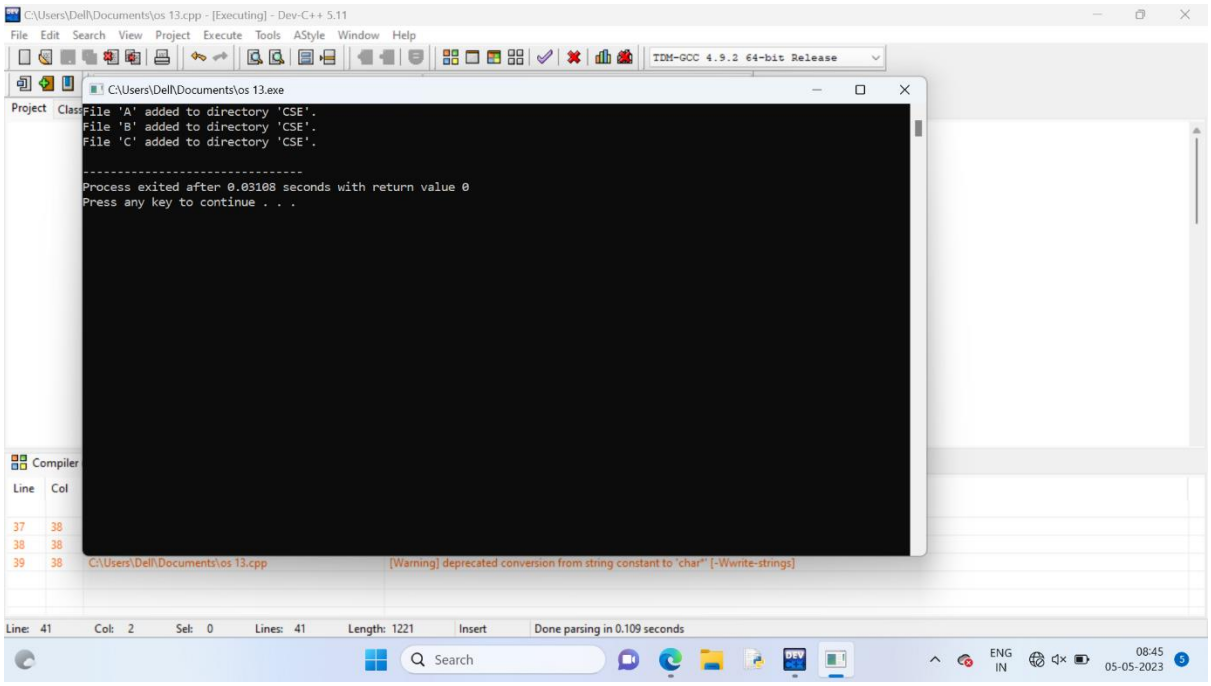
13. Write a C program to implement single-level directory system. In which all the files are placed in one directory and there are no sub directories.

Test Case: Create one directory with the name of CSE and Add 3 files(A,B,C) in to that directory

