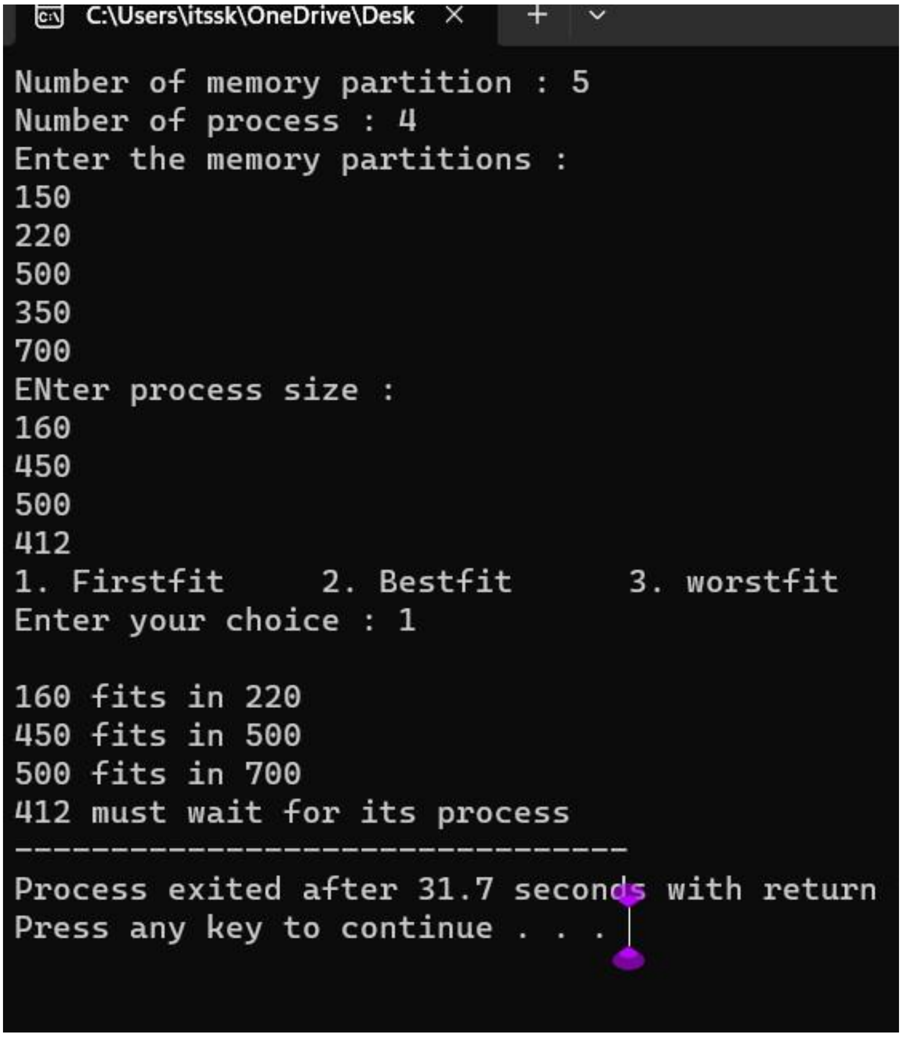
13. **Construct a C program to implement various memory allocation strategies.**

PROGRAM :

```
#include<stdio.h>
void bestfit(int mp[],int p[],int m,int n){ int j=0; for(int
        i=0;i<n;i++){ if(mp[i]>p[j]){ printf("\n%d fits in
        %d",p[j],mp[i]); mp[i]=mp[i]-p[j++]; i=i-1;
                }
        }
        for(int i=j;i<m;i++)
        {
                printf("\n%d must wait for its process",p[i]);
        }
}


void rsort(int a[],int n){ for(int
        i=0;i<n;i++){ for(int
        j=0;j<n;j++){ if(a[i]>a[j]){ int
        t=a[i]; a[i]=a[j]; a[j]=t;
                        }
                }
        }
}


void sort(int a[],int n){ for(int
        i=0;i<n;i++){ for(int
        j=0;j<n;j++){ if(a[i]<a[j]){ int
        t=a[i]; a[i]=a[j]; a[j]=t;
                        }
                }
```

```
        }
}
void firstfit(int mp[],int p[],int m,int
        n){ sort(mp,n); sort(p,m);
        bestfit(mp,p,m,n);
}
void worstfit(int mp[],int p[],int m,int n){
        rsort(mp,n); sort(p,m);
        bestfit(mp,p,m,n);
} int main(){ int m,n,mp[20],p[20],ch; printf("Number
of memory partition : "); scanf("%d",&n);
printf("Number of process : "); scanf("%d",&m);
printf("Enter the memory partitions : \n"); for(int
i=0;i<n;i++){ scanf("%d",&mp[i]);
        }
        printf("ENter process size : \n");
        for(int i=0;i<m;i++){
        scanf("%d",&p[i]);

        }
        printf("1. Firstfit\t2. Bestfit\t3. worstfit\nEnter your choice :
        "); scanf("%d",&ch); switch(ch){ case 1: bestfit(mp,p,m,n);
        break; case 2: firstfit(mp,p,m,n); break; case 3:
        worstfit(mp,p,m,n); break;
        default:
                printf("invalid");
                break;
        }
}
```

OUTPUT :



```
C:\Users\itssk\OneDrive\Desk

Number of memory partition : 5
Number of process : 4
Enter the memory partitions :
150
220
500
350
700
ENter process size :
160
450
500
412
1. Firstfit     2. Bestfit      3. worstfit
Enter your choice : 1

160 fits in 220
450 fits in 500
500 fits in 700
412 must wait for its process
--------------------------------
Process exited after 31.7 seconds with return
Press any key to continue . . .
```

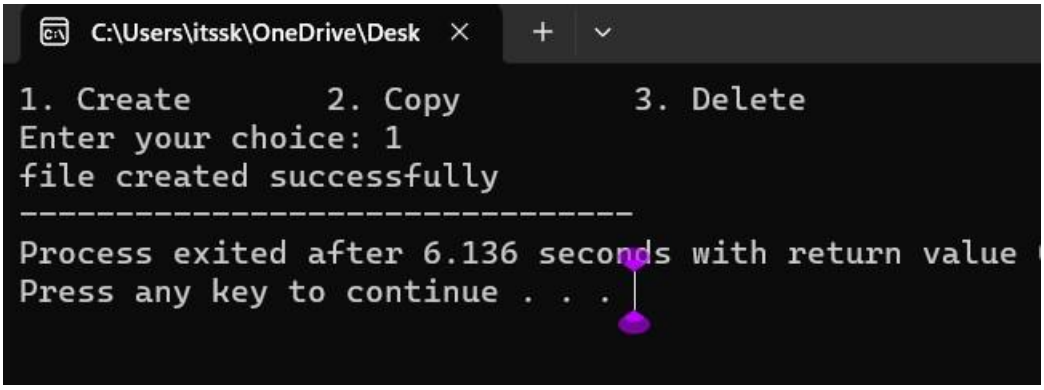
14. Construct a C program to organize the file using single level directory

