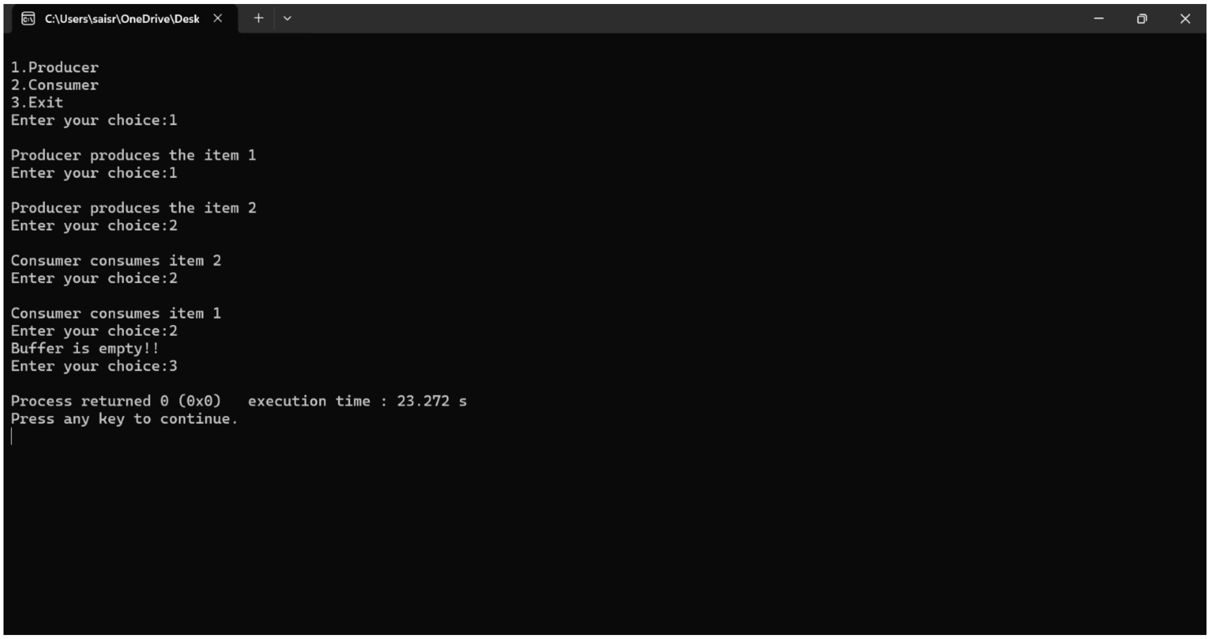
4.Write a C program to simulate producer-consumer problem using semaphore

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

#define BUFFER_SIZE 5
#define MAX_ITEMS 20

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int produced_count = 0;
int consumed_count = 0;
sem_t mutex;
sem_t full;
sem_t empty;

void* producer(void* arg) {
        int item = 1;
        while (produced_count < MAX_ITEMS) {
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        printf("Produced: %d\n", item);
        item++;
        in = (in + 1) % BUFFER_SIZE;
        produced_count++;
        sem_post(&mutex);
        sem_post(&full);
        }
        pthread_exit(NULL);
}

void* consumer(void* arg) {
        while (consumed_count < MAX_ITEMS) {
        sem_wait(&full);
        sem_wait(&mutex);
        int item = buffer[out];
        printf("Consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        consumed_count++;
        sem_post(&mutex);
        sem_post(&empty);
        }
        pthread_exit(NULL);
}

int main() {
        pthread_t producerThread, consumerThread;
        sem_init(&mutex, 0, 1);
        sem_init(&full, 0, 0);
        sem_init(&empty, 0, BUFFER_SIZE);

        pthread_create(&producerThread, NULL, producer, NULL);
        pthread_create(&consumerThread, NULL, consumer, NULL);

        pthread_join(producerThread, NULL);
        pthread_join(consumerThread, NULL);

        sem_destroy(&mutex);
        sem_destroy(&full);
        sem_destroy(&empty);

