OS

ASSIGNMENT

By

NAME - Jatinderpal Singh

ROLL NO. - 2022A1R027

BRANCH - C.S.E

Semester - 3rd

SEC - A2

GROUP - E



Model Institute of Engineering & Technology (Autonomous)

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

Jammu, India 2023

**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;

}

**OUTPUTS:-**

**A computer screen with white text

Description automatically generated**

**A computer screen with white text

Description automatically generated**

**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;

}

***Outputs:-***

**A computer screen with white text

Description automatically generated**

**A computer screen with white text

Description automatically generated**

**A computer screen shot of white text

Description automatically generated**

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

**Reader Processes:**

* **Access pattern**: Readers only read the shared data without modifying it.
* **Synchronization requirements**: Multiple readers can access the shared data concurrently, as their actions do not interfere with each other. However, readers must not access the data while a writer is modifying it, as this could lead to inconsistent data being read.

**Writer Processes:**

* **Access pattern**: Writers modify the shared data.
* **Synchronization requirements**: Writers require exclusive access to the data to prevent data corruption, meaning only one writer can access the data at a time. All readers must wait until the writer has finished modifying the data before they can access it again.

**Common synchronization mechanisms:**

1. **Semaphores**: Data structures that control access to a shared resource by counting the number of processes using it.
2. **Mutexes:** Locks that allow only one process to access a shared resource at a time.
3. **Locks:** Mechanisms that prevent other processes from accessing a shared resource while one process is using it.

**Readers-Writers Problem:**

The readers-writers problem is a classic synchronization problem that arises when multiple reader and writer processes need to access a shared data structure concurrently. The goal is to find a synchronization solution that allows multiple readers to access the data concurrently while ensuring that writers have exclusive access to the data.

REPORT TASK-1

The program serves as a simulation platform for three homepage switching algorithms: FIFO, LRU, and Optimal. These algorithms play an important role in memory management. This report briefly analyzes the performance of each application algorithm, using sample output pages to illustrate its results.

Page replacement algorithm:

1. FIFO (First In, First Out):

- Use a circular queue for page storage.

- Replace the oldest page when a page causing the problem.

- It stands out with its simple and understandable usage.

2. LRU (least used):

- Use an array to control the order of pages to be sent.

- Change the last used page when an error occurs.

- The order page should be maintained regularly.

3. Best for:

- Try to predict the future by changing the page with the longest time until the next event.

- considered the best, although difficult to implement.

Evaluation:

The program systematically executes each algorithm and reveals the offending pages. This method provides insight into comparing algorithms in specific situations.

**REPORT** TASK-2

The program provides simulation of a classic read-write problem, the synchronization problem where multiple readers and one writer access a share. The program uses semaphores for synchronization to ensure data consistency.

**Program structure:**

The program is written in C using the pthread library function for thread management and semaphores for synchronization. Two semaphores, mutexes and write blocks, are used to control access to shared resources. This service creates threads for readers (read operation) and writers (write operation) to represent the interaction between these operations.

**1. Synchronization of Readers:**

Readers use shared semaphores to synchronize access to shared resources.

rcount (reader count) is used to track the number of active readers.

Increases rcount when reader enters. The first reader also takes the script to prevent the writers from accessing the sources.

After reading the data the reader reduces rcount. If this is the final reader, a write block semaphore is released, allowing the writer to access the resource.

**2. Synchronization of Writer:**

Authors use the write function to synchronize access to shared resources.

The author received a draft before writing the document to confirm access to the resource.

The author obtained written permission before collecting information to provide access to certain resources.

After writing, the author publishes the draft, allowing readers or other authors to access the source.

Process:

The main function initializes the semaphore and creates the execution path for readers and writers. Insert threads to ensure proper termination. The examples given constitute our title for readers and writers.

GROUP E:-

Discussion of members of Group E on Readers and Writer Problem, LIFO, Optimal Page Utilization and other topics.

****