ASSIGNMENT

By
Ujjawal Singh Charak
2022A1R029
3rd sem
CSE department

Model Institute of Engineering & Technology (Autonomous)
(Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade)
Jammu, India
2023

ASSIGNMENT

Subject Code: Operating System [COM-302]

Due Date: 4/12/23

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
Total Marks			20	
Faculty Signature: Dr. Mekhla Sharma (Assistant Professor) **Email:** mekhla.cse@mietjammu.in				

Table Of Contents

Task-1

Write a program that simulates page replacement algorithms like FIFO, LRU, and Optimal Page Replacement. Create a memory management system that swaps pages in and out to demonstrate the effectiveness of these algorithms in different scenarios.

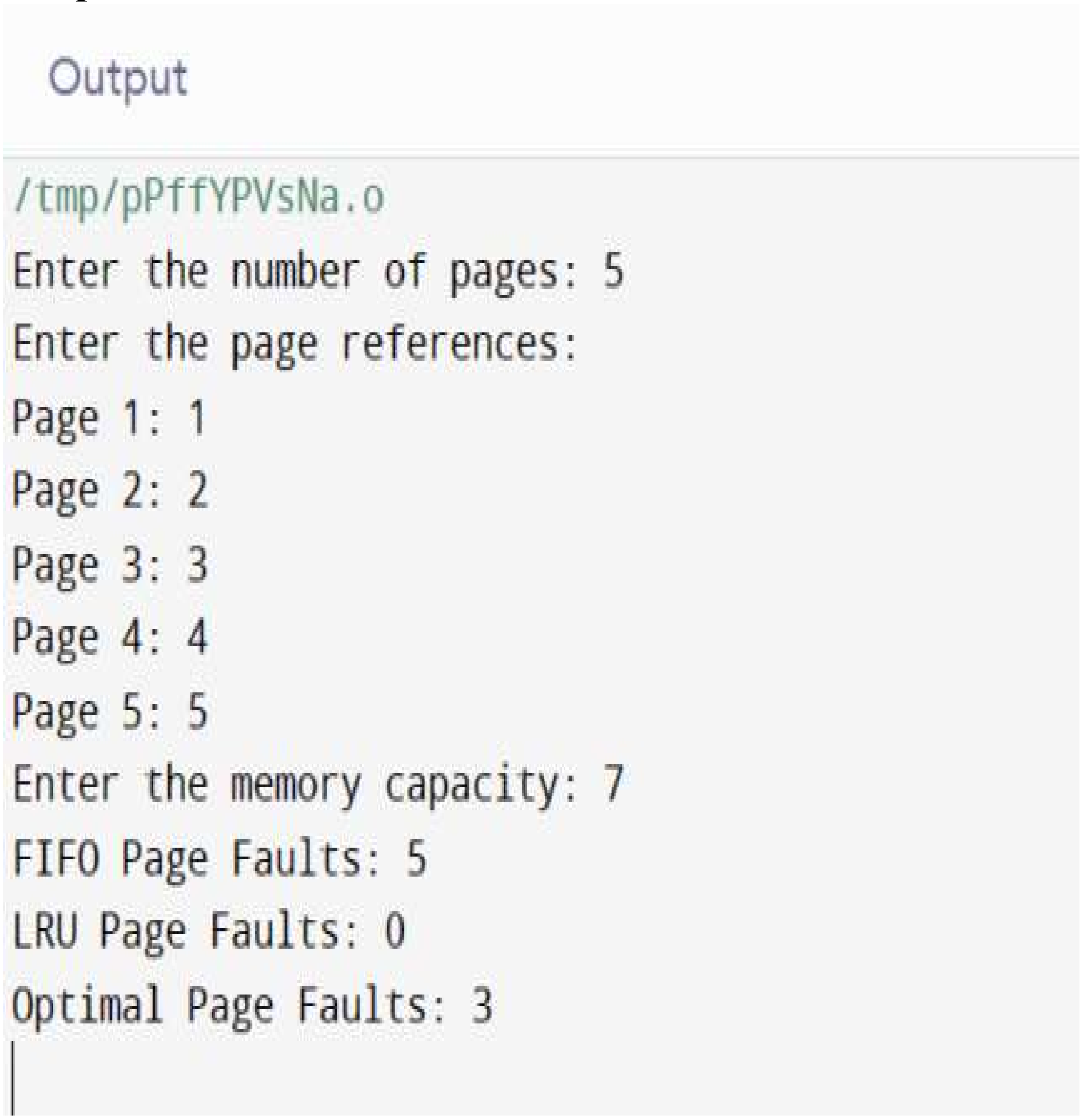
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_PAGES 20