PROGRAM :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFFER_SIZE 4096
void copy(){ const char
*sourcefile=
"C:/Users/itssk/OneDrive/Desktop/sasi.txt";             const             char
   *destination_file="C:/Users/itssk/OneDrive/Desktop/sk.txt"; int source_fd
   = open(sourcefile, O_RDONLY); int dest_fd = open(destination_file,
   O_WRONLY | O_CREAT | O_TRUNC,
0666); char buffer[BUFFER_SIZE]; ssize_t bytesRead, bytesWritten; while
   ((bytesRead = read(source_fd, buffer, BUFFER_SIZE)) > 0) { bytesWritten =
   write(dest_fd, buffer, bytesRead);
   }
   close(source_fd);
   close(dest_fd);
   printf("File copied successfully.\n");
} void
create()
{ char
path[100];
        FILE *fp; fp=fopen("C:/Users/itssk/OneDrive/Desktop/sasi.txt","w");
        printf("file created successfully");
}
int main(){
        int n;
```

```
printf("1. Create \t2. Copy \t3. Delete\nEnter your choice: "
); scanf("%d",&n); switch(n){
        case 1:
create(); break;
case 2:
        copy();
        break;
        case 3:
                remove("C:/Users/itssk/OneDrive/Desktop/sasi.txt"); printf("Deleted
                successfully");
}}
```

OUTPUT :



15. **Design a C program to organize the file using two level directory structure.**

PROGRAM :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> int main() { char mainDirectory[] =
"C:/Users/itssk/OneDrive/Desktop"; char subDirectory[] = "os";
    char   fileName[]   =   "example.txt";
    char          filePath[200];          char
    mainDirPath[200];
   snprintf(mainDirPath, sizeof(mainDirPath), "%s/%s/", mainDirectory, subDirectory);
          snprintf(filePath, sizeof(filePath), "%s%s", mainDirPath, fileName); FILE *file
    = fopen(filePath, "w"); if (file ==
    NULL)  {  printf("Error  creating
    file.\n"); return 1;
    } fprintf(file, "This is an example file
content."); printf("File created successfully:
%s\n"); }
```

OUTPUT :

```
File created successfully

--------------------------------
Process exited after 1.379 seconds with return value 0
Press any key to continue . . .
```

16. Develop a C program for implementing random access file for processing the employee details

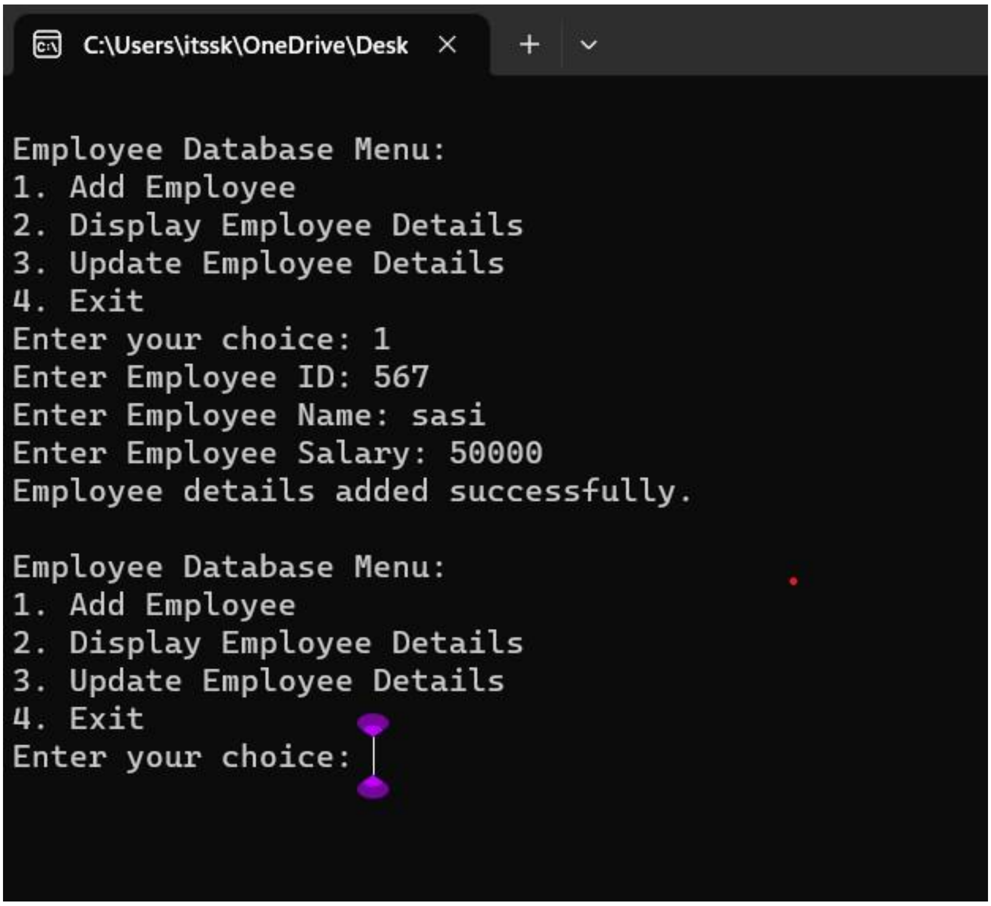
PROGRAM :

```
#include      <stdio.h>
#include      <stdlib.h>
struct Employee {
    int empId; char
    empName[50];
    float empSalary;};
int main() { FILE *filePtr; struct Employee
    emp; filePtr = fopen("employee.dat",
    "rb+"); if
    (filePtr == NULL) { filePtr =
        fopen("employee.dat", "wb+"); if
        (filePtr == NULL) { printf("Error
            creating the file.\n"); return 1;
                }
    } int
    choice; do
    {
        printf("\nEmployee          Database  Menu:\n");
        printf("1. Add Employee\n");
        printf("2.    Display    Employee    Details\n");
        printf("3.    Update    Employee    Details\n");
        printf("4.                Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice); switch
        (choice) {
            case 1:
                printf("Enter Employee ID: "); scanf("%d",
                &emp.empId); printf("Enter Employee
                Name: ");
```

```
            scanf("%s",         emp.empName);
            printf("Enter Employee Salary: ");
            scanf("%f", &emp.empSalary);
            fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET); fwrite(&emp,  sizeof(struct    Employee),     1,
               filePtr); printf("Employee details added
            successfully.\n"); break;
        case 2:
            printf("Enter Employee ID to display: "); scanf("%d",
            &emp.empId);
            fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);
            fread(&emp, sizeof(struct Employee), 1, filePtr);
            printf("Employee ID: %d\n", emp.empId); printf("Employee
            Name: %s\n", emp.empName); printf("Employee Salary:
            %.2f\n", emp.empSalary); break;
        case 3:
            printf("Enter Employee ID to update: ");
            scanf("%d", &emp.empId);
            fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);
            fread(&emp, sizeof(struct Employee), 1, filePtr);
            printf("Enter  Employee  Name:  ");  scanf("%s",
            emp.empName); printf("Enter Employee Salary:
            "); scanf("%f", &emp.empSalary);
            fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);
            fwrite(&emp, sizeof(struct Employee), 1, filePtr);
            printf("Employee details updated successfully.\n"); break;
        case 4:
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
```

```
} while (choice !=
4); fclose(filePtr);
return 0;
```

OUTPUT :