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_FILES 100
#define MAX_FILENAME_LENGTH 20
struct file {
   char name[MAX_FILENAME_LENGTH];
   int size;
};
struct directory {
   char name[MAX_FILENAME_LENGTH];
   struct file files[MAX_FILES];
   int num_files;
};
void add_file(struct directory *dir, char *filename, int size) {
   if (dir->num_files >= MAX_FILES) {
      printf("Directory is full.\n");
      return;
   }
   for (int i = 0; i < dir->num_files; i++) {
      if (strcmp(dir->files[i].name, filename) == 0) {
         printf("File already exists in directory.\n");
         return;
      }
   }
   struct file new_file;
   strncpy(new_file.name, filename, MAX_FILENAME_LENGTH);
   new_file.size = size;
   dir->files[dir->num_files] = new_file;
   dir->num_files++;
         printf("File '%s' added to directory '%s'.\n", filename, dir->name);
}
int main() {
   struct directory cse_directory;
   strncpy(cse_directory.name, "CSE", MAX_FILENAME_LENGTH);
   cse_directory.num_files = 0;
   add_file(&cse_directory, "A", 100);
   add_file(&cse_directory, "B", 200);
   add_file(&cse_directory, "C", 300);
          return 0;
}
```

Output:



Write a C program to illustrate the page replacement method where the page which is not in demand for the longest future time is replaced by the new page and determine the number of page faults for the following test case:

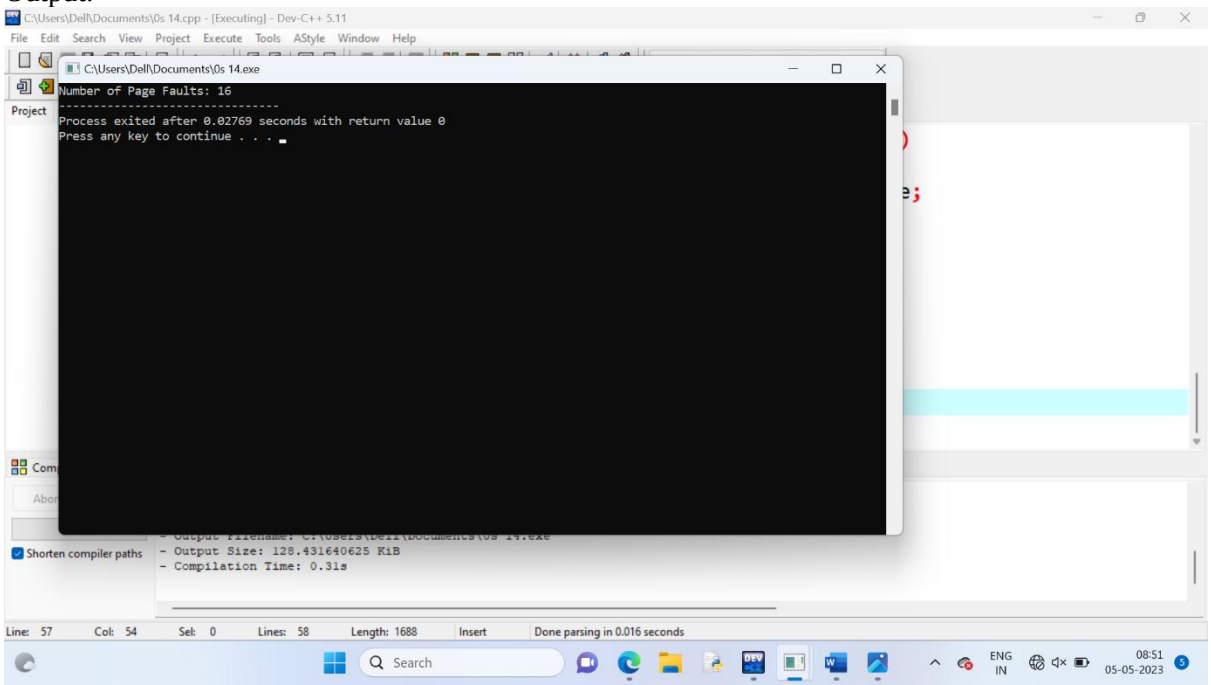
No. of page frames: 3; Page reference sequence 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0 and 1.

Program:

```
#include <stdio.h>
#define MAX_FRAMES 3
int main()
{
   int frames[MAX_FRAMES], pages[MAX_FRAMES], page_faults = 0;
   int page_reference[] = {7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1};
   int num_pages = sizeof(page_reference)/sizeof(page_reference[0]);
   int i, j, k, max_future_distance, page_to_replace;
         for(i = 0; i < MAX_FRAMES; i++)
   {
      frames[i] = -1;
      pages[i] = -1;
   }
         for(i = 0; i < num_pages; i++)
   {
      int page_found = 0;
      int page = page_reference[i];
                  for(j = 0; j < MAX_FRAMES; j++)
      {
         if(frames[j] == page)
         {
            page_found = 1;
            break;
         }
```