        return 0;
}
```



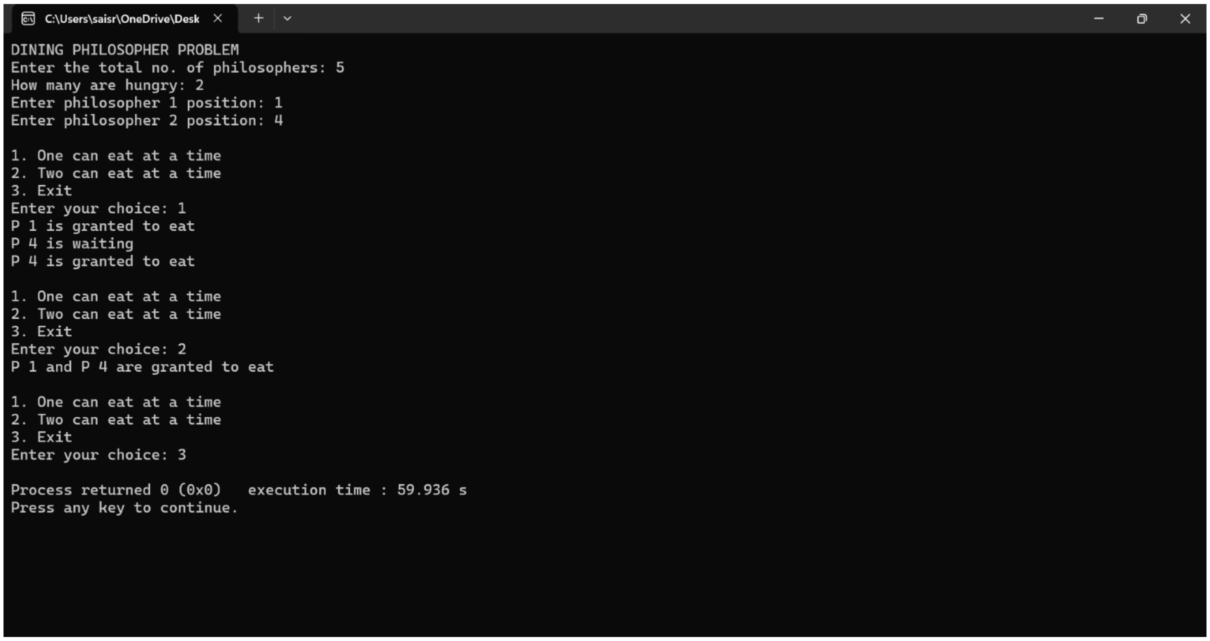
```
1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
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

Process returned 0 (0x0)   execution time : 23.272 s
Press any key to continue.
```

6..

Write a C program to simulate the concept of Dining-Philosophers problem.

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_PHILOSOPHERS 5

void allow_one_to_eat(int hungry[], int n) {

        int isWaiting[MAX_PHILOSOPHERS];
```

```
    for (int i = 0; i < n; i++) {
    isWaiting[i] = 1;
    }
    for (int i = 0; i < n; i++) {
    printf("P %d is granted to eat\n", hungry[i]);
    isWaiting[hungry[i]] = 0;
    for (int j = 0; j < n; j++) {
    if (isWaiting[hungry[j]]) {
        printf("P %d is waiting\n", hungry[j]);
    }
    }
    for (int k = 0; k < n; k++) {
    isWaiting[k] = 1;
    }
    isWaiting[hungry[i]] = 0;
    }
}
void allow_two_to_eat(int hungry[], int n) {
    if (n < 2 || n > MAX_PHILOSOPHERS) {
    printf("Invalid number of philosophers.\n");
    return;
    }
    for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
    printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);
    for (int k = 0; k < n; k++) {
        if (k != i && k != j) {
        printf("P %d is waiting\n", hungry[k]);
        }
    }
    }
    }
}
int main() {
    int total_philosophers, hungry_count;
    int hungry_positions[MAX_PHILOSOPHERS];
    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers: ");
    scanf("%d", &total_philosophers);
    if (total_philosophers > MAX_PHILOSOPHERS || total_philosophers < 2) {
    printf("Invalid number of philosophers.\n");
    return 1;
    }
    printf("How many are hungry: ");
    scanf("%d", &hungry_count);
    if (hungry_count < 1 || hungry_count > total_philosophers) {
    printf("Invalid number of hungry philosophers.\n");
    return 1;
    }
    for (int i = 0; i < hungry_count; i++) {
    printf("Enter philosopher %d position: ", i + 1);
    scanf("%d", &hungry_positions[i]);
    if (hungry_positions[i] < 0 || hungry_positions[i] >= total_philosophers) {
    printf("Invalid philosopher position.\n");
    return 1;
    }
    }
    int choice;
    while (1) {
    printf("\n1. One can eat at a time\n");
    printf("2. Two can eat at a time\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
    case 1:
        allow_one_to_eat(hungry_positions, hungry_count);
        break;
    case 2:
        allow_two_to_eat(hungry_positions, hungry_count);
        break;
    case 3:
        exit(0);
    default:
        printf("Invalid choice\n");
    }
    }
    return 0;
}
```

