**Ex. No. : 17**
**Date:**

# PAGE REPLACEMENT ALGORITHM

## a) FIFO

**Problem description:**
The FIFO algorithm follows a simple principle: the page that has been in memory the longest is the one selected for replacement.
It maintains a queue of all the pages currently in memory. When a page needs to be replaced, the page at the front of the queue (oldest page) is chosen.

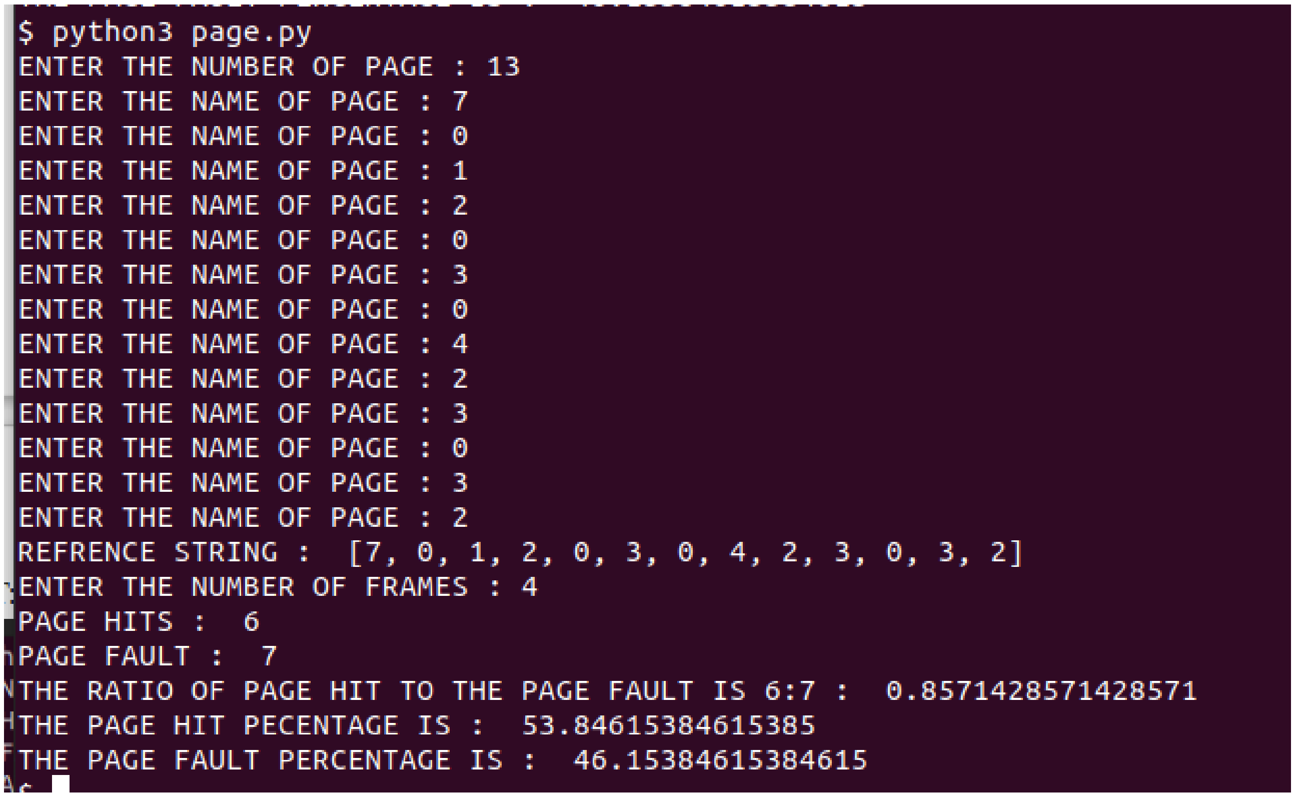
**Algorithm:**

- Create a queue to represent page order in memory.
- Set the queue size to the total number of frames.
- If the page is in the queue, update any relevant information.
- If the page is not in the queue (page fault), proceed to page fault handling.
- If there is an available frame:
- Add the new page to the end of the queue.
- If there are no available frames:
- Remove the page at the front of the queue (oldest page).
- Add the new page to the end of the queue.
- Reflect the current order of pages in memory.
- Repeat steps 2-4 for each page access or until the end of the input sequence.

**Program:**

```
from queue import Queue

def pageFaults(pages, n, capacity):

    s = set()

    indexes = Queue()

    page_faults = 0
    page_hits = 0
    for i in range(n):
        if (len(s) < capacity):
            if (pages[i] not in s):
                s.add(pages[i])
                page_faults += 1
                indexes.put(pages[i])
            else:
                page_hits += 1
        else:
            if (pages[i] not in s):
                val = indexes.queue[0]
                indexes.get()
                s.remove(val)
                s.add(pages[i])
                indexes.put(pages[i])
                page_faults += 1
            else:
                page_hits += 1

    return [page_faults , page_hits]

if __name__ == '__main__':
    pages= []
    no_of_page = int(input(“ENTER THE NUMBER OF PAGES : “))
    for i in range(no_of_page):
        a = int(input(“ENTRT THE REFERENCE STRING OF A PAGE : ”))
        pages.append(a)
    print("REFRENCE STRING : ",pages)
    n = len(pages)
    capacity = int(input(“ENTER THE NUMBER OF FRAMES : ”))
    a = pageFaults(pages, n, capacity)
    print("PAGE HITS : ", a[1])
    print("PAGE FAULT : ", a[0])
    print(f"THE RATIO OF PAGE HIT TO THE PAGE FAULT IS {a[1]}:{a[0]} : ", a[1]/a[0])
    print("THE PAGE HIT PECENTAGE IS : ", a[0] * 100 / n)
    print("THE PAGE FAULT PERCENTAGE IS : ", a[1] * 100 / n)
```

**OUTPUT:**



```
$ python3 page.py
ENTER THE NUMBER OF PAGE : 13
ENTER THE NAME OF PAGE : 7
ENTER THE NAME OF PAGE : 0
ENTER THE NAME OF PAGE : 1
ENTER THE NAME OF PAGE : 2
ENTER THE NAME OF PAGE : 0
ENTER THE NAME OF PAGE : 3
ENTER THE NAME OF PAGE : 0
ENTER THE NAME OF PAGE : 4
ENTER THE NAME OF PAGE : 2
ENTER THE NAME OF PAGE : 3
ENTER THE NAME OF PAGE : 0
ENTER THE NAME OF PAGE : 3
ENTER THE NAME OF PAGE : 2
REFRENCE STRING :  [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
ENTER THE NUMBER OF FRAMES : 4
PAGE HITS :  6
PAGE FAULT :  7
THE RATIO OF PAGE HIT TO THE PAGE FAULT IS 6:7 :  0.8571428571428571
THE PAGE HIT PECENTAGE IS :  53.84615384615385
THE PAGE FAULT PERCENTAGE IS :  46.15384615384615
```

**Result:**

Thus, FIFO Page Replacement Algorithm was implemented and executed successfully.

**b) LRU (Least Recently Used Algorithm)**

**Problem Description:**

The problem is to develop a page replacement algorithm, specifically the Least Recently Used (LRU) algorithm, to efficiently manage a fixed-size memory (cache) in a computer system. The system operates with virtual memory, and when a page is needed that is not currently in the cache, it results in a page fault. The goal is to minimize the number of page faults by determining which page to evict from the cache when it is full.