```
    }
                if(page_found == 0)
    {
      for(j = 0; j < MAX_FRAMES; j++)
      {
        int page_exists = 0;
        int future_distance = 0;
                                for(k = i + 1; k < num_pages; k++)
        {
          if(frames[j] == page_reference[k])
          {
            page_exists = 1;
            future_distance = k - i;
            break;
          }
        }
                                if(page_exists == 0)
        {
          page_faults++;
          frames[j] = page;
          break;
        }
                                if(future_distance > max_future_distance)
        {
          max_future_distance = future_distance;
          page_to_replace = j;
        }
      }
                          page_faults++;
      frames[page_to_replace] = page;
    }
  }
     printf("Number of Page Faults: %d", page_faults);
}
```

Output:



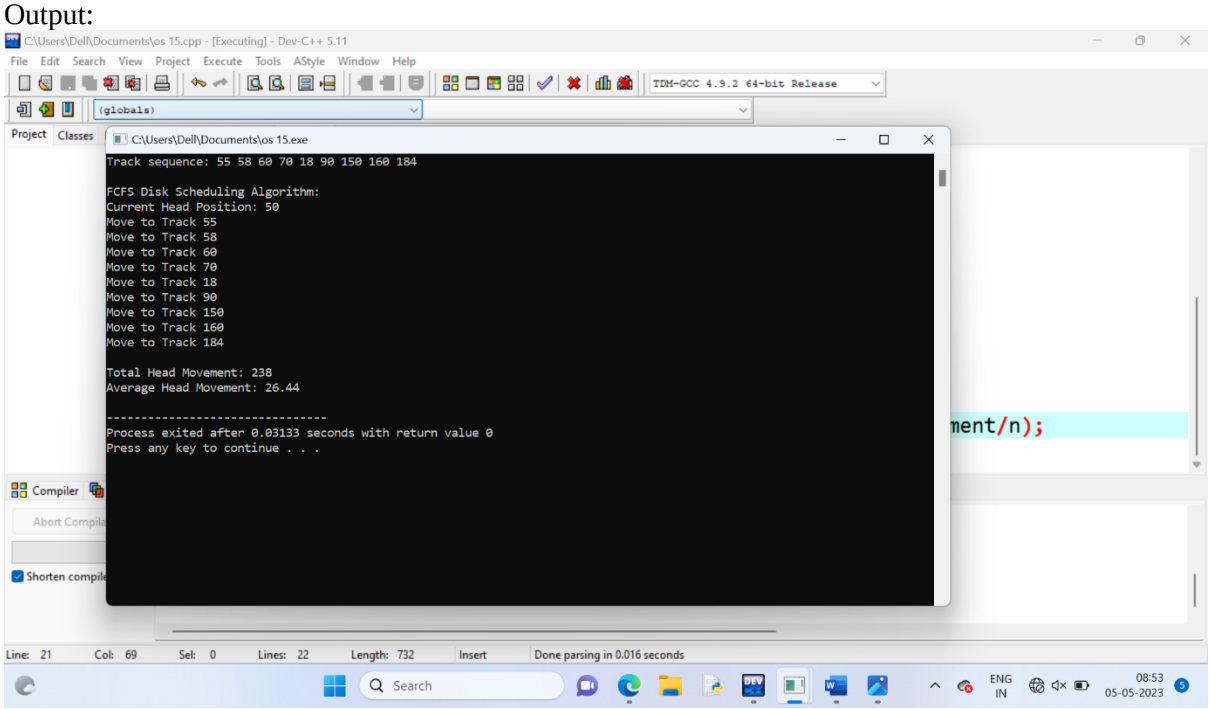
15.Write a C program to simulate FCFS disk scheduling algorithms and execute your program and find out and print the average head movement for the following test case.

No of tracks:9; Track position:55 58 60 70 18 90 150 160 184

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int main()
{
   int n=9, head_pos=50, curr_track, head_movement=0, i;
   int tracks[] = {55, 58, 60, 70, 18, 90, 150, 160, 184};
   printf("Track sequence: ");
   for(i=0; i<n; i++)
      printf("%d ", tracks[i]);
   printf("\n\nFCFS Disk Scheduling Algorithm:\n");
   printf("Current Head Position: %d\n", head_pos);
   for(i=0; i<n; i++)
   {
      curr_track = tracks[i];
      head_movement += abs(head_pos - curr_track);
      head_pos = curr_track;
      printf("Move to Track %d\n", curr_track);
   }
   printf("\nTotal Head Movement: %d\n", head_movement);
   printf("Average Head Movement: %.2f\n", (float)head_movement/n);
}
```

Output:



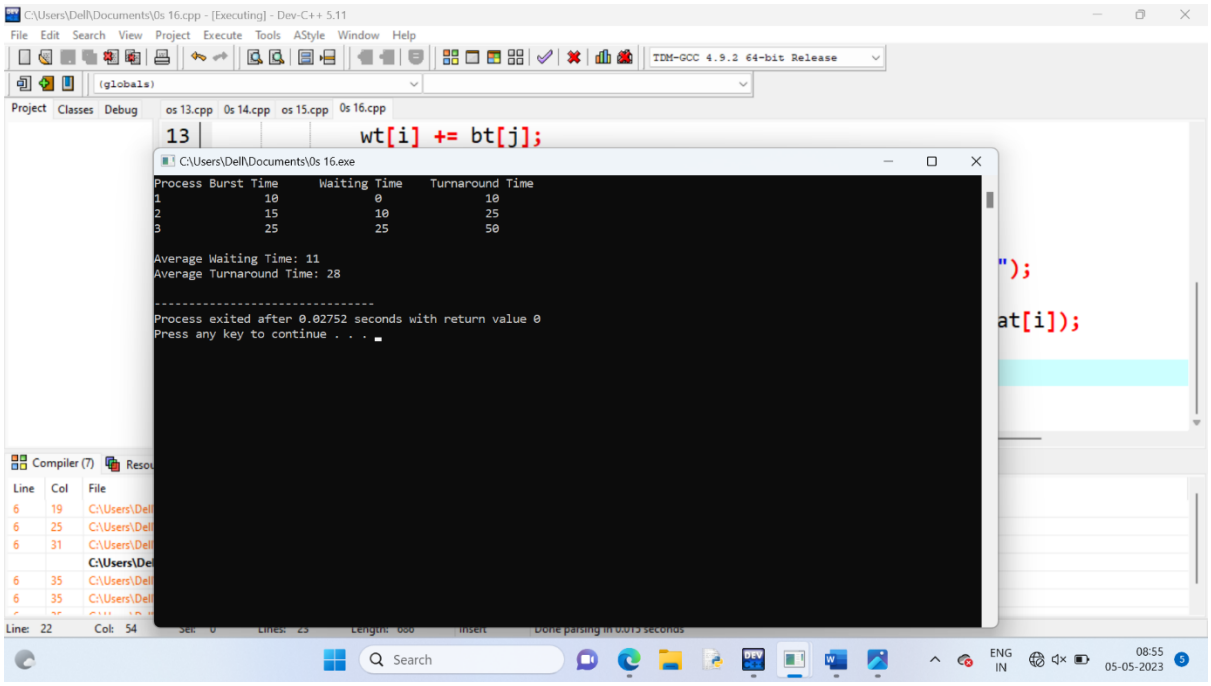
16. Write a program to compute the average waiting time and average turnaround time based on First Come First Serve for the following process with the given CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process	Burst Time
P1	10
P2	15
P3	25