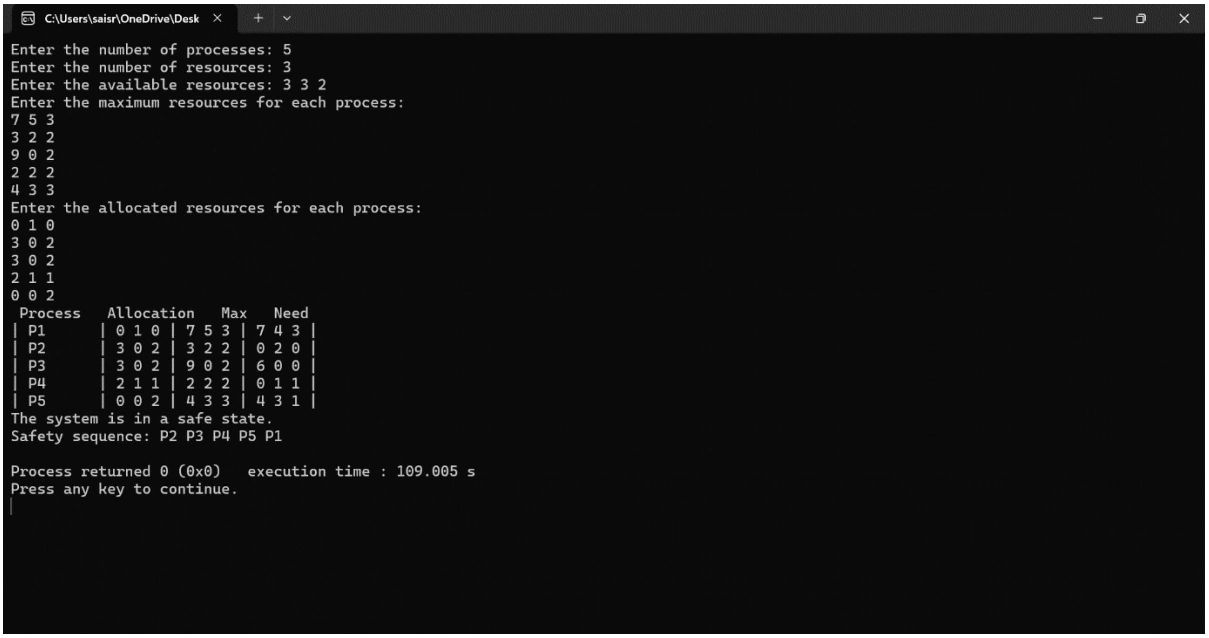


```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 2
Enter philosopher 1 position: 1
Enter philosopher 2 position: 4

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1
P 1 is granted to eat
P 4 is waiting
P 4 is granted to eat

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 2
P 1 and P 4 are granted to eat

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 3

Process returned 0 (0x0)   execution time : 59.936 s
Press any key to continue.
```

7. Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

```
#include <stdio.h>

int main() {

        int n, m;

        printf("Enter the number of processes: ");

        scanf("%d", &n);

        printf("Enter the number of resources: ");

        scanf("%d", &m);

        int available[m];

        printf("Enter the available resources: ");

        for (int i = 0; i < m; i++) {

        scanf("%d", &available[i]);

        }

        int maximum[n][m];

        printf("Enter the maximum resources for each process:\n");

        for (int i = 0; i < n; i++) {

        for (int j = 0; j < m; j++) {

        scanf("%d", &maximum[i][j]);
}
}
int allocation[n][m];
printf("Enter the allocated resources for each process:\n");
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
scanf("%d", &allocation[i][j]);
}
}
int need[n][m];
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
need[i][j] = maximum[i][j] - allocation[i][j];
}
}
printf(" Process   Allocation   Max   Need      \n");
for (int i = 0; i < n; i++) {
printf("| P%d   | ", i + 1);
for (int j = 0; j < m; j++) {
printf("%d ", allocation[i][j]);
}
printf("| ");
for (int j = 0; j < m; j++) {
printf("%d ", maximum[i][j]);
}
printf("| ");
for (int j = 0; j < m; j++) {
printf("%d ", need[i][j]);
}
```

```
printf("|\n");
}
int work[m];
for (int i = 0; i < m; i++) {
work[i] = available[i];
}
int finish[n];
for (int i = 0; i < n; i++) {
finish[i] = 0;
}
int safeSequence[n];
int count = 0;
int safe = 1;
while (count < n) {
int found = 0;
for (int i = 0; i < n; i++) {
if (finish[i] == 0) {
        int j;
        for (j = 0; j < m; j++) {
        if (need[i][j] > work[j]) {
        break;
        }
        }
        if (j == m) {
        for (j = 0; j < m; j++) {
        work[j] += allocation[i][j];
        }
        finish[i] = 1;
        safeSequence[count++] = i;
        found = 1;
        }
    }
    }
    if (!found) {
    safe = 0;
    break;
    }
    }
    if (safe) {
    printf("The system is in a safe state.\n");
    printf("Safety sequence: ");
    for (int i = 0; i < n; i++) {
    printf("P%d ", safeSequence[i] + 1);
    }
    printf("\n");
    } else {
    printf("The system is in an unsafe state and might lead to deadlock.\n");
    }
    return 0;
}
```



