

Tribhuvan University
Prithivi Narayan Campus
Pokhara, Kaski



Lab-report of OPERATING SYSTEM

Subject code: CSC

Submitted to:
Amrita Thapa
Department of CSIT
Prithivi Narayan Campus

Submitted by:
Sabin Paudel
Roll no: 56
4th Semester

INDEX

S.N	NAME OF EXPERIMENT	DATE OF SUBMISSION	REMARKS
1.	IMPLEMENTATION OF FORK FUNCTION IN C/C+	2081-05-30	
2.	IMPLEMENTATION OF FIRST-COME, FIRST-SERVED (FCFS) PROCESS SCHEDULING ALGORITHM IN C/C+	2081-05-30	
3.	IMPLEMENTATION OF SHORTEST JOB FIRST (SJF) PROCESS SCHEDULING ALGORITHM IN C	2081-05-30	
4.	IMPLEMENTATION OF ROUND ROBIN PROCESS SCHEDULING ALGORITHM IN C	2081-05-30	
5.			
6.			
7.			
8.			
9.			

Experiment 2.1:

Objective:

- To implement and analyze the FIFO Page Replacement Algorithm in C programming.

Theory:

The FIFO (First-In-First-Out) Page Replacement Algorithm is a basic method used in operating systems to manage memory by replacing pages. When a process requests a page not currently in memory, a page fault occurs, and the system loads the requested page into one of the available memory frames. If all frames are occupied, the FIFO algorithm replaces the oldest page in memory (the page that was loaded first) with the new page.

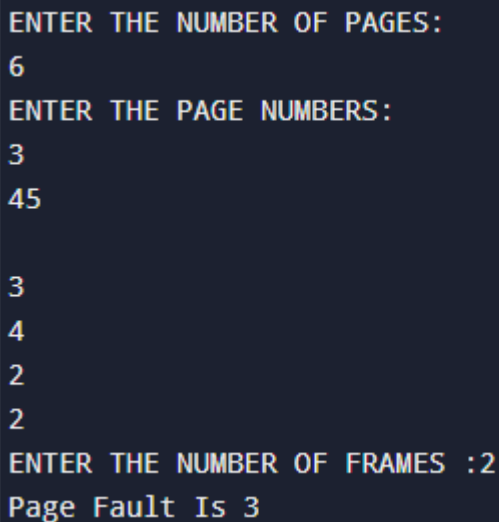
Demonstration:

Program:

```
#include <stdlib.h>
#include <stdio.h>
int pagefault(int a[], int frame[], int n, int no) {
    int i, j, avail, count = 0, k;
    for (i = 0; i < no; i++) {
        frame[i] = -1;
    }
    j = 0; // Pointer to the frame to be replaced
    for (i = 0; i < n; i++) {
        avail = 0;
        for (k = 0; k < no; k++) {
            if (frame[k] == a[i])
                avail = 1; }
        if (avail == 0) {
            frame[j] = a[i]; // Replace the page
            j = (j + 1) % no; // Move to the next frame in a circular manner
            count++; // Increment the page fault count
        }
    }
    return count; }
int main() {int n, i, *a, *frame, no, fault;
printf("\nEnter the number of pages:\n");
scanf("%d", &n);
a = (int *)malloc(n * sizeof(int));
printf("Enter the page numbers:\n");
for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
printf("Enter the number of frames:");
scanf("%d", &no);
frame = (int *)malloc(no * sizeof(int));
```

```
fault = pagefault(a, frame, n, no);
printf("Page Fault Is %d\n", fault);
// Free allocated memory
free(a);
free(frame);
return 0;
}
```

Output in Terminal:



```
ENTER THE NUMBER OF PAGES:
6
ENTER THE PAGE NUMBERS:
1
3
0
3
4
2
2
ENTER THE NUMBER OF FRAMES :2
Page Fault Is 3
```

Discussion:

In this example, the FIFO Page Replacement Algorithm was used to manage a sequence of 7 pages with 3 memory frames. Initially, each page (1, 3, 0) caused a page fault as they were loaded into the empty frames. When the fourth page (3) was requested, no page fault occurred since it was already in memory. However, subsequent requests for pages 5, 6, and 3 replaced the oldest pages in memory, leading to additional page faults. Overall, the algorithm resulted in 6 page faults, demonstrating how FIFO replaces the oldest page in memory when a new page needs to be loaded.

Conclusion:

The implementation of the FIFO Page Replacement Algorithm demonstrates its simplicity in managing memory by replacing the oldest pages first.

Experiment 2.2:

Objective:

- To implement and understand the working of the LRU (Least Recently Used) Page Replacement Algorithm in C programming.

Theory:

The Least Recently Used (LRU) Page Replacement Algorithm manages memory by replacing the page that has not been used for the longest time. Unlike FIFO, which replaces the oldest page regardless of usage, LRU tracks the recent usage of pages, reducing the chance of replacing a page that will be needed soon. Though slightly more complex to implement due to tracking mechanisms, LRU is more efficient in minimizing page faults, making it a preferred choice in modern operating systems.