**Least Recently Used Algorithm:**

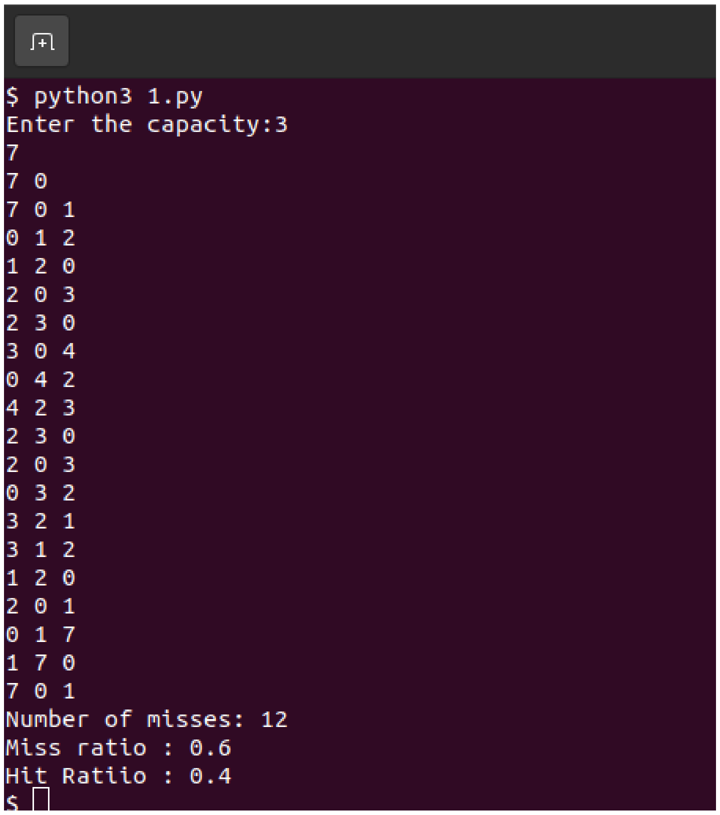
- Let capacity be the number of pages that memory can hold.
- Let set be the current set of pages in memory.
    1. Start traversing the pages.
        a. If the set holds less pages than capacity.
            i. Insert pages into the set one by one until the size of the set reaches capacity or all page requests are processed.
            ii. Simultaneously maintain the recently occurred index of each page in a map called indexes.
            iii. Increment page fault
        b. If the current page is present in the set, do nothing.
        c. Else
            i. Find the page in the set that was least recently used. We find it using an index array.
            ii. We basically need to replace the page with a minimum index.
            iii. Replace the found page with the current page.
            iv. Increment page faults.
            v. Update index of current page.
    2. Return page faults

**Code:**

```
capacity = int(input("Enter the capacity:"))
'''
n = int(input('Size:'))
processList = []
for i in range(n):
   val = int(input("Num:"))
   processList.append(val)
```

```
'''
processList = [7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1]
s = []
pageFaults = 0
for i in processList:
    if i not in s:
            if(len(s) == capacity):
                            s.remove(s[0])
                            s.append(i)
            else:
                    s.append(i)
            pageFaults +=1
    else:
                    s.remove(i)
                    s.append(i)
    for j in s:
            print(j,end=" ")
    print()
ratio = pageFaults/(len(processList))
print('Number of misses:', pageFaults)
print('Miss ratio :',ratio)
print('Hit Ratiio :', 1-ratio)
```

**Output:**



```
$ python3 1.py
Enter the capacity:3
7
7 0
7 0 1
0 1 2
1 2 0
2 0 3
2 3 0
3 0 4
0 4 2
4 2 3
2 3 0
2 0 3
0 3 2
3 2 1
3 1 2
1 2 0
2 0 1
0 1 7
1 7 0
7 0 1
Number of misses: 12
Miss ratio : 0.6
Hit Ratiio : 0.4
$ 
```

**Result:**

Thus the LRU page replacement algorithm minimizes page faults by evicting the least recently used page when the cache is full.

**c) Optimal**

**Problem description:**
Optimal Page Replacement aims to minimize page faults in a memory management system. Given a memory capacity and a sequence of page references, the algorithm maintains a set of pages currently in memory. When a page reference is encountered:

a. If the page is already in memory, no action is taken.
b. If the page is not in memory:
i. If the memory set is not full, the page is added, and a page fault is recorded.
ii. If the memory set is full, the algorithm replaces the page that will be accessed furthest in the future, minimizing page faults.

The algorithm requires knowledge of future page accesses, which is often not practical in real-time scenarios.