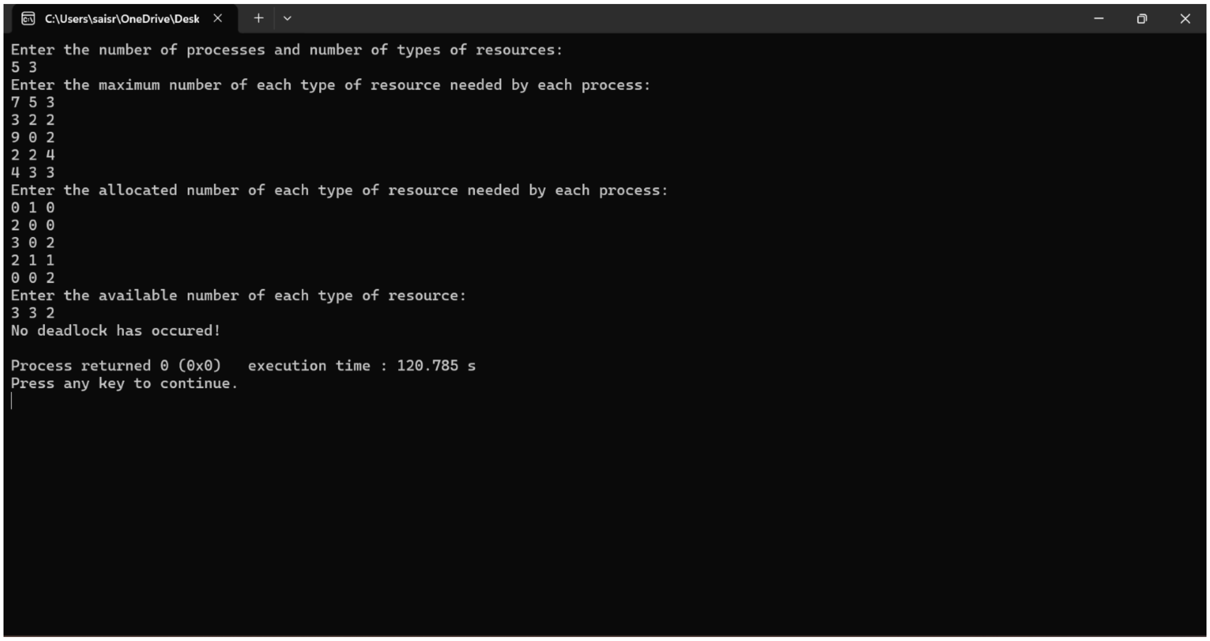
```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the available resources: 3 3 2
Enter the maximum resources for each process:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the allocated resources for each process:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2
 Process   Allocation   Max   Need
| P1      | 0 1 0 | 7 5 3 | 7 4 3 |
| P2      | 3 0 2 | 3 2 2 | 0 2 0 |
| P3      | 3 0 2 | 9 0 2 | 6 0 0 |
| P4      | 2 1 1 | 2 2 2 | 0 1 1 |
| P5      | 0 0 2 | 4 3 3 | 4 3 1 |
The system is in a safe state.
Safety sequence: P2 P3 P4 P5 P1

Process returned 0 (0x0)   execution time : 109.005 s
Press any key to continue.
```

8. Write a C program to simulate deadlock detection

```
#include<stdio.h>
void main()
{
   int n,m,i,j;
   printf("Enter the number of processes and number of types of resources:\n");
   scanf("%d %d",&n,&m);
   int max[n][m],need[n][m],all[n][m],ava[m],flag=1,finish[n],dead[n],c=0;
   printf("Enter the maximum number of each type of resource needed by each process:\n");
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         scanf("%d",&max[i][j]);
      }
   }
   printf("Enter the allocated number of each type of resource needed by each process:\n");
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         scanf("%d",&all[i][j]);
      }
   }
   printf("Enter the available number of each type of resource:\n");
   for(j=0;j<m;j++)
   {
```

```
   scanf("%d",&ava[j]);
}
for(i=0;i<n;i++)
{
   for(j=0;j<m;j++)
   {
      need[i][j]=max[i][j]-all[i][j];
   }
}
for(i=0;i<n;i++)
{
   finish[i]=0;
}
while(flag)
{
   flag=0;
   for(i=0;i<n;i++)
   {
      c=0;
      for(j=0;j<m;j++)
      {
         if(finish[i]==0 && need[i][j]<=ava[j])
         {
            c++;
            if(c==m)
            {
               for(j=0;j<m;j++)
               {
                  ava[j]+=all[i][j];
                  finish[i]=1;
                  flag=1;
               }
               if(finish[i]==1)
               {
                  i=n;
               }
            }
         }
      }
   }
}
j=0;
flag=0;
for(i=0;i<n;i++)
```

```
  {
    if(finish[i]==0)
    {
      dead[j]=i;
      j++;
      flag=1;
    }
  }
  if(flag==1)
  {
    printf("Deadlock has occured:\n");
    printf("The deadlock processes are:\n");
    for(i=0;i<n;i++)
    {
      printf("P%d ",dead[i]);
    }
  }
  else
  printf("No deadlock has occured!\n");
}
```



```
Enter the number of processes and number of types of resources:
5 3
Enter the maximum number of each type of resource needed by each process:
7 5 3
3 2 2
9 0 2
2 2 4
4 3 3
Enter the allocated number of each type of resource needed by each process:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available number of each type of resource:
3 3 2
No deadlock has occured!

Process returned 0 (0x0)   execution time : 120.785 s
Press any key to continue.
```