Program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int n=3, i, j, wt[10]={0}, tat[10]={0}, bt[10]={10, 15, 25}, ct=0, avg_wt=0, avg_tat=0;
   char pid[] = {'P1', 'P2', 'P3'};
   for(i=0; i<n; i++)
   {
      ct += bt[i];
      tat[i] = ct;
      avg_tat += tat[i];
                  for(j=0; j<i; j++)
         wt[i] += bt[j];
      avg_wt += wt[i];
   }
   avg_wt /= n;
   avg_tat /= n;
   printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
   for(i=0; i<n; i++)
      printf("%c\t\t%d\t\t%d\t\t%d\n", pid[i], bt[i], wt[i], tat[i]);
   printf("\nAverage Waiting Time: %d\n", avg_wt);
   printf("Average Turnaround Time: %d\n", avg_tat);
}
```

Output:



17. Write a program to compute the average waiting time and average turnaround time based on Round Robin scheduling for the following process with the given CPU burst times and quantum time slots 4 ms, (and the assumption that all jobs arrive at the same time.)

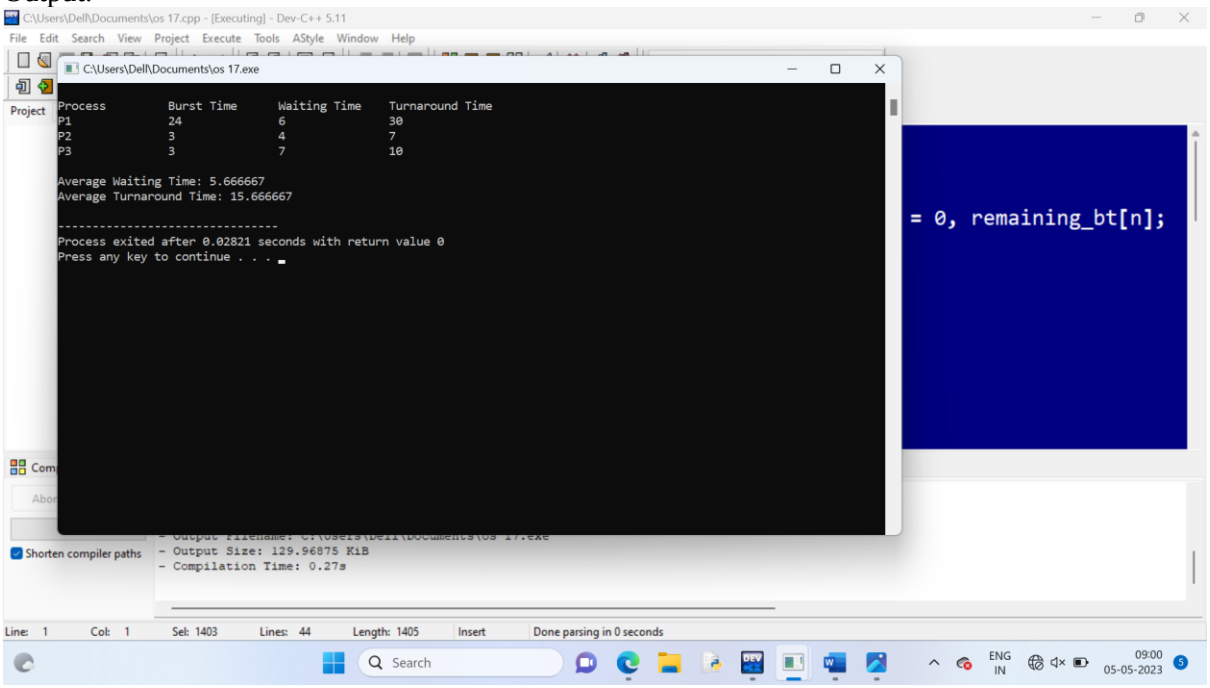
Process	Burst Time
P1	24
P2	3
P3	3

Program:

```
#include<stdio.h>
#include<stdlib.h>
void round_robin(int bt[], int n, int quantum) {
   int wt[n], tat[n], total_wt = 0, total_tat = 0, time = 0, remaining_bt[n];
   for(int i = 0; i < n; i++)
      remaining_bt[i] = bt[i];
   while(1) {
      int done = 1;
      for(int i = 0; i < n; i++) {
         if(remaining_bt[i] > 0) {
            done = 0;
            if(remaining_bt[i] > quantum) {
               time += quantum;
               remaining_bt[i] -= quantum;
            }
            else {
```