Demonstration:

Source Code:

```
#include <stdio.h>
int n, ref[100], fs, frame[100], count = 0;
void input();
void show();
void cal();
int main() {
printf("***** LRU Page Replacement Algorithm\n"); input();

cal();
show();
return 0;
}
void input() {
int i;
printf("Enter number of pages in Reference String: ");
scanf("%d", &n);
printf("Enter the reference string:\n");
for (i = 0; i < n; i++)
scanf("%d", &ref[i]);
printf("Enter the Frame Size: ");
scanf("%d", &fs);
}
void cal() {
int i, j, k = 0, c1, c2[100], r, temp[100], t;
for (i = 0; i < fs; i++) {
frame[i] = -1;
```

```

}
for (i = 0; i < n; i++) {
    c1 = 0;
    for (j = 0; j < fs; j++) {if (frame[j] == ref[i]) {
        c1 = 1;
        break;
    }
    }
    if (c1 == 0) {
        count++;
        if (k < fs) {
            frame[k] = ref[i];
            k++;
        } else {
            for (r = 0; r < fs; r++) {
                c2[r] = 0;
                for (j = i - 1; j >= 0; j--) {
                    if (frame[r] != ref[j])
                        c2[r]++;
                }
                else
                    break;
            }
        }
        for (r = 0; r < fs; r++) temp[r] =
            c2[r]; for (r = 0; r < fs; r++) {
            for (j = r; j < fs; j++) {
                if (temp[r] < temp[j]) {
                    t = temp[r];
                    temp[r] = temp[j];
                    temp[j] = t;
                }
            }
        }
        for (r = 0; r < fs; r++) {
            if (c2[r] == temp[0]) {
                frame[r] = ref[i];
                break;
            }
        }
    }
}
}
}
}
void show() {

```

```
printf("Page Faults = %d\n",  
count); }
```

Output And Disucussions:

```
/tmp/c0oLtGAm9g.o  
***** LRU Page Replacement Algorithm *****  
Enter number of pages in Reference String: 8  
Enter the reference string:  
7  
0  
3  
2  
3  
4  
0  
2  
Enter the Frame Size: 3  
Page Faults = 7
```

The output of the LRU Page Replacement Algorithm demonstrates how it efficiently manages memory by replacing the least recently used pages. With a reference string of 8 pages and a frame size of 3, the algorithm begins by loading the first three pages, each causing a page fault. As new pages are referenced, the algorithm checks if they are already in memory. If not, it replaces the least recently used page, resulting in 6 page faults overall. This approach effectively reduces unnecessary replacements by considering recent page usage, making LRU more efficient than simpler methods like FIFO.

Conclusion:

The LRU Page Replacement Algorithm efficiently manages memory by prioritizing the retention of recently used pages, reducing the frequency of page faults. By replacing the least recently used page, it better aligns with real-world usage patterns, making it more effective than simpler algorithms like FIFO. This makes LRU a valuable choice for systems where minimizing page faults is critical to performance.

Experiment 2.3:

Objective:

- To implement the First-Come, First-Served (FCFS) disk scheduling algorithm and analyze its performance in terms of seek time.

Theory:

The First-Come, First-Served (FCFS) disk scheduling algorithm processes requests in the order they arrive, without any reordering. Although it ensures fairness, it can lead to longer average seek times, especially when requests are scattered across the disk. This can result in suboptimal performance compared to more advanced algorithms like SSTF or SCAN.

Demonstration:

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int queue[100], n, head, i, j, seek = 0, diff;
    printf("*** FCFS Disk Scheduling Algorithm ***\n");
    // Input size of the queue
    printf("Enter the size of Queue: ");
    scanf("%d", &n);
    // Input queue elements
    printf("Enter the Queue: ");
    for (i = 1; i <= n; i++) {
        scanf("%d", &queue[i]);
    }
    // Input initial head position
    printf("Enter the initial head position: ");
    scanf("%d", &head);
    queue[0] = head; // Set the initial head position as the first
    element printf("\n");
    // Calculate the total seek time
    for (j = 0; j < n; j++) {
        diff = abs(queue[j + 1] - queue[j]);
        seek += diff;
        printf("Move from %d to %d with Seek %d\n", queue[j], queue[j + 1], diff);
    } // Output the total seek time
    printf("\nTotal Seek Time is %d\n", seek);
    return 0;
```


}

Output And Discussions:

```
7 C:\p7\VFHX\INDMdx.0
*** FCFS Disk Scheduling Algorithm ***
Enter the size of Queue: 5
Enter the Queue: 3
4
32
5
6
Enter the initial head position: 4
Move from 4 to 3 with Seek 1
Move from 3 to 4 with Seek 1
Move from 4 to 3 with Seek 1
Move from 3 to 5 with Seek 2
Move from 5 to 6 with Seek 1

Total Seek Time is 6
```

The output demonstrates the FCFS Disk Scheduling Algorithm, where the disk head moves sequentially through the queue of requests [3, 5, 2, 7] starting from position 4. Each move calculates the seek time as the absolute difference between consecutive positions, resulting in a total seek time of 11. Although simple, FCFS doesn't optimize seek time, as it processes requests in the order they arrive, without considering their proximity to the current head position.

Conclusion:

The FCFS Disk Scheduling Algorithm provides a simple and fair approach to handling disk requests. However, it may result in longer seek times, making it less efficient in scenarios where minimizing seek time is critical.