9. Write a C program to simulate the following contiguous memory allocation techniques

a) Worst-fit

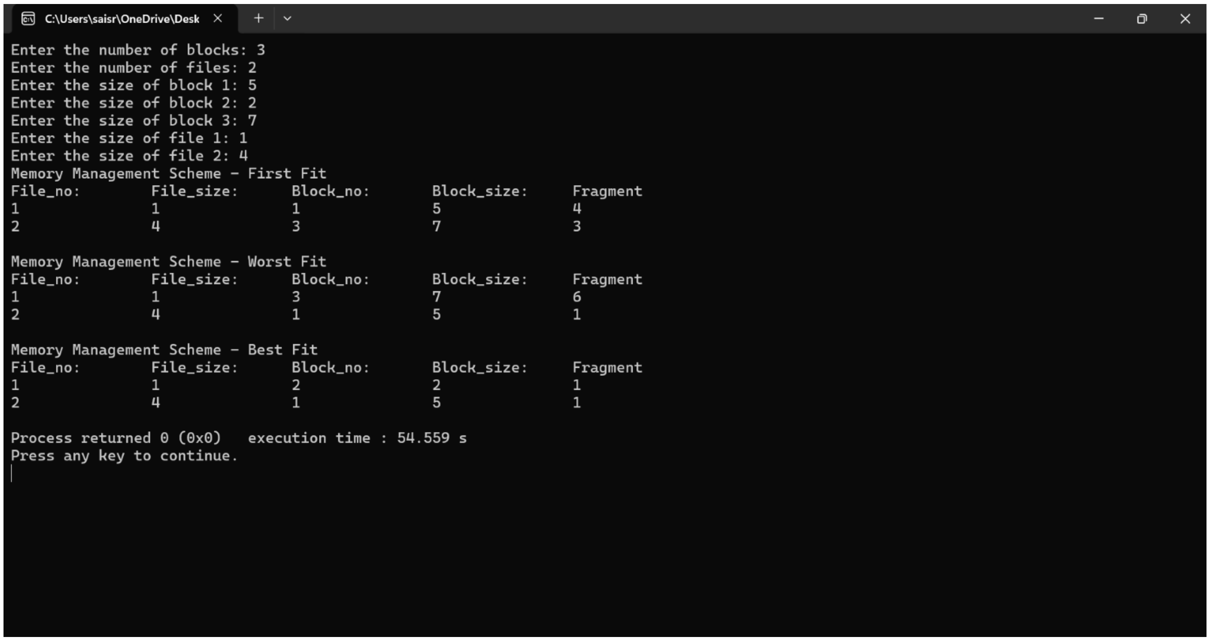
b) Best-fit

c) First-fit

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

```
struct Block {
        int block_no;
        int block_size;
        int is_free; // 1 for free, 0 for allocated
};

struct File {
        int file_no;
        int file_size;
};

void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
        printf("Memory Management Scheme - First Fit\n");
        printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

        for (int i = 0; i < n_files; i++) {
        for (int j = 0; j < n_blocks; j++) {
        if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
        blocks[j].is_free = 0;
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size, blocks[j].block_no,
blocks[j].block_size, blocks[j].block_size - files[i].file_size);
        break;
        }
        }
        }
}

void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
        printf("Memory Management Scheme - Worst Fit\n");
        printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

        for (int i = 0; i < n_files; i++) {
        int worst_fit_block = -1;
        int max_fragment = -1; // Initialize with a negative value
        for (int j = 0; j < n_blocks; j++) {
        if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
        int fragment = blocks[j].block_size - files[i].file_size;
        if (fragment > max_fragment) {
                max_fragment = fragment;
                worst_fit_block = j;
        }
        }
        }

        if (worst_fit_block!= -1) {
```

```
        blocks[worst_fit_block].is_free = 0;
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[worst_fit_block].block_no, blocks[worst_fit_block].block_size, max_fragment);
        }
        }
}

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
        printf("Memory Management Scheme - Best Fit\n");
        printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

        for (int i = 0; i < n_files; i++) {
        int best_fit_block = -1;
        int min_fragment = 10000; // Initialize with a large value
        for (int j = 0; j < n_blocks; j++) {
        if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
        int fragment = blocks[j].block_size - files[i].file_size;
        if (fragment < min_fragment) {
                min_fragment = fragment;
                best_fit_block = j;
        }
        }
        }

        if (best_fit_block!= -1) {
        blocks[best_fit_block].is_free = 0;
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[best_fit_block].block_no, blocks[best_fit_block].block_size, min_fragment);
        }
        }
}

int main() {
        int n_blocks, n_files;
        printf("Enter the number of blocks: ");
        scanf("%d", &n_blocks);
        printf("Enter the number of files: ");
        scanf("%d", &n_files);

        struct Block blocks[n_blocks];
        for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;
        }
        struct File files[n_files];
        for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
        }

        firstFit(blocks, n_blocks, files, n_files);
        printf("\n");

        // Reset blocks for worst fit
        for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
        }

        worstFit(blocks, n_blocks, files, n_files);
        printf("\n");

// Reset blocks for best fit
        for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
        }

        bestFit(blocks, n_blocks, files, n_files);

        return 0;
}
```