```
                time += remaining_bt[i];
                wt[i] = time - bt[i];
                remaining_bt[i] = 0;
            }
        }
    }
    if(done == 1)
        break;
  }
  for(int i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
    total_wt += wt[i];
    total_tat += tat[i];
  }
  printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
  for(int i = 0; i < n; i++) {
    printf("\nP%d\t\t%d\t\t%d\t\t%d", i+1, bt[i], wt[i], tat[i]);
  }
  float avg_wt = (float) total_wt / n;
  float avg_tat = (float) total_tat / n;
  printf("\n\nAverage Waiting Time: %f", avg_wt);
  printf("\nAverage Turnaround Time: %f\n", avg_tat);
}
int main() {
  int n = 3, quantum = 4;
  int bt[] = {24, 3, 3};
  round_robin(bt, n, quantum);
}
```

Output:



18. 18. Write a program for solving the producer consumer problem with the following scenario: The producer should produce data only when the buffer is not

full. Data can only be consumed by the consumer if and only if the memory buffer is not empty.

Test Case:

Buffer Size: 3

Consume an item in the beginning and show that the buffer is EMPTY

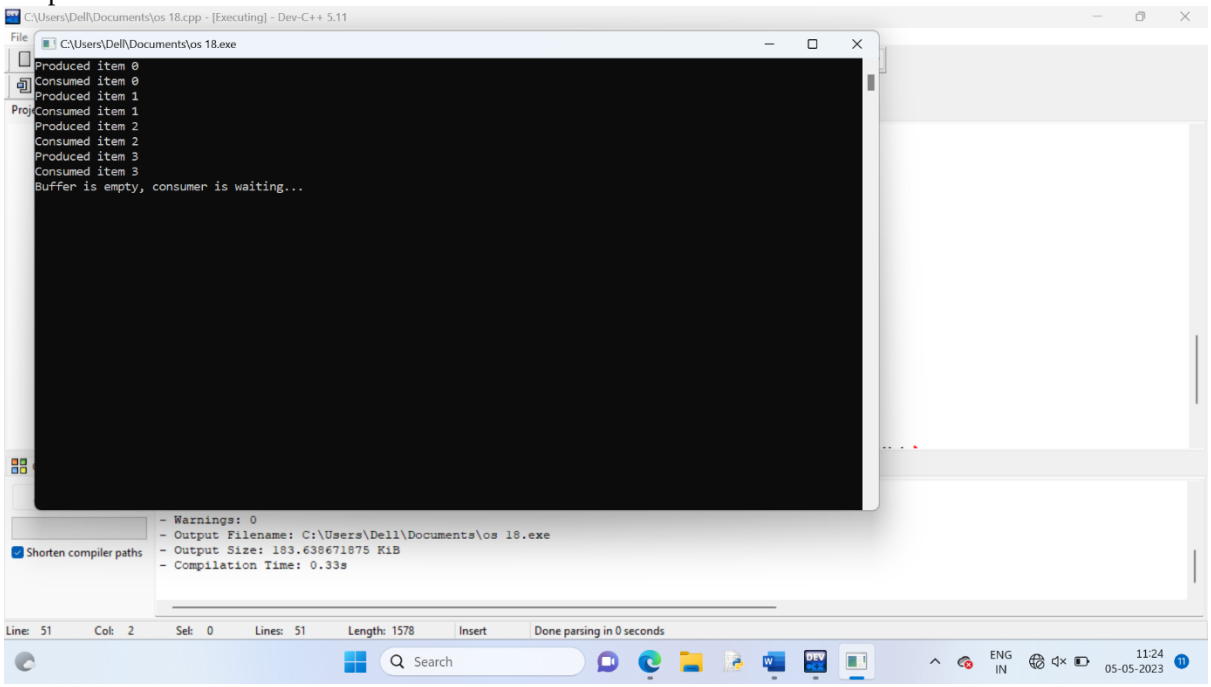
Produce 4 items and show that the buffer is FULL

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define BUFFER_SIZE 3
int buffer[BUFFER_SIZE];
int count = 0;
int in = 0;
int out = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
void *producer(void *arg) {
   int i;
   for (i = 0; i < 4; i++) {
      pthread_mutex_lock(&mutex);
      while (count == BUFFER_SIZE) {
         printf("Buffer is full, producer is waiting...\n");
         pthread_cond_wait(&full, &mutex);
      }
      buffer[in] = i;
      printf("Produced item %d\n", buffer[in]);
      in = (in + 1) % BUFFER_SIZE;
      count++;
      pthread_cond_signal(&empty);
      pthread_mutex_unlock(&mutex);
   }
   pthread_exit(NULL);
}
void *consumer(void *arg) {
   int item;
   while (1) {
      pthread_mutex_lock(&mutex);
      while (count == 0) {
         printf("Buffer is empty, consumer is waiting...\n");
         pthread_cond_wait(&empty, &mutex);
      }
      item = buffer[out];
      printf("Consumed item %d\n", item);
      out = (out + 1) % BUFFER_SIZE;
      count--;
```

```
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&mutex);
    }
}
int main() {
    pthread_t producer_thread, consumer_thread;
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
}
```

Output:



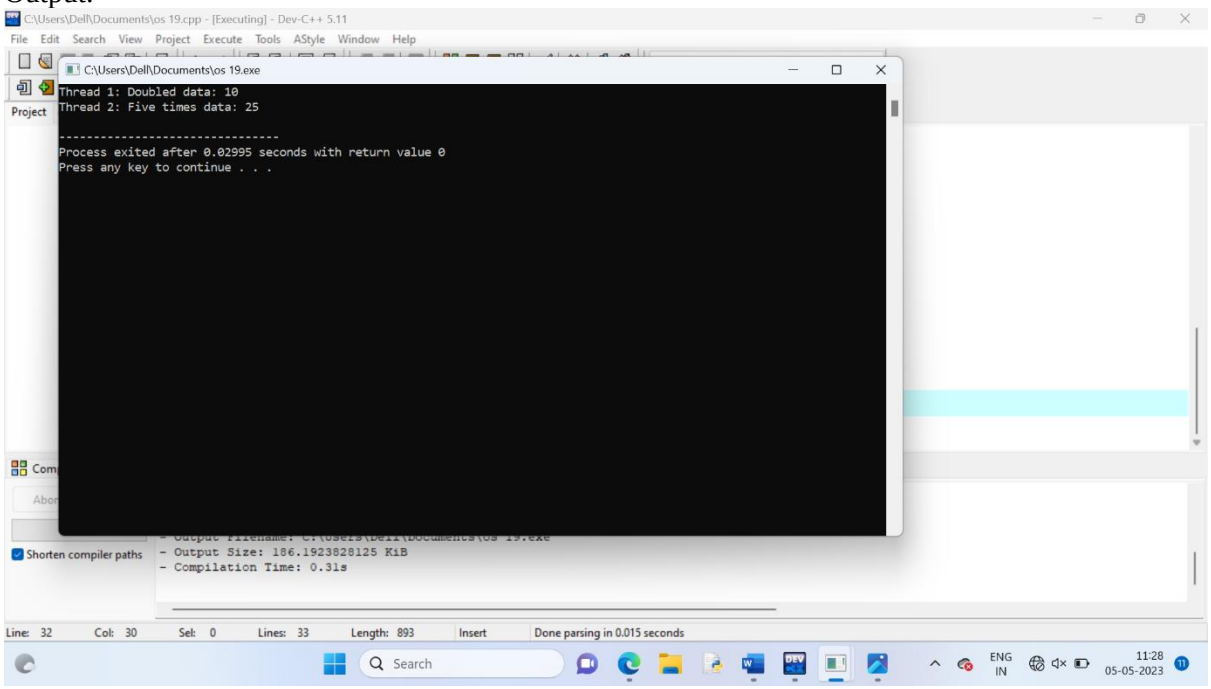
19. Write a C program to create two threads to access shared memory which is an integer in a synchronized fashion using semaphore. In the first thread print the doubled the integer data after reading from the shared memory. In the second thread, print the five times of the integer data after reading from the shared memory

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
int shared_data = 5;
sem_t semaphore;
void* thread1_func(void* arg) {
    int data;
    sem_wait(&semaphore);
    data = shared_data;
    data = data * 2;
    printf("Thread 1: Doubled data: %d\n", data);
    sem_post(&semaphore);
    pthread_exit(NULL);
}
void* thread2_func(void* arg) {
    int data;
```