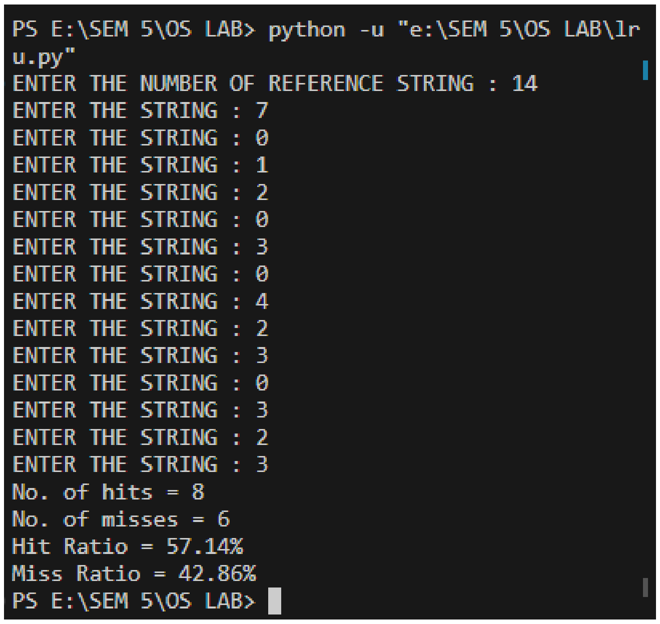
**Algorithm:**

```
Let capacity be the number of pages that memory can hold.
Let frames be the current set of pages in memory.
Start traversing the page references.
a. If frames hold fewer pages than capacity.
        i. Insert pages into frames one by one until the size of frames reaches capacity or all page
        requests are processed.
        ii. Simultaneously maintain a map called indexes to keep track of the most recent
        occurrence index of each page.
        iii. Increment page faults.
b. If the current page is present in frames, do nothing.
c. Else:
        i. Find the page in frames that will be least recently used by using the indexes map.
        ii. Replace the found page with the current page.
        iii. Increment page faults.
        iv. Update the index of the current page.
Return the total number of page faults.
```

**CODE:**

```
def exists_in_frame(key, frame):
    for i in range(len(frame)):
        if frame[i] == key:
            return True
    return False

def find_least_recently_used(pg, frame, pn, index):
    result = -1
    farthest = index
    for i in range(len(frame)):
        j = 0
        for j in range(index, pn):
            if frame[i] == pg[j]:
                if j > farthest:
                    farthest = j
                    result = i
                break
        if j == pn - 1:
            return i
    return 0 if result == -1 else result

def optimal_page(pg, pn, frame_count):
    frames = []
    hits = 0
    for i in range(pn):
        if exists_in_frame(pg[i], frames):
            hits += 1
            continue
        if len(frames) < frame_count:
            frames.append(pg[i])
        else:
            j = find_least_recently_used(pg, frames, pn, i + 1)
            frames[j] = pg[i]
    misses = pn - hits
    hit_ratio = (hits / pn) * 100
    miss_ratio = (misses / pn) * 100
    print("No. of hits =", hits)
    print("No. of misses =", misses)
    print("Hit Ratio =", "{:.2f}%".format(hit_ratio))
    print("Miss Ratio =", "{:.2f}%".format(miss_ratio))

# Driver Code
page_references = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3]
page_count = len(page_references)
frame_number = 4
optimal_page(page_references, page_count, frame_number)
```

OUTPUT :



```
PS E:\SEM 5\OS LAB> python -u "e:\SEM 5\OS LAB\lr
u.py"
ENTER THE NUMBER OF REFERENCE STRING : 14
ENTER THE STRING : 7
ENTER THE STRING : 0
ENTER THE STRING : 1
ENTER THE STRING : 2
ENTER THE STRING : 0
ENTER THE STRING : 3
ENTER THE STRING : 0
ENTER THE STRING : 4
ENTER THE STRING : 2
ENTER THE STRING : 3
ENTER THE STRING : 0
ENTER THE STRING : 3
ENTER THE STRING : 2
ENTER THE STRING : 3
No. of hits = 8
No. of misses = 6
Hit Ratio = 57.14%
Miss Ratio = 42.86%
PS E:\SEM 5\OS LAB>
```

**Result:**

Thus the Optimal page replacement algorithm minimizes page faults by evicting the least recently used page when the cache is full.