// Function to simulate FIFO page replacement algorithm
void fifo(int pages[], int n, int capacity) {
    int page_queue[capacity];  // Circular queue to store pages
    int front = 0, rear = 0;   // Pointers for front and rear of the queue
    int page_faults = 0;       // Counter for page faults

    for (int i = 0; i < n; ++i) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in the queue
        for (int j = 0; j < capacity; ++j) {
            if (page_queue[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            ++page_faults;
            // If the queue is full, replace the front page
            if (rear == capacity) {
                rear = 0; // Wrap around in a circular queue
            }
            page_queue[rear++] = page;  // Add the new page to the
queue
        }
    }

    printf("FIFO Page Faults: %d\n", page_faults);
}

// Function to simulate LRU page replacement algorithm
void lru(int pages[], int n, int capacity) {
    int page_order[MAX_PAGES];  // Array to store the order of
pages
    int page_faults = 0;        // Counter for page faults

    for (int i = 0; i < n; ++i) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in the order array
        for (int j = 0; j < capacity; ++j) {
            if (page_order[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            ++page_faults;
            // If the array is full, shift all elements to the left
            if (capacity == n) {
                for (int k = 0; k < capacity - 1; ++k) {
                    page_order[k] = page_order[k + 1];
                }
            }
            page_order[capacity - 1] = page;  // Add the new page to
the end of the array
        } else {
            // If the page is already in the array, move it to the end
            for (int k = 0; k < capacity; ++k) {
                if (page_order[k] == page) {
                    for (int l = k; l < capacity - 1; ++l) {
                        page_order[l] = page_order[l + 1];
                    }
                    page_order[capacity - 1] = page;  // Move the page to
the end
                    break;
                }
            }
        }
    }

    printf("LRU Page Faults: %d\n", page_faults);
```

```
}

// Function to simulate Optimal page replacement algorithm
void optimal(int pages[], int n, int capacity) {
   int page_order[MAX_PAGES];  // Array to store the order of
pages
   int page_faults = 0;        // Counter for page faults

   for (int i = 0; i < n; ++i) {
      int page = pages[i];
      int found = 0;

      // Check if the page is already in the order array
      for (int j = 0; j < capacity; ++j) {
         if (page_order[j] == page) {
            found = 1;
            break;
         }
      }

      if (!found) {
         ++page_faults;
         // If the array is full, find the page to be replaced that will
not be used for the longest time
         if (capacity == n) {
            int future_occurrences[MAX_PAGES];
            for (int k = 0; k < capacity; ++k) {
               // Find the index of the next occurrence of each page
in the remaining sequence
               int page_to_find = page_order[k];
```

```
                int found_index = -1;
                for (int l = i + 1; l < n; ++l) {
                    if (pages[l] == page_to_find) {
                        found_index = l;
                        break;
                    }
                }
                future_occurrences[k] = (found_index == -1) ? n + 1
: found_index;
            }
            // Find the page with the maximum future occurrence
index
            int max_index_page = 0;
            for (int k = 1; k < capacity; ++k) {
                if              (future_occurrences[k]              >
future_occurrences[max_index_page]) {
                    max_index_page = k;
                }
            }
            //  Replace  the  page  with  the  maximum  future
occurrence index
            page_order[max_index_page] = page;
        } else {
            // If the array is not full, add the page to the end of the
array
            page_order[capacity - 1] = page;
        }
    }
}
```

```c
    printf("Optimal Page Faults: %d\n", page_faults);
}

int main() {
    int pages[MAX_PAGES];
    int n, capacity;

    // Input the number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &n);

    // Input the page references
    printf("Enter the page references:\n");
    for (int i = 0; i < n; ++i) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

    // Input the memory capacity
    printf("Enter the memory capacity: ");
    scanf("%d", &capacity);

    // Run the page replacement algorithms
    fifo(pages, n, capacity);
    lru(pages, n, capacity);
    optimal(pages, n, capacity);

    return 0;
}
```

Output:



```
/tmp/pPffYPVsNa.o
Enter the number of pages: 5
Enter the page references:
Page 1: 1
Page 2: 2
Page 3: 3
Page 4: 4
Page 5: 5
Enter the memory capacity: 7
FIFO Page Faults: 5
LRU Page Faults: 0
Optimal Page Faults: 3
```

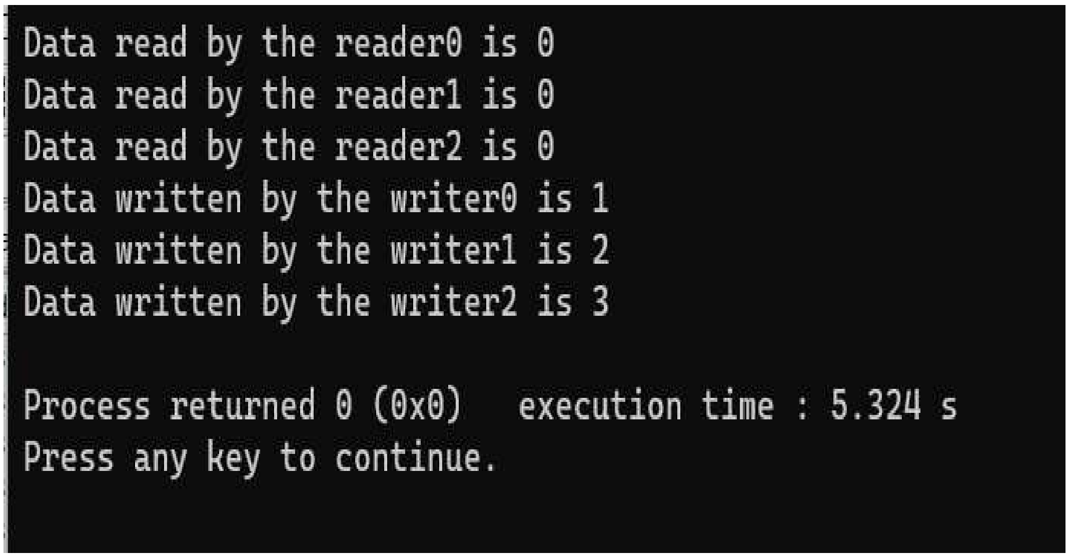
Task-2

Implement a program that simulates the Reader-Writer problem, allowing multiple readers or a single writer to access a shared resource. Use semaphores or another synchronization mechanism to maintain data consistency. Explain the differences between reader and writer processes in terms of synchronization.

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>


sem_t mutex, writeblock;

int data = 0, rcount = 0;


void *reader(void *arg)

{

  int f;

  f = ((int)arg);

  sem_wait(&mutex);

  rcount = rcount + 1;

  if(rcount==1)

    sem_wait(&writeblock);

  sem_post(&mutex);

  printf("Data read by the reader%d is %d\n",f, data);

  sleep(1);
```

```
 sem_wait(&mutex);
 rcount = rcount - 1;
 if(rcount==0)
   sem_post(&writeblock);
 sem_post(&mutex);
}

void *writer(void *arg)
{
 int f;
 f = ((int) arg);
 sem_wait(&writeblock);
 data++;
 printf("Data written by the writer%d is %d\n",f,data);
 sleep(1);
 sem_post(&writeblock);
}

int main()
{
 int i,b;
 pthread_t rtid[5],wtid[5];
```

```
  sem_init(&mutex,0,1);
  sem_init(&writeblock,0,1);
  for(i=0; i<=2; i++)
  {
    pthread_create(&wtid[i],NULL,writer,(void *)i);
    pthread_create(&rtid[i],NULL,reader,(void *)i);
  }
  for(i=0;i<=2;i++)
  {
    pthread_join(wtid[i],NULL);
    pthread_join(rtid[i],NULL);
  }
  return 0;
}
```



```
Data read by the reader0 is 0
Data read by the reader1 is 0
Data read by the reader2 is 0
Data written by the writer0 is 1
Data written by the writer1 is 2
Data written by the writer2 is 3

Process returned 0 (0x0)   execution time : 5.324 s
Press any key to continue.
```

In terms of synchronization, the differences between reader and writer processes are as follows:

- **Reader processes**: Multiple reader processes can read the shared resource simultaneously as long as there is no writer process writing to it. This is achieved by using a semaphore to keep track of the number of reader processes currently accessing the resource. When the first reader process enters, it blocks any writer processes. Subsequent reader processes can enter without blocking further because the writer processes are already blocked. When the last reader process is done, it unblocks the writer processes.

Writer processes: Writer processes require exclusive access to the shared resource. When a writer process is writing to the resource, all other processes (both reader and writer) are blocked from accessing the resource. This is achieved by using a semaphore that provides mutual exclusion to the shared resource for the writer processes. When a writer process wants to write, it checks this semaphore and if it’s available (i.e., no other writer process is currently writing), it proceeds, blocking all other processes. Once it’s done, it releases the semaphore, unblocking other processes.