```
    sem_wait(&semaphore);
    data = shared_data;
    data = data * 5;
    printf("Thread 2: Five times data: %d\n", data);
    sem_post(&semaphore);
    pthread_exit(NULL);
}
int main() {
    pthread_t thread1, thread2;
    sem_init(&semaphore, 0, 1);
    pthread_create(&thread1, NULL, thread1_func, NULL);
    pthread_create(&thread2, NULL, thread2_func, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    sem_destroy(&semaphore);
}
```

Output:



20. Write a C program to implement the worst-fit algorithm and allocate the memory block to each process.

Test Case:

Memory partitions: 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order),

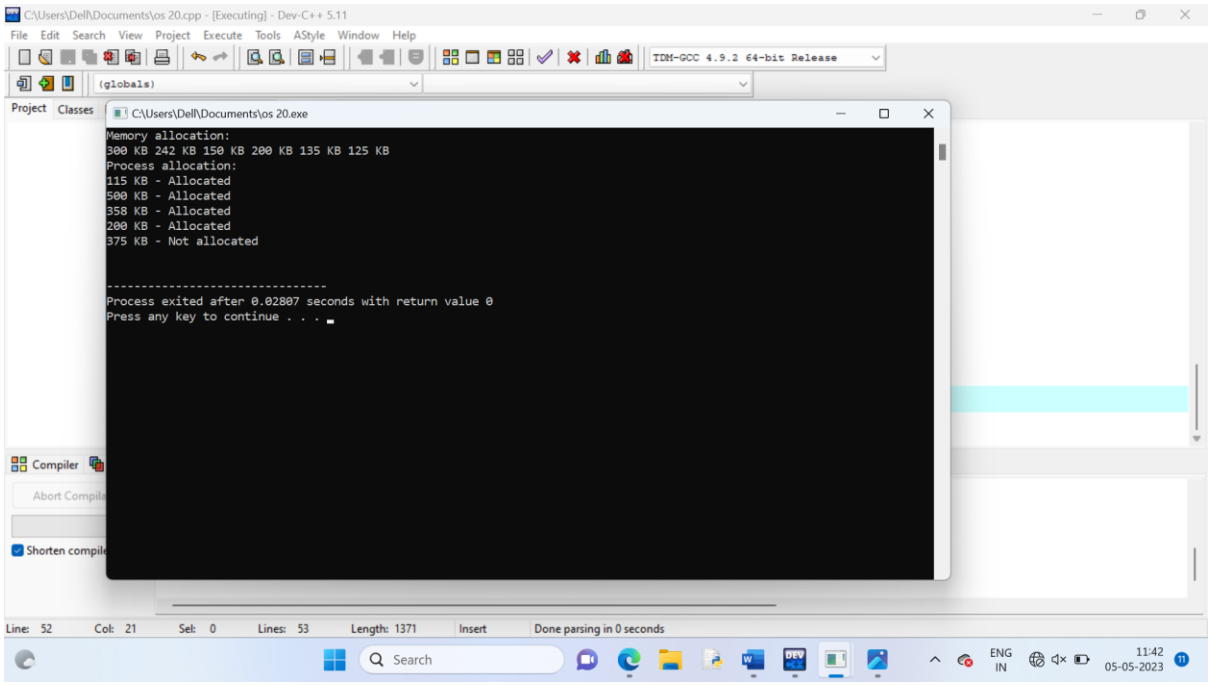
Show the outcome for the test case with the worst-fit algorithms to place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)

Program:

```
#include<stdio.h>
#include<stdlib.h>
int memory_size = 6;
```

```
int process_size = 5;
int memory[6] = {300, 600, 350, 200, 750, 125};
int process[5] = {115, 500, 358, 200, 375};
int allocated[5] = {0};
void worst_fit_algorithm() {
   int i, j, max_index;
   for(i = 0; i < process_size; i++) {
      max_index = -1;
      for(j = 0; j < memory_size; j++) {
         if(memory[j] >= process[i]) {
            if(max_index == -1) {
               max_index = j;
            }
            else if(memory[j] > memory[max_index]) {
               max_index = j;
            }
         }
      }
      if(max_index != -1) {
         allocated[i] = 1;
         memory[max_index] -= process[i];
      }
   }
}
void print_memory() {
   int i;
   printf("Memory allocation:\n");
   for(i = 0; i < memory_size; i++) {
      printf("%d KB ", memory[i]);
   }
   printf("\n");
}
void print_process() {
   int i;
   printf("Process allocation:\n");
   for(i = 0; i < process_size; i++) {
      if(allocated[i]) {
         printf("%d KB - Allocated\n", process[i]);
      }
      else {
         printf("%d KB - Not allocated\n", process[i]);
      }
   }
   printf("\n");
}
int main() {
   worst_fit_algorithm();
   print_memory();
   print_process();
}
```

Output:


C:\Users\Dell\Documents\os 20.cpp - [Executing] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
Project Classes
C:\Users\Dell\Documents\os 20.exe
Memory allocation:
300 KB 242 KB 150 KB 200 KB 135 KB 125 KB
Process allocation:
115 KB - Allocated
500 KB - Allocated
358 KB - Allocated
200 KB - Allocated
375 KB - Not allocated

Process exited after 0.02807 seconds with return value 0
Press any key to continue . . .
Compiler
Line: 52 Col: 21 Sel: 0 Lines: 53 Length: 1371 Insert Done parsing in 0 seconds
Search
ENG
IN
11:42
05-05-2023