17. Illustrate the deadlock avoidance concept by simulating Banker's algorithm using C.

PROGRAM :

```
#include <stdio.h>


#define MAX_PROCESSES 5
#define MAX_RESOURCES 3 int
is_safe(); int available[MAX_RESOURCES] = {3, 3, 2}; // Available instances of
each resource
int maximum[MAX_PROCESSES][MAX_RESOURCES] = {{7, 5, 3},
{3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};
int allocation[MAX_PROCESSES][MAX_RESOURCES] = {{0, 1, 0},
{2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};
```

```
int request_resources(int process_num, int request[]) {
    // Check if request can be granted for (int i =
    0; i < MAX_RESOURCES; i++) {
            if (request[i] > available[i] || request[i] > maximum[process_num][i]
- allocation[process_num][i])
          return 0; // Request cannot be granted
    }

    // Try allocating resources temporarily for (int
    i = 0; i < MAX_RESOURCES; i++) {
    available[i] -= request[i];
    allocation[process_num][i] += request[i];
        // Update maximum and need matrix if request is granted
        maximum[process_num][i] -= request[i];
    }


    // Check if system is in safe state after allocation if
    (is_safe()) { return 1; // Request
        is granted
    } else {
        // Roll back changes if not safe for (int i = 0; i
        < MAX_RESOURCES; i++) { available[i]
        += request[i]; allocation[process_num][i] -=
        request[i]; maximum[process_num][i] +=
        request[i];
        }
        return 0; // Request is denied
    }
}
```

```
int is_safe() {
    int work[MAX_RESOURCES];
    int finish[MAX_PROCESSES] = {0};


    // Initialize work array
    for (int i = 0; i < MAX_RESOURCES; i++) { work[i]
        = available[i];
    }

    // Check if processes can finish int count = 0;
    while (count < MAX_PROCESSES) { int found
    = 0; for (int i = 0; i < MAX_PROCESSES; i++)
    { if
            (finish[i] == 0) { int
                j;
                for (j = 0; j < MAX_RESOURCES; j++) { if
                    (maximum[i][j] - allocation[i][j] > work[j]) break;
                }
                if (j == MAX_RESOURCES) {
                    // Process can finish, update work and mark as finished for (int k
                    = 0; k < MAX_RESOURCES; k++) {
                        work[k] += allocation[i][k];
                    } finish[i] =
                    1; found =
                    1; count++;
                }
            }
        }
        if (found == 0) { return 0; // No process can
        finish, not safe state }
    }
    return 1; // All processes can finish, safe state
}
```

```
int main() {
    int process_num, request[MAX_RESOURCES];
    printf("Enter process number (0 to 4): "); scanf("%d",
    &process_num);

    printf("Enter resource request (e.g., 0 1 0): "); for (int
    i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &request[i]);
    }

    if (request_resources(process_num, request)) {
        printf("Request granted.\n");
    } else { printf("Request denied. System is not in safe
        state.\n");
    }
  return 0;
```

OUTPUT :

```
C:\Users\itssk\OneDrive\Desk
Enter process number (0 to 4): 2
Enter resource request (e.g., 0 1 0): 1
0
1
Request denied. System is not in safe state.

--------------------------------
Process exited after 10.02 seconds with return v
Press any key to continue . . .
```

18. Construct a C program to simulate producer consumer problem using semaphores.

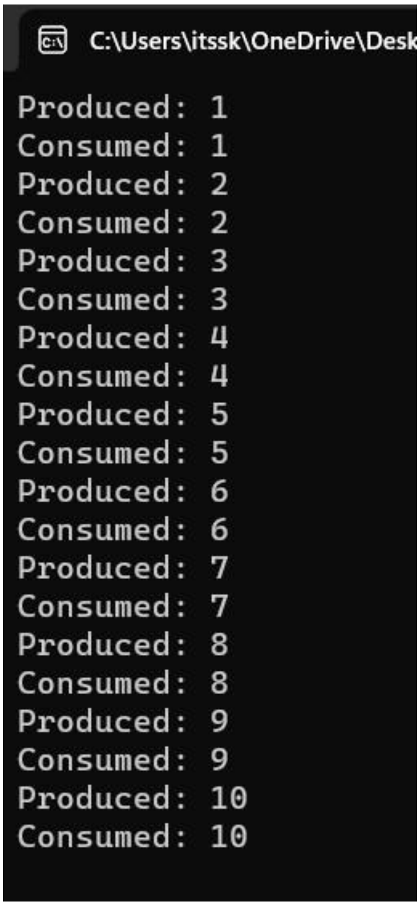
PROGRAM :

```
#include <stdio.h> #include
<pthread.h> #include
<semaphore.h>
#include<Windows.h>

#define BUFFER_SIZE 5
#define MAX_ITEMS 10 // Maximum number of items to be
produced/consumed


int buffer[BUFFER_SIZE];
sem_t empty, full;
int produced_items = 0, consumed_items = 0;


void*    producer(void*    arg)    {    while
   (produced_items    <    MAX_ITEMS)    {
   sem_wait(&empty);
      // Critical section: add item to buffer for
      (int i = 0; i < BUFFER_SIZE; ++i) { if
         (buffer[i]    ==    0)    {    buffer[i]    =
            produced_items          +          1;
            printf("Produced: %d\n", buffer[i]);
            produced_items++; break;
         } }
      sem_post(&full)
      ;
      Sleep(1); // Sleep for a while
   } return
   NULL;
}
```

```
void* consumer(void* arg) {
    while (consumed_items < MAX_ITEMS) {
        sem_wait(&full);
        // Critical section: remove item from buffer for
        (int i = 0; i < BUFFER_SIZE; ++i) { if (buffer[i]
        != 0) { printf("Consumed: %d\n", buffer[i]);
        buffer[i] = 0;
                consumed_items++;
                break;
            }
        }
        sem_post(&empty); Sleep(2); // Sleep
        for a while
    } return
    NULL;
}

int main() {
    pthread_t producer_thread, consumer_thread;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);


    // Create producer and consumer threads pthread_create(&producer_thread,
    NULL, producer, NULL); pthread_create(&consumer_thread, NULL,
    consumer, NULL);

    // Wait for threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread,
    NULL);
```

```
    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);


    return 0;
}
```

OUTPUT :

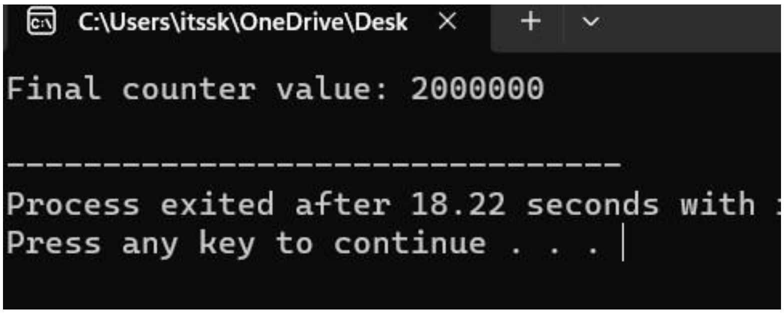


19. design a C program to implement process synchronization using mutex locks.