Analysis Report

Task –1

The provided code above is a C program that simulates three different page replacement algorithms: FIFO (First-In-First-Out), LRU (Least Recently Used), and Optimal. These algorithms are commonly used in operating systems to manage page faults in virtual memory.

Analysis of the code:

Header Files and Definitions:

The code includes standard C libraries stdio.h and stdlib.h.

A macro MAX_PAGES is defined to set the maximum number of pages.

FIFO Algorithm (fifo function):

Uses a circular queue (page_queue) to store pages.

Implements a loop to iterate through the page references.

Checks if the page is already in the queue. If not, increments page_faults and adds the page to the queue.

If the queue is full, replaces the front page.

LRU Algorithm (lru function):

Uses an array (page_order) to store the order of pages.

Implements a loop to iterate through the page references.

Checks if the page is already in the array. If not, increments page_faults.

If the array is full, shifts all elements to the left. Adds the new page to the end of the array.

If the page is already in the array, moves it to the end.

Optimal Algorithm (optimal function):

Uses an array (page_order) to store the order of pages.

Implements a loop to iterate through the page references.

Checks if the page is already in the array. If not, increments page_faults.

If the array is full, calculates the future occurrences of each page in the remaining sequence.

Finds the page with the maximum future occurrence index and replaces it.

Main Function (main):

Takes user input for the number of pages, page references, and memory capacity.

Calls the fifo, lru, and optimal functions with the input parameters.

Outputs the number of page faults for each algorithm.

User Input:

The program prompts the user to input the number of pages, page references, and memory capacity.

Output:

The program prints the number of page faults for each algorithm (FIFO, LRU, Optimal).

Improvements:

The code could benefit from better modularization and encapsulation.

Error handling for invalid input (e.g., non-numeric input) is not implemented.

The input validation for the memory capacity might be improved.

Efficiency:

The algorithms have different time complexities. Optimal has the potential to perform the best but is not always feasible in real-world scenarios.

Usage:

The program provides insights into how different page replacement algorithms perform in terms of page faults for a given sequence of page references.

In summary, the code simulates three page replacement algorithms and can be used to compare their performance based on the provided input sequence and memory capacity.

Task –2

The provided code above is C program that demonstrates the Readers-Writers problem using pthreads and semaphores. The code includes two functions, reader and writer, which represent threads that read and write data, respectively.

Analysis of the code:

Header Files:

#include <stdio.h>: Standard input-output functions.

#include <pthread.h>: POSIX threads library for thread creation and synchronization.

#include <semaphore.h>: Semaphore library for synchronization.

Global Variables:

sem_t mutex, writeblock: Semaphores for synchronization.

int data = 0: Shared data that readers and writers access.

int rcount = 0: Reader count to keep track of the number of active readers.

****Reader Function (`void reader(void arg)):**

Accepts an argument (arg) representing the reader's identifier.

Uses semaphores for mutual exclusion and synchronization.

Increments the reader count (rcount) in the critical section.

If it's the first reader, it acquires the writeblock semaphore to prevent writers.

Reads the shared data and prints it.

Sleeps for 1 second to simulate reading.

Decrements the reader count and releases the writeblock semaphore if no readers are left.

****Writer Function (`void writer(void arg)):**

Accepts an argument (arg) representing the writer's identifier.

Uses the writeblock semaphore for mutual exclusion.

Increments the shared data (data) and prints the updated value.

Sleeps for 1 second to simulate writing.

Releases the writeblock semaphore, allowing other writers or readers to access the shared data.

Main Function (`int main()):

Initializes semaphores (mutex and writeblock) using sem_init.

Creates threads for writers (wtid) and readers (rtid) in a loop.

Joins created threads, waiting for them to finish their execution.

Returns 0 to indicate successful program completion.

Thread Creation and Execution:

The program creates three reader and three writer threads in a loop.

Each thread is assigned a unique identifier (i).

Semaphore Initialization:

sem_init(&mutex, 0, 1): Initializes the mutex semaphore with an initial value of 1.

sem_init(&writeblock, 0, 1): Initializes the writeblock semaphore with an initial value of 1.

Sleeping for Simulation:

The sleep(1) statements in both the reader and writer functions simulate some processing time.

Thread Joining:

The pthread_join function is used to wait for the completion of the created threads before the program exits.

Concurrency Control:

The semaphores (mutex and writeblock) are used to control access to the shared data and synchronize the execution of threads.

Improvements:

Error handling for thread creation and semaphore initialization could be added.

The program assumes a fixed number of threads. It could be made more flexible to handle a variable number of readers and writers.

Overall, the program effectively demonstrates the use of semaphores to address the Readers-Writers problem in a multithreaded environment.

Group discussion: