**ASSIGNMENT**

**By**

**Sarthak Dhar**

**2022a1r031**

**3rd**

**C.S.E.**



**Model Institute of Engineering & Technology (Autonomous)**

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with “A” Grade) Jammu, India

2023

|  |
| --- |
|  |

A close-up of a logo

Description automatically generated

INDEX Page No.

1.TASK 1 3-10

2. TASK 2 11-14

3. Report on TASK 1 15

4. Report on TASK 2 16-17

5. Group discussion on our Assignment 18

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;

            }

        }

    }

   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);

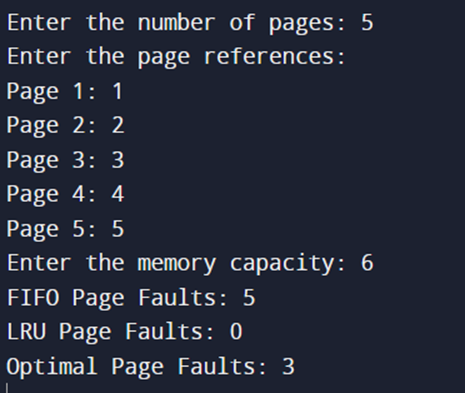
    lru(pages, n, capacity);

   optimal(pages, n, capacity);

    return 0;

}

Output :



**r**

**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;

}

A computer screen with white text

Description automatically generated*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*

### Report on Task 1

**Introduction: This report evaluates the performance of three pivotal page replacement algorithms—FIFO, LRU, and Optimal—integral in managing memory within operating systems. It highlights their efficiency using a specific dataset of page references, showcasing their operational effectiveness.**

**Program Structure: Crafted in C, the program embodies distinct functions dedicated to each algorithm: FIFO, LRU, and Optimal. The main function facilitates user interaction, capturing various page-related parameters, and orchestrates the execution of each algorithm to exhibit resulting page faults.**

**Page Replacement Algorithms Overview:**

1. **FIFO (First-In-First-Out):**
   * **Employs a circular queue to store pages.**
   * **Substitutes the oldest page upon encountering a page fault.**
   * **Renowned for its simplicity and straightforward implementation.**
2. **LRU (Least Recently Used):**
   * **Utilizes an array to maintain the sequence of page references.**
   * **Replaces the least recently used page upon a fault.**
   * **Demands consistent upkeep of the order of pages.**
3. **Optimal:**
   * **Aims to forecast the future by replacing the page with the longest duration until its next usage.**
   * **Acknowledged as the most optimal approach but challenging to practically implement.**

**Results and Analysis: The user-input parameters encompass the quantity of pages, page references, and memory capacity. The program systematically conducts each algorithm, unveiling the resulting page faults for comprehensive analysis.**

**To further enrich this report, exploring the specific advantages and drawbacks of each algorithm could be beneficial. Highlighting how these algorithms perform under diverse scenarios, such as varying memory capacities or different patterns of page references, could offer a more comprehensive understanding. Additionally, discussing the computational complexities or resource implications of each algorithm choice in practical settings would augment the report's value for readers seeking practical insights.**

**Top of Form**

### Report on Task 2

**Overview**: This report delves into a program designed to simulate the classical synchronization challenge, the Reader-Writer problem. This problem scenario involves multiple readers and a solitary writer accessing a shared resource, necessitating synchronization techniques for consistent data handling.

**Program Architecture**: Crafted in C, this program harnesses the pthread library to efficiently manage threads and employs semaphores—mutex and writeblock—as pivotal tools for synchronization. Thread creation for both readers and writers exemplifies their interplay within the system, highlighting synchronization intricacies.

**Synchronization Mechanisms**:

1. **Readers**:
   * Utilization of the mutex semaphore for controlled access to shared resources.
   * Tracking the count of active readers via rcount.
   * Upon entry, increments rcount. The first reader secures the writeblock semaphore, preventing writer access.
   * Post-reading, decrements rcount. The last reader releases the writeblock semaphore, granting writer access.
2. **Writers**:
   * Exclusive resource access facilitated by the writeblock semaphore.
   * Acquisition of the writeblock semaphore before writing ensures singular access.
   * Release of the writeblock semaphore post-writing enables subsequent reader or writer access.

**Execution Flow**: The primary function initializes semaphores and spawns threads for readers and writers. Thread joining guarantees orderly termination. The demonstration includes three threads for both readers and writers, showcasing synchronization behaviors.

**Concluding Remarks**: The program adeptly illustrates synchronization strategies in addressing the complexities of the Reader-Writer problem. It emphasizes the necessity of orchestrated synchronization among multiple readers and a single writer to maintain data integrity within a shared resource environment.

Feel free to elaborate on the challenges faced during implementation or extend the discussion by analyzing performance under varying thread counts or different scenarios for a more comprehensive report.

Top of Form

GROUP Discussion on our Assignment

****

We first delved into the Page Replacement Algorithms program in the group discussion. We examined the structure of the C program, focusing on the FIFO, LRU, and Optimal algorithms. Our conversation revolved around the logic behind each algorithm, user input, and the resulting page fault performance.

Shifting to the Reader-Writer Problem program, we discussed its structure, functions, and the use of semaphores for synchronization. Specifically, we explored how readers and writers are synchronized using semaphores like mutex and writeblock, emphasizing the role of rcount in tracking active readers and preventing race conditions.