PROGRAM :

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

// Shared variables int
counter = 0;
pthread_mutex_t mutex;


// Function to be executed by threads void
*threadFunction(void *arg) {
    int i;
    for (i = 0; i < 1000000; ++i) { } return
    NULL;
}


int main() {pthread_mutex_init(&mutex, NULL);
   pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, threadFunction, NULL);
    pthread_create(&thread2, NULL, threadFunction, NULL);

    // Wait for the threads to finish pthread_join(thread1,
        NULL); pthread_join(thread2, NULL);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    // Print the final value of the counter printf("Final
    counter value: %d\n", counter);


    return 0;
}
```

OUTPUT :



20. Construct a C program to simulate Reader-Writer problem using semaphores

PROGRAM :

```
#include <stdio.h> #include
<pthread.h> #include
<semaphore.h>

sem_t mutex, writeBlock;
int data = 0, readersCount = 0;

void *reader(void *arg) { int
          i=0;
    while (i<10) {
        sem_wait(&mutex);
        readersCount++; if
        (readersCount == 1) {
            sem_wait(&writeBlock);
        }
        sem_post(&mutex);

        // Reading operation
        printf("Reader reads data: %d\n", data);
```

```
        sem_wait(&mutex);
        readersCount--; if
        (readersCount == 0) {
            sem_post(&writeBlock);
        }
        sem_post(&mutex);
        i++;
    }
}


void *writer(void *arg) { int
          i=0;
    while (i<10) {
        sem_wait(&writeBlock);

        //      Writing      operation      data++;
        printf("Writer writes data: %d\n", data);



        sem_post(&writeBlock); i++;
    }
}


int main() {
    pthread_t readers, writers; sem_init(&mutex,
    0, 1);
    sem_init(&writeBlock, 0, 1); pthread_create(&readers,
   NULL, reader, NULL); pthread_create(&writers, NULL,
   writer, NULL); pthread_join(readers, NULL);
   pthread_join(writers, NULL); sem_destroy(&mutex);
   sem_destroy(&writeBlock);
    return 0;
```

}

OUTPUT :

```
Writer writes data: 1
Reader reads data: 1
Writer writes data: 2
Reader reads data: 2
Writer writes data: 3
Reader reads data: 3
Writer writes data: 4
Reader reads data: 4
Writer writes data: 5
Reader reads data: 5
Writer writes data: 6
Reader reads data: 6
Writer writes data: 7
Reader reads data: 7
Writer writes data: 8
Reader reads data: 8
Writer writes data: 9
Reader reads data: 9
Writer writes data: 10
Reader reads data: 10

--------------------------------
Process exited after 12.44 seconds with
```

21. Develop a C program to implement worst fit algorithm of memory management.

PROGRAM:

```
#include <stdio.h>

#define MAX_MEMORY 1000 int

memory[MAX_MEMORY];

// Function to initialize memory void
initializeMemory() {
    for (int i = 0; i < MAX_MEMORY; i++) { memory[i] = -1; // -1
        indicates that the memory is unallocated
    }
}

// Function to display memory status void
displayMemory() {
    int i, j;
    int count = 0; printf("Memory
    Status:\n");

    for (i = 0; i < MAX_MEMORY; i++) {
        if (memory[i] == -1) {
            count++;
            j = i;
            while (memory[j] == -1 && j < MAX_MEMORY) { j++;
            }
            printf("Free memory block %d-%d\n", i, j - 1); i = j -
            1;
        }
    }

    if (count == 0) { printf("No free memory
        available.\n");
    }
}

// Function to allocate memory using worst-fit algorithm
void allocateMemory(int processId, int size) { int start = -1;
int blockSize = 0;

    for (int i = 0; i < MAX_MEMORY; i++) { if
        (memory[i] == -1) { if
            (blockSize == 0) {
```

```
            start = i; }
            blockSize++;
        } else { blockSize
            = 0;
        }

        if (blockSize >= size) {
            break;
        }
    }

    if (blockSize >= size) { for (int i = start; i
        < start + size; i++) { memory[i] =
        processId;
        }
        printf("Allocated memory block %d-%d to Process %d\n", start, start + size - 1,
processId);
    } else { printf("Memory allocation for Process %d failed (not enough
        contiguous
memory).\n", processId);
    }
}

// Function to deallocate memory void
deallocateMemory(int processId) { for (int i =
0; i < MAX_MEMORY; i++) { if
        (memory[i] == processId) {
            memory[i] = -1;
        }
    }
    printf("Memory released by Process %d\n", processId);
}

int main() {
    initializeMemory();
    displayMemory();

    allocateMemory(1, 200);
    displayMemory();

    allocateMemory(2, 300);
    displayMemory();

    deallocateMemory(1);
    displayMemory();
```

```
    allocateMemory(3, 400);
    displayMemory();

    return 0;
}
```

OUTPUT:

```
        Memory Management Scheme - Worst Fit
Enter the number of blocks:3
Enter the number of files:2

Enter the size of the blocks:-
Block 1:5
Block 2:2
Block 3:7
Enter the size of the files :-
File 1:1
File 2:4

File_no:        File_size :     Block_no:       Block_size:     Fragement
1               1               3               7               6
2               4               1               5               1_
```

22. Construct a C program to implement best fit algorithm of memory management.

PROGRAM:

```
#include <stdio.h>

#define MAX_MEMORY 1000 int

memory[MAX_MEMORY];

// Function to initialize memory void
initializeMemory() {
    for (int i = 0; i < MAX_MEMORY; i++) { memory[i] = -1; // -1
        indicates that the memory is unallocated
    }
}

// Function to display memory status void
displayMemory() {
    int i, j;
    int count = 0; printf("Memory
    Status:\n");

    for (i = 0; i < MAX_MEMORY; i++) {
        if (memory[i] == -1) {
            count++;
            j = i;
            while (memory[j] == -1 && j < MAX_MEMORY) { j++;
            }
            printf("Free memory block %d-%d\n", i, j - 1); i = j -
            1;
        }
    }

    if (count == 0) { printf("No free memory
        available.\n");
    }
}
```

```
// Function to allocate memory using best-fit algorithm
void allocateMemory(int processId, int size) { int start = -
1;
    int blockSize = MAX_MEMORY; int
    bestStart = -1;
    int bestSize = MAX_MEMORY;

    for (int i = 0; i < MAX_MEMORY; i++) { if
        (memory[i]    ==    -1)    {    if    (blockSize    ==
            MAX_MEMORY) { start = i;
            }
            blockSize++;
        } else { if (blockSize >= size && blockSize < bestSize) {
            bestSize
                = blockSize;
                bestStart = start;
            }
            blockSize = 0;
        }
    }

    if (bestSize >= size) { for (int i = bestStart; i <
        bestStart + size; i++) { memory[i] = processId;
        }
        printf("Allocated memory block %d-%d to Process %d\n", bestStart, bestStart +
size - 1, processId);
    } else { printf("Memory allocation for Process %d failed (not enough
        contiguous
memory).\n", processId);
    }
}

//    Function    to    deallocate    memory    void
deallocateMemory(int processId) { for (int i =
0; i < MAX_MEMORY; i++) { if
        (memory[i] == processId) {
            memory[i] = -1;
        }
    }
    printf("Memory released by Process %d\n", processId);
}

int main() {
    initializeMemory();
    displayMemory();
```

```
    allocateMemory(1, 200);
    displayMemory();

    allocateMemory(2, 300);
    displayMemory();

    deallocateMemory(1);
    displayMemory();

    allocateMemory(3, 400);
    displayMemory();


    return 0;
}
```

OUTPUT:

```
Memory Status:
Free memory block 0-999
Allocated memory block -1-198 to Process 1
Memory Status:
Free memory block 199-999
Allocated memory block -1-298 to Process 2
Memory Status:
Free memory block 299-999
Memory released by Process 1
Memory Status:
Free memory block 299-999
Allocated memory block -1-398 to Process 3
Memory Status:
Free memory block 399-999


--------------------------------
Process exited after 0.06954 seconds with return value 0
Press any key to continue . . .
```

23. Construct a C program to implement first fit algorithm of memory management.

PROGRAM:

```
#include <stdio.h>

#define MAX_MEMORY 1000 int

memory[MAX_MEMORY];

// Function to initialize memory void
initializeMemory() {
    for (int i = 0; i < MAX_MEMORY; i++) { memory[i] = -1; // -1
    indicates that the memory is unallocated }
}

// Function to display memory status void
displayMemory() {
    int i, j;
    int count = 0; printf("Memory
    Status:\n"); for (i = 0; i <
    MAX_MEMORY; i++) { if
    (memory[i] == -1) {
            count++;
            j = i;
            while (memory[j] == -1 && j < MAX_MEMORY) { j++;
            }
            printf("Free memory block %d-%d\n", i, j - 1); i = j -
            1;
        }
    }

    if (count == 0) { printf("No free memory
        available.\n");
    }
}

// Function to allocate memory using first-fit algorithm
void allocateMemory(int processId, int size) { int start = -
1; int blockSize = 0;

    for (int i = 0; i < MAX_MEMORY; i++) { if
        (memory[i] == -1) { if
            (blockSize == 0) {
            start = i; }
            blockSize++;
        } else {
```

```
            blockSize = 0;
        }

        if (blockSize >= size) {
            break;
        }
    }


    if (blockSize >= size) { for (int i = start; i
        < start + size; i++) { memory[i] =
        processId;
        }
        printf("Allocated memory block %d-%d to Process %d\n", start, start + size - 1,
processId);
    } else { printf("Memory allocation for Process %d failed (not enough
        contiguous
memory).\n", processId);
    }
}

// Function to deallocate memory void
deallocateMemory(int processId) { for (int i =
0; i < MAX_MEMORY; i++) { if
        (memory[i] == processId) {
            memory[i] = -1;
        }
    }
    printf("Memory released by Process %d\n", processId);
}

int main() {
    initializeMemory();
    displayMemory();

    allocateMemory(1, 200);
    displayMemory();

    allocateMemory(2, 300);
    displayMemory();

    deallocateMemory(1);
    displayMemory();

    allocateMemory(3, 400);
    displayMemory();


    return 0;
```

}

OUTPUT:

```
Memory Status:
Free memory block 0-999
Allocated memory block 0-199 to Process 1
Memory Status:
Free memory block 200-999
Allocated memory block 200-499 to Process 2
Memory Status:
Free memory block 500-999
Memory released by Process 1
Memory Status:
Free memory block 0-199
Free memory block 500-999
Allocated memory block 500-899 to Process 3
Memory Status:
Free memory block 0-199
Free memory block 900-999

--------------------------------
Process exited after 0.01792 seconds with return value 0
Press any key to continue . . .
```

24. Design a C program to demonstrate UNIX system calls for file management.

PROGRAM:

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

int main() {
    int fd;
    char buffer[100];

    // Creating a new file
    fd = creat("sample.txt", S_IRWXU); if (fd
    == -1) {
        perror("create");
        exit(1);
    } else { printf("File 'sample.txt' created successfully.\n");
        close(fd);
    }

    // Opening an existing file for writing
    fd = open("sample.txt", O_WRONLY | O_APPEND); if (fd ==
    -1) {
        perror("open");
        exit(1);
    } else { printf("File 'sample.txt' opened for
        writing.\n");
    }

    // Writing data to the file write(fd,
    "Hello, World!\n", 14);
    printf("Data written to 'sample.txt'.\n");
    close(fd);

    // Opening the file for reading
    fd = open("sample.txt", O_RDONLY); if (fd
    == -1) {
        perror("open");
        exit(1);
    } else { printf("File 'sample.txt' opened for
    reading.\n"); }

    // Reading data from the file
    int bytesRead = read(fd, buffer, sizeof(buffer)); if
```

```
    (bytesRead == -1) {
        perror("read");
        exit(1);
    } else { printf("Data read from 'sample.txt':\n"); write(STDOUT_FILENO,
        buffer, bytesRead);
    }
    close(fd);

    // Deleting the file if
    (remove("sample.txt") == -1) {
    perror("remove");
        exit(1);
    } else { printf("File 'sample.txt'
    deleted.\n"); }
    return 0;
}
```

OUTPUT:

```
File 'sample.txt' created successfully.
File 'sample.txt' opened for writing.
Data written to 'sample.txt'.
File 'sample.txt' opened for reading.
Data read from 'sample.txt':
Hello, World!
File 'sample.txt' deleted.

--------------------------------
Process exited after 0.02066 seconds with return value 0
Press any key to continue . . .
```

25) Construct a C program to implement the I/O system calls of UNIX (fcntl, seek, stat, opendir, readdir)

PROGRAM:

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno; int
main()
{

        int fd = open("foo.txt", O_RDONLY | O_CREAT);
        printf("fd = %d\n", fd);
        if (fd ==-1)
        { printf("Error Number % d\n", errno);
                perror("Program");
        } return
        0;
}
```

OUTPUT:

```
fd = 3

--------------------------------
Process exited after 0.08362 seconds with return value 0
Press any key to continue . . .
```

26) Construct a C program to implement the file management operations.

PROGRAM:

```
#include        <stdio.h>
 #include <stdlib.h> int main()
 {
       FILE *file; file =
     fopen("example.txt", "w"); if
     (file == NULL) { printf("Error opening the file for
     writing.\n"); return 1; }
     fprintf(file, "Hello, World!\n");
     fprintf(file, "This is a C file management example.\n");
     fclose(file);
     file = fopen("example.txt", "r"); if
     (file == NULL) { printf("Error opening the file for
     reading.\n"); return 1; } char buffer[100];
     while (fgets(buffer, sizeof(buffer), file) != NULL) {
         printf("%s", buffer);
     }
     fclose(file);

     return 0;
 }
```

OUTPUT:

```
Hello, World!
This is a C file management example.

--------------------------------
Process exited after 0.1135 seconds with return value 0
Press any key to continue . . .
```

27) Develop a C program for simulating the function of ls UNIX Command.

PROGRAM:

```
#include<stdio.h>
#include<dirent.h>
   int main()
       { char fn[10], pat[10], temp[200];
     FILE
     *fp;
     printf("\n Enter file name : ");
     scanf("%s", fn); printf("Enter the
     pattern: "); scanf("%s", pat); fp =
     fopen(fn, "r"); while (!feof(fp)) {
     fgets(temp, sizeof(fp), fp); if
     (strcmp(temp, pat)) printf("%s",
     temp);
     }
     fclose(fp);
     return 1;

  }
```

OUTPUT:

```
This is a sample line.
Hello, World!
Sample pattern in this line.
Another sample line.
```

28) Write a C program for simulation of GREP UNIX command.

PROGRAM :

```
#include        <stdio.h>
#include       <stdlib.h>
#include <string.h>
#define MAX_LINE_LENGTH 1024
void searchFile(const char *pattern, const char *filename)
{
FILE *file = fopen(filename, "r"); if
    (file == NULL) { perror("Error
    opening file"); exit(1);
    }
    char line[MAX_LINE_LENGTH]; while
    (fgets(line,  sizeof(line),  file))  {  if
        (strstr(line,  pattern)  !=  NULL)  {
        printf("%s", line);
        } }
    fclose(file)
    ;
}

int main(int argc, char *argv[]) { if
    (argc != 3) { fprintf(stderr, "Usage: %s <pattern> <filename>\n",
        argv[0]); return 1;
    }
    const char *pattern = argv[1]; const
    char     *filename     =     argv[2];
    searchFile(pattern,         filename);
    return 0;
}
```

OUTPUT:

```
Usage: D:\anshul\c program easy level\2).exe <pattern> <filename>

--------------------------------
Process exited after 0.06583 seconds with return value 1
Press any key to continue . . .
```

29) Write a C program to simulate the solution of Classical Process Synchronization Problem

PROGRAM:

```
#include        <stdio.h>
 #include <stdlib.h> int mutex
 = 1;
 int full = 0; int empty
 = 10, 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 full!");
```

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

        case 3:
            exit(0);
            break;
        }
    }
}
```

OUTPUT:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer producesitem 1
Enter your choice:1

Producer producesitem 2
Enter your choice:1

Producer producesitem 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:3

--------------------------------
Process exited after 6.797 seconds with return value 0
Press any key to continue . . .
```

30. **Write C programs to demonstrate the following thread related concepts.**

PROGRAM:

```
#include <pthread.h>
#include     <stdio.h>
#include    <stdlib.h>
void* func(void* arg)
{ pthread_detach(pthread_self()); printf("Inside the
        thread\n"); pthread_exit(NULL);
} void
fun()
{ pthread_t ptid;
        pthread_create(&ptid, NULL, &func, NULL);
        printf("This line may be printed"
                " before thread terminates\n");
        if(pthread_equal(ptid, pthread_self()))
        { printf("Threads are equal\n");
        }

        else printf("Threads are not equal\n");
        pthread_join(ptid, NULL);
        printf("This line will be printed" "
        after thread ends\n");
        pthread_exit(NULL);
}

int main()
{ fun(); return
        0;
}
```

OUTPUT:

```
This line may be printed before thread terminates
Inside the thread
Threads are not equal
This line will be printed after thread ends
```

31. Construct a C program to simulate the First in First Out paging technique of memory management.

PROGRAM:

```
#include <stdio.h>

#define MAX_FRAMES 3 // Maximum number of frames in memory

void printFrames(int frames[], int n) { for
    (int i = 0; i < n; i++) {
        if (frames[i] == -1) {
        printf(" - ");
        } else { printf(" %d ",
            frames[i]);
        } }
    printf("\n")
    ;
}


int main() { int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3,
    2}; int n = sizeof(referenceString) / sizeof(referenceString[0]);
    int frames[MAX_FRAMES];
    int framePointer = 0; // Points to the current frame to be replaced

    for (int i = 0; i < MAX_FRAMES; i++)
    { frames[i] = -1; // Initialize all frames to -1 (indicating empty)
    }

    printf("Reference String: "); for
    (int i = 0; i < n; i++) { printf("%d ",
        referenceString[i]);
    }printf("\n\n");
    printf("Page Replacement
    Order:\n"); for (int i = 0; i < n; i++) {
```

```
    int page = referenceString[i]; int
    pageFound = 0;
        // Check if the page is already in memory for (int j
        = 0; j < MAX_FRAMES; j++) { if
            (frames[j] == page) {
                pageFound = 1; break;
            }
        }


        if (!pageFound) {
            printf("Page %d -> ", page); frames[framePointer] =
            page; framePointer = (framePointer + 1) %
            MAX_FRAMES; printFrames(frames,
            MAX_FRAMES);
        }
    }


    return 0;
}
```

OUTPUT:

```
Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 ->   7  -  -
Page 0 ->   7  0  -
Page 1 ->   7  0  1
Page 2 ->   2  0  1
Page 3 ->   2  3  1
Page 0 ->   2  3  0
Page 4 ->   4  3  0
Page 2 ->   4  2  0
Page 3 ->   4  2  3
Page 0 ->   0  2  3

--------------------------------
Process exited after 0.0623 seconds with return value 0
Press any key to continue . . .
```

32. Construct a C program to simulate the Least Recently Used paging technique of memory management.

PROGRAM:

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

#define MAX_FRAMES 3

void printFrames(int frames[], int n) { for
    (int i = 0; i < n; i++) {
        if (frames[i] == -1) {
        printf(" - ");
        } else { printf(" %d ",
            frames[i]);
        }
    }
    printf("\n");
}

int main() {
    int frames[MAX_FRAMES]; int usageHistory[MAX_FRAMES]; // To store
    the usage history of pages for (int i = 0; i < MAX_FRAMES; i++) { frames[i]
    = -1; // Initialize frames to -1 (empty) usageHistory[i] = 0; // Initialize usage
    history
    }


    int pageFaults = 0; int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4,
    2, 3, 0, 3, 2}; int n = sizeof(referenceString) /
    sizeof(referenceString[0]);

    printf("Reference String: "); for
    (int i = 0; i < n; i++) { printf("%d
        ", referenceString[i]);
    } printf("\n\n"); printf("Page

    Replacement Order:\n"); for


    (int i = 0; i < n; i++) { int page =
        referenceString[i]; int
        pageFound = 0;

        // Check if the page is already in memory (a page hit) for (int j
```

```
= 0; j < MAX_FRAMES; j++) {
    if (frames[j] == page) {
    pageFound = 1;
        // Update the usage history by incrementing other pages for (int k
        = 0; k < MAX_FRAMES; k++) {
            if (k != j) {
                usageHistory[k]++;
            }
        }
    usageHistory[j] = 0; // Reset the usage counter for the used page break;
    }
}

if (!pageFound) {
    printf("Page %d -> ", page);

    // Find the page with the maximum usage counter (least recently
used)  int lruPage = 0;
    for (int j = 1; j < MAX_FRAMES; j++) { if
        (usageHistory[j] > usageHistory[lruPage]) { lruPage
        = j;
        }
    }

    int replacedPage = frames[lruPage];
    frames[lruPage] = page;
    usageHistory[lruPage] = 0;

    if (replacedPage != -1) { printf("Replace %d with %d:
        ", replacedPage, page);
    } else { printf("Load into an empty
        frame: ");
    }

    printFrames(frames, MAX_FRAMES);
    pageFaults++;
}
}

printf("\nTotal Page Faults: %d\n", pageFaults);

return 0;
```

}

OUTPUT:

```
Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 -> Load into an empty frame:  7  -  -
Page 0 -> Replace 7 with 0:  0  -  -
Page 1 -> Replace 0 with 1:  1  -  -
Page 2 -> Replace 1 with 2:  2  -  -
Page 0 -> Replace 2 with 0:  0  -  -
Page 3 -> Replace 0 with 3:  3  -  -
Page 0 -> Replace 3 with 0:  0  -  -
Page 4 -> Replace 0 with 4:  4  -  -
Page 2 -> Replace 4 with 2:  2  -  -
Page 3 -> Replace 2 with 3:  3  -  -
Page 0 -> Replace 3 with 0:  0  -  -
Page 3 -> Replace 0 with 3:  3  -  -
Page 2 -> Replace 3 with 2:  2  -  -

Total Page Faults: 13

--------------------------------
Process exited after 0.05045 seconds with return value 0
Press any key to continue . . .
```

33. Construct a C program to simulate the optimal paging technique of memory management

PROGRAM:

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

#define MAX_FRAMES 3

void printFrames(int frames[], int n) { for
    (int i = 0; i < n; i++) {
        if (frames[i] == -1) {
        printf(" - ");
        } else { printf(" %d ",
            frames[i]);
        }
    }
    printf("\n");
}


int main() {
    int frames[MAX_FRAMES];
    for (int i = 0; i < MAX_FRAMES; i++) { frames[i] = -1;
        // Initialize frames to -1 (empty)
    }

    int pageFaults = 0; int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4,
    2, 3, 0, 3, 2}; int n = sizeof(referenceString) /
    sizeof(referenceString[0]);

    printf("Reference String: "); for
    (int i = 0; i < n; i++) { printf("%d ",
        referenceString[i]);
    } printf("\n\n"); printf("Page

    Replacement Order:\n"); for


    (int i = 0; i < n; i++) { int page
        = referenceString[i];
        int pageFound = 0;

        // Check if the page is already in memory (a page hit) for (int j
```

```
        = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
                pageFound = 1; break;
            }
        }

        if (!pageFound) { printf("Page
            %d -> ", page);

            int optimalPage = -1; int
            farthestDistance = 0;

            for (int j = 0; j < MAX_FRAMES; j++) { int
                futureDistance = 0; for (int k = i + 1; k < n;
                k++) { if (referenceString[k] == frames[j]) {
                break;
                    }
                    futureDistance++;
                }

                if (futureDistance > farthestDistance) {
                    farthestDistance = futureDistance;
                    optimalPage = j;
                }
            }

            frames[optimalPage] = page;

            printFrames(frames, MAX_FRAMES);
            pageFaults++;
        }
    }

    printf("\nTotal Page Faults: %d\n", pageFaults);

    return 0;
}
```

OUTPUT

```
Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 ->  7  -  -
Page 0 ->  0  -  -
Page 1 ->  0  1  -
Page 2 ->  0  2  -
Page 3 ->  0  2  3
Page 4 ->  4  2  3
Page 0 ->  0  2  3

Total Page Faults: 7

--------------------------------
Process exited after 0.05286 seconds with return value 0
Press any key to continue . . .
```

34. Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a record struct

Record { int

    recordNumber;

    char data[256]; // Adjust the size as needed for your records

};


int  main()  {  FILE  *file;

    struct   Record   record;

    int recordNumber;

    // Open or create a file in write mode (for writing records) file

    = fopen("sequential_file.txt", "w"); if (file == NULL) {

    printf("Error opening the file.\n");

        return 1;

    }
```

```
// Write records sequentially to the file
printf("Enter records (Enter '0' as record number to exit):\n"); while (1)
{ printf("Record Number: "); scanf("%d",
    &record.recordNumber); if
    (record.recordNumber == 0) {
        break;
    }

    // Input data for the record
    printf("Data: ");
    scanf(" %[^\n]", record.data);


    // Write the record to the file
    fwrite(&record, sizeof(struct Record), 1, file);
}

fclose(file);

// Reopen the file in read mode (for reading records) file
= fopen("sequential_file.txt", "r"); if (file == NULL) {
printf("Error opening the file.\n"); return 1;
}

// Read a specific record from the file while
(1) { printf("Enter the record number to read (0 to exit):
    "); scanf("%d", &recordNumber); if (recordNumber
    == 0) {
        break;
    }

    // Read and display records up to the requested record while
```

```
        (fread(&record, sizeof(struct Record), 1, file)) {
            printf("Record Number: %d\n", record.recordNumber);
            printf("Data: %s\n", record.data); if
            (record.recordNumber == recordNumber) { break; }
        }


        rewind(file); // Reset the file pointer to the beginning of the file
    }


    fclose(file);
    return 0;
}
```

OUTPUT:

```
Enter records (Enter '0' as record number to exit):
Record Number: 389
Data: JASWANTH
Record Number: 0
Enter the record number to read (0 to exit): 389
Record Number: 389
Data: JASWANTH
Enter the record number to read (0 to exit): 0

--------------------------------
Process exited after 29.88 seconds with return value 0
Press any key to continue . . .
```

35. Consider a file system that brings all the file pointers together into an index block. The with entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.

PROGRAM:

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


// Structure to represent a block
struct Block {int blockNumber;
    char data[256]; // Adjust the size as needed for your blocks
};


int main() { FILE *file;
    struct   Block   block;
    int blockNumber;

    // Create an index block that contains pointers to data blocks int
    indexBlock[100] = {0}; // Adjust the size as needed

    // Open or create a file in write mode (for writing blocks) file =
    fopen("indexed_file.txt", "w"); if
    (file == NULL) { printf("Error
    opening the file.\n"); return 1;
    }


    // Write blocks and update the index block
    printf("Enter blocks (Enter '0' as block number to exit):\n"); while (1)
    { printf("Block Number: "); scanf("%d",
        &block.blockNumber); if
        (block.blockNumber == 0) {
            break;
        }
```

```
    // Input data for the block
    printf("Data: ");
    scanf(" %[^\n]", block.data);

    // Write the block to the file
    fwrite(&block, sizeof(struct Block), 1, file);


    // Update the index block with the pointer to the data block indexBlock[block.blockNumber] =
    ftell(file) - sizeof(struct Block);
}


fclose(file);

// Reopen the file in read mode (for reading blocks) file
= fopen("indexed_file.txt", "r"); if (file == NULL) {
printf("Error opening the file.\n"); return 1;
}

// Read a specific block from the file while
(1) { printf("Enter the block number to read (0 to exit):
    "); scanf("%d", &blockNumber); if (blockNumber
    == 0) {
        break;
    }

    if (indexBlock[blockNumber] == 0) {
        printf("Block not found.\n");
    } else {
        // Seek to the data block using the index block fseek(file,
        indexBlock[blockNumber], SEEK_SET); fread(&block,
        sizeof(struct Block), 1, file);
```

```
            printf("Block Number: %d\n", block.blockNumber);
            printf("Data: %s\n", block.data);
        }
    }

    fclose(file);
    return 0;
}
```

OUTPUT:

```
Enter blocks (Enter '0' as block number to exit):
Block Number: 39
Data: JSAWNTH
Block Number: 43
Data: SAI
Block Number: 12
Data: FRIEND
Block Number: 0
Enter the block number to read (0 to exit): 12
Block Number: 12
Data: FRIEND
Enter the block number to read (0 to exit): 0

--------------------------------
Process exited after 34.15 seconds with return value 0
Press any key to continue . . .
```

36. **With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.**

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
// Structure to represent a block struct Block { char data[256]; //
Adjust the size as needed for your blocks struct Block* next;
};

int main() { struct Block* firstBlock = NULL; // Pointer to the first block in the
    linked list struct Block* lastBlock = NULL; // Pointer to the last block in the
    linked list

    int blockCount = 0; // Count of blocks in the linked list

    int blockNumber;
    char data[256]; char
    choice;

    printf("Linked Allocation Simulation\n");

    while (1) { printf("Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit:
        "); scanf(" %c", &choice);

        if (choice == 'Q' || choice == 'q') { break;
        }

        if (choice == 'W' || choice == 'w') {
            printf("Enter data for the block: "); scanf("
            %[^\n]", data);

            // Create a new block
```

```
struct Block* newBlock = (struct Block*)malloc(sizeof(struct Block)); for (int i =
0; i < 256; i++) { newBlock-
    >data[i] = data[i];
}
newBlock->next = NULL;

if (blockCount == 0) {
    // This is the first block firstBlock
     =          newBlock; lastBlock =
    newBlock;
} else {
    // Link the new block to the last block lastBlock-
    >next = newBlock; lastBlock = newBlock;
}


blockCount++;
} else if (choice == 'R' || choice == 'r') { printf("Enter the block
number to read (1-%d): ", blockCount); scanf("%d",
&blockNumber);

if (blockNumber < 1 || blockNumber > blockCount) { printf("Invalid
    block number. The valid range is 1-%d.\n",
blockCount);
} else { struct Block* currentBlock =
    firstBlock; for
    (int i = 1; i < blockNumber; i++) {
        currentBlock = currentBlock->next;
    }


        printf("Block %d Data: %s\n", blockNumber, currentBlock->data);
}
```

```
        }
    }


    // Free the allocated memory for blocks before exiting
    struct Block* currentBlock = firstBlock; while
    (currentBlock != NULL) { struct Block* nextBlock =
    currentBlock->next; free(currentBlock); currentBlock =
    nextBlock;
    }


    return 0;
}
```

OUTPUT:

```
Linked Allocation Simulation
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: W
Enter data for the block: SAI IS WORST
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: W
Enter data for the block: JASWANTH IS GOOD
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: R
Enter the block number to read (1-2): 2
Block 2 Data: JASWANTH IS GOOD
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: R
Enter the block number to read (1-2): 1
Block 1 Data: SAI IS WORST
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: Q

--------------------------------
Process exited after 46.7 seconds with return value 0
Press any key to continue . . .
```

37.Construct a C program to simulate the First Come First Served disk scheduling algorithm.

1. Start at the current position of the disk head.
2. For each disk request in the queue:

- Move the disk head to the requested track.
- Calculate the seek time as the absolute difference between the new position of the disk head and the previous position.
- Add the seek time to the total seek time.
- Update the previous position of the disk head to the current position.

3. Repeat step 2 for all disk requests in the queue.
4. After serving all the requests, calculate and display the total seek time.
5. Calculate and display the average seek time, which is the total seek time divided by the number of requests.

PROGRAM:-

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

int main() {
    int n, head, seek_time = 0;


    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    int request_queue[n];


    printf("Enter the disk request queue:\n"); for
    (int i = 0; i < n; i++) { scanf("%d",
    &request_queue[i]);
    }
```

```
    printf("Enter the initial position of the disk head: ");
    scanf("%d", &head);

    // FCFS Scheduling
    printf("\nFCFS Disk Scheduling:\n"); printf("Head
    Movement Sequence: %d", head); for (int i = 0; i < n;
    i++) { seek_time += abs(head - request_queue[i]);
    head = request_queue[i]; printf(" -> %d", head);
    }

    printf("\nTotal Seek Time: %d\n", seek_time); printf("Average Seek Time:
    %.2f\n", (float) seek_time / n);

    return 0;
}
```

OUTPUT:-

```
Enter the number of disk requests: 3
Enter the disk request queue:
222
22
123
Enter the initial position of the disk head: 1

FCFS Disk Scheduling:
Head Movement Sequence: 1 -> 222 -> 22 -> 123
Total Seek Time: 522
Average Seek Time: 174.00
```

38. Design a C program to simulate SCAN disk scheduling algorithm.

PROGRAM:

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

int main() {
    int n, head, seek_time = 0;


    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    int request_queue[n];


    printf("Enter the disk request queue:\n");
    for (int i = 0; i < n; i++) { scanf("%d",
    &request_queue[i]);
    }

    printf("Enter the initial position of the disk head: ");
    scanf("%d", &head);

    // Sort the request queue to simplify SCAN algorithm for (int i
    = 0; i < n - 1; i++) { for (int j =
        i + 1; j < n; j++) {
            if (request_queue[i] > request_queue[j]) { int
                temp = request_queue[i]; request_queue[i] =
                request_queue[j]; request_queue[j] = temp;
            }
        }
    }
```

```
    // SCAN (Elevator) Scheduling
    printf("\nSCAN (Elevator) Disk Scheduling:\n"); int start
    = 0; int end =
    n - 1;
    int current_direction = 1; // 1 for moving right, -1 for moving left

    while (start <= end) {
        if (current_direction == 1) { for
            (int i = start; i <= end; i++) {
                if (request_queue[i] >= head) {
                    seek_time += abs(head - request_queue[i]); head
                    = request_queue[i]; start = i + 1; break;
                } }
            current_direction = -1; // Change direction
        } else { for (int i = end; i >= start;
            i--) {
                if (request_queue[i] <= head) {
                    seek_time += abs(head - request_queue[i]); head
                    = request_queue[i]; end = i - 1;
                    break;
                } }
            current_direction = 1; // Change direction
        }
    }

    printf("Total Seek Time: %d\n", seek_time); printf("Average Seek Time:
    %.2f\n", (float)seek_time / n);

    return 0;
}
```

Output:-

```
Enter the number of disk requests: 3
Enter the disk request queue:
12
34
45
Enter the initial position of the disk head: 45

SCAN (Elevator) Disk Scheduling:
Total Seek Time: 0
Average Seek Time: 0.00
```

39. Develop a C program to simulate C-SCAN disk scheduling algorithm.

PROGRAM:-

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

int main() { int n, head,
    seek_time = 0;

    printf("Enter the number of disk requests: ");
    scanf("%d", &n);
int request_queue[n];

    printf("Enter the disk request queue:\n"); for
    (int i = 0; i < n; i++) { scanf("%d",
    &request_queue[i]);
    }

    printf("Enter the initial position of the disk head: ");
    scanf("%d", &head);

    // Sort the request queue for simplicity for
    (int i = 0; i < n - 1; i++) { for (int j = i + 1; j < n; j++)
        { if (request_queue[i] > request_queue[j]) { int
        temp = request_queue[i]; request_queue[i] =
        request_queue[j]; request_queue[j] = temp;
            }
        }
    }

    // C-SCAN Scheduling
    printf("\nC-SCAN Disk Scheduling:\n"); int
    start = 0;
    int end = n - 1;

    while (start <= end) { for (int i =
        start; i <= end; i++) { if
            (request_queue[i] >= head) { seek_time += abs(head
                - request_queue[i]); head = request_queue[i]; start
                = i + 1;
            }
        }
        // Move the head to the end in the current
        direction seek_time += abs(head - 0); head = 0;
```

```
        // Change direction to the opposite side seek_time +=
        abs(head - request_queue[end]); head =
        request_queue[end];
        end = n - 2; // Exclude the last request, as it has already been served

    }

    printf("Total Seek Time: %d\n", seek_time); printf("Average Seek Time:
    %.2f\n", (float)seek_time / n);

    return 0;
}
```

OUTPUT:-

```
Enter the number of disk requests: 3
Enter the disk request queue:
12
13
14
Enter the initial position of the disk head: 5

C-SCAN Disk Scheduling:
Total Seek Time: 37
Average Seek Time: 12.33
```

40. Illustrate the various File Access Permission and different types users in Linux.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main() { char filename[] =
    "file.txt";
    int new_permissions = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH; // rw-
rwr--

    if (chmod(filename, new_permissions) == 0) { printf("File
        permissions changed successfully.\n");
    }        else        {
        perror("chmod")
        ; return 1;
    }

    return 0;
}
```

OUTPUT:

1. Compile the C program (assuming it's saved in a file named `change_permissions.c`):

bash

```bash
gcc -o change_permissions change_permissions.c
```

1. Run the program:

bash

```bash
./change_permissions
```

Output:

If the program executes successfully, it should display the following output:

arduino

```arduino
File permissions changed successfully.
```