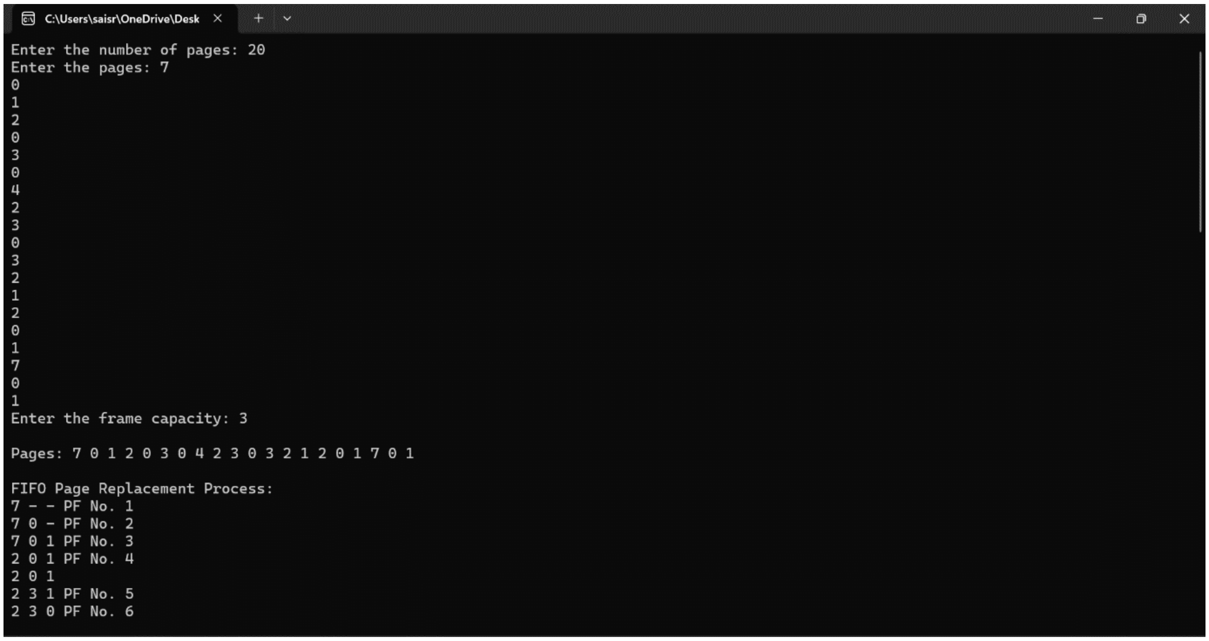
```
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of block 1: 5
Enter the size of block 2: 2
Enter the size of block 3: 7
Enter the size of file 1: 1
Enter the size of file 2: 4
Memory Management Scheme - First Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               1               1               5               4
2               4               3               7               3

Memory Management Scheme - Worst Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               1               3               7               6
2               4               1               5               1

Memory Management Scheme - Best Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               1               2               2               1
2               4               1               5               1

Process returned 0 (0x0)   execution time : 54.559 s
Press any key to continue.
```

