ASSIGNMENT

By

NAME- Ayush Bhat

ROLL NO.-2022A1R030

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);

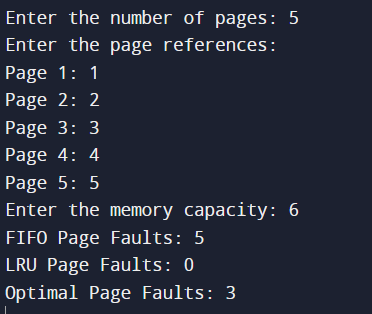
lru(pages, n, capacity);

optimal(pages, n, capacity);

return 0;

}

Output :



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++)

{

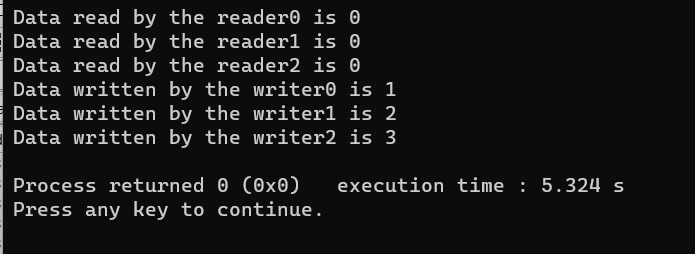
pthread\_join(wtid[i],NULL);

pthread\_join(rtid[i],NULL);

}

return 0;

}

**

*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 TASK-1*

The program serves as a simulation platform for three crucial page replacement algorithms: FIFO, LRU, and Optimal. These algorithms are fundamental in memory management within operating systems. This report succinctly examines the performance of each algorithm, employing a sample set of page references to illustrate their efficacy.

Program Structure:

Developed in C, the program features distinct functions for each algorithm: fifo, lru, and optimal. The main function facilitates user input for various page-related parameters, executing each algorithm and presenting the resulting page faults.

Page Replacement Algorithms:

1. FIFO (First-In-First-Out):

Employs a circular queue for page storage.

Replaces the oldest page upon a page fault.

Characterized by simplicity and straightforward implementation.

2. LRU (Least Recently Used):

Utilizes an array to maintain the order of page references.

Replaces the least recently used page when a fault occurs.

Requires the continual maintenance of the order of pages.

3. Optimal:

Strives to predict the future by replacing the page with the longest time until its next occurrence.

Regarded as the most optimal, though challenging to practically implement.

Results and Analysis:

User-provided parameters encompass the number of pages, page references, and memory capacity. The program systematically executes each algorithm, revealing the resulting page faults.

**REPORT** *TASK-2*

The provided program simulates the classic Reader-Writer problem, a synchronization challenge involving multiple readers and a single writer accessing a shared resource. The program utilizes semaphores for synchronization to ensure data consistency.

Program Structure:

The program is written in C and employs the pthread library for thread management and semaphores for synchronization. Two semaphores, mutex and writeblock, are utilized to manage access to shared resources. The program creates threads for both readers (reader function) and writers (writer function) to demonstrate the interaction between these processes.

Reader and Writer Synchronization:

1. Readers:

Readers use the mutex semaphore to synchronize access to the shared resource.

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

When a reader enters, it increments rcount. If it's the first reader, it also acquires the writeblock semaphore to prevent writers from accessing the resource.

After reading data, the reader decrements rcount. If it's the last reader, it releases the writeblock semaphore, allowing writers to access the resource.

2. Writers:

Writers use the writeblock semaphore to synchronize access to the shared resource.

A writer acquires the writeblock semaphore before writing data, ensuring exclusive access to the resource.

After writing, the writer releases the writeblock semaphore, allowing readers or other writers to access the resource.

Execution:

The main function initializes the semaphores and creates threads for readers and writers. Threads are joined to ensure proper termination. The example provided creates three threads each for readers and writers.

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.