10. Write a C program to simulate paging technique of memory management.

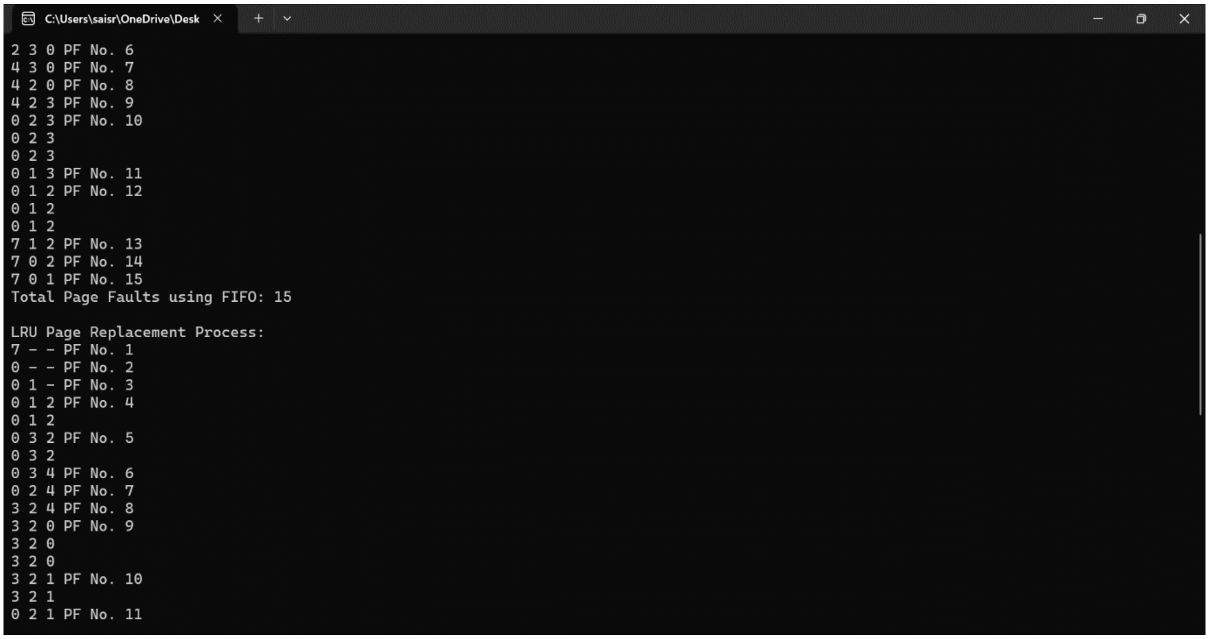
```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

void print_frames(int frame[], int capacity, int page_faults) {
        for (int i = 0; i < capacity; i++) {
        if (frame[i] == -1)
        printf("- ");
        else
        printf("%d ", frame[i]);
        }
        if (page_faults > 0)
        printf("PF No. %d", page_faults);
        printf("\n");
}

void fifo(int pages[], int n, int capacity) {
        int frame[capacity], index = 0, page_faults = 0;
        for (int i = 0; i < capacity; i++)
        frame[i] = -1;

        printf("FIFO Page Replacement Process:\n");
        for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
        if (frame[j] == pages[i]) {
        found = 1;
        break;
        }
        }
        if (!found) {
        frame[index] = pages[i];
        index = (index + 1) % capacity;
        page_faults++;
        }
        print_frames(frame, capacity, found ? 0 : page_faults);
        }
        printf("Total Page Faults using FIFO: %d\n\n", page_faults);
}

void lru(int pages[], int n, int capacity) {
        int frame[capacity], counter[capacity], time = 0, page_faults = 0;
        for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
        counter[i] = 0;
        }
```

```
    printf("LRU Page Replacement Process:\n");
    for (int i = 0; i < n; i++) {
    int found = 0;
    for (int j = 0; j < capacity; j++) {
    if (frame[j] == pages[i]) {
    found = 1;
    counter[j] = time++;
    break;
    }
    }
    if (!found) {
    int min = INT_MAX, min_index = -1;
    for (int j = 0; j < capacity; j++) {
    if (counter[j] < min) {
        min = counter[j];
        min_index = j;
    }
    }
    frame[min_index] = pages[i];
    counter[min_index] = time++;
    page_faults++;
    }
    print_frames(frame, capacity, found ? 0 : page_faults);
    }
    printf("Total Page Faults using LRU: %d\n\n", page_faults);
}

void optimal(int pages[], int n, int capacity) {
    int frame[capacity], page_faults = 0;
    for (int i = 0; i < capacity; i++)
    frame[i] = -1;

    printf("Optimal Page Replacement Process:\n");
    for (int i = 0; i < n; i++) {
    int found = 0;
    for (int j = 0; j < capacity; j++) {
    if (frame[j] == pages[i]) {
    found = 1;
    break;
    }
    }
    if (!found) {
    int farthest = i + 1, index = -1;
    for (int j = 0; j < capacity; j++) {
    int k;
    for (k = i + 1; k < n; k++) {
        if (frame[j] == pages[k])
        break;
    }
    if (k > farthest) {
        farthest = k;
        index = j;
    }
    }
    if (index == -1) {
    for (int j = 0; j < capacity; j++) {
        if (frame[j] == -1) {
        index = j;
        break;
        }
    }
    }
    frame[index] = pages[i];
    page_faults++;
    }
    print_frames(frame, capacity, found ? 0 : page_faults);
    }
    printf("Total Page Faults using Optimal: %d\n\n", page_faults);
}

int main() {
    int n, capacity;
    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int *pages = (int*)malloc(n * sizeof(int));
    printf("Enter the pages: ");
    for (int i = 0; i < n; i++)
    scanf("%d", &pages[i]);
    printf("Enter the frame capacity: ");
    scanf("%d", &capacity);

    printf("\nPages: ");
    for (int i = 0; i < n; i++)
    printf("%d ", pages[i]);
    printf("\n\n");

    fifo(pages, n, capacity);
    lru(pages, n, capacity);
    optimal(pages, n, capacity);

    free(pages);
```

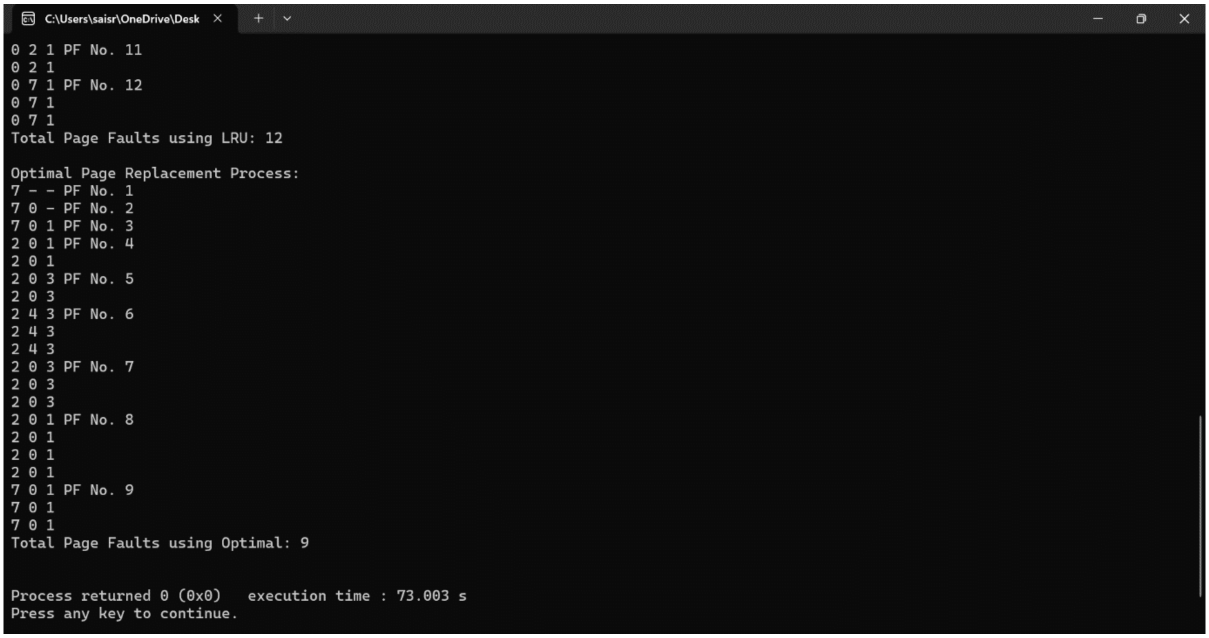
```
            return 0;
}
```



```
Enter the number of pages: 20
Enter the pages: 7
0
1
2
0
3
0
4
2
3
0
3
2
1
2
0
1
7
0
1
Enter the frame capacity: 3

Pages: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

FIFO Page Replacement Process:
7 - - PF No. 1
7 0 - PF No. 2
7 0 1 PF No. 3
2 0 1 PF No. 4
2 0 1
2 3 1 PF No. 5
2 3 0 PF No. 6
```



```
2 3 0 PF No. 6
4 3 0 PF No. 7
4 2 0 PF No. 8
4 2 3 PF No. 9
0 2 3 PF No. 10
0 2 3
0 2 3
0 1 3 PF No. 11
0 1 2 PF No. 12
0 1 2
0 1 2
7 1 2 PF No. 13
7 0 2 PF No. 14
7 0 1 PF No. 15
Total Page Faults using FIFO: 15

LRU Page Replacement Process:
7 - - PF No. 1
0 - - PF No. 2
0 1 - PF No. 3
0 1 2 PF No. 4
0 1 2
0 3 2 PF No. 5
0 3 2
0 3 4 PF No. 6
0 2 4 PF No. 7
3 2 4 PF No. 8
3 2 0 PF No. 9
3 2 0
3 2 0
3 2 1 PF No. 10
3 2 1
0 2 1 PF No. 11
```



```
0 2 1 PF No. 11
0 2 1
0 7 1 PF No. 12
0 7 1
0 7 1
Total Page Faults using LRU: 12

Optimal Page Replacement Process:
7 - - PF No. 1
7 0 - PF No. 2
7 0 1 PF No. 3
2 0 1 PF No. 4
2 0 1
2 0 3 PF No. 5
2 0 3
2 4 3 PF No. 6
2 4 3
2 4 3
2 0 3 PF No. 7
2 0 3
2 0 3
2 0 1 PF No. 8
2 0 1
2 0 1
2 0 1
7 0 1 PF No. 9
7 0 1
7 0 1
Total Page Faults using Optimal: 9


Process returned 0 (0x0)   execution time : 73.003 s
Press any key to continue.
```