A.Y.: 2022-23 **Class: S.Y.B.Tech** **Sub:** System Fundamentals

Experiment 8

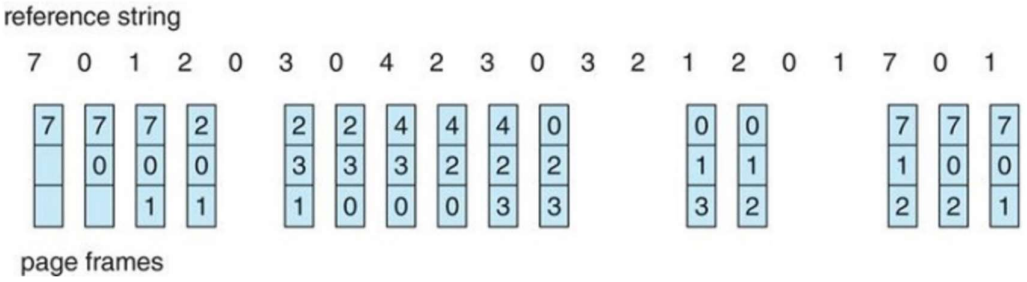
(Cache / Page replacement policies)

Aim: Implement various cache/page replacement policies
(i) FIFO (ii) OPTIMAL (iii) LRU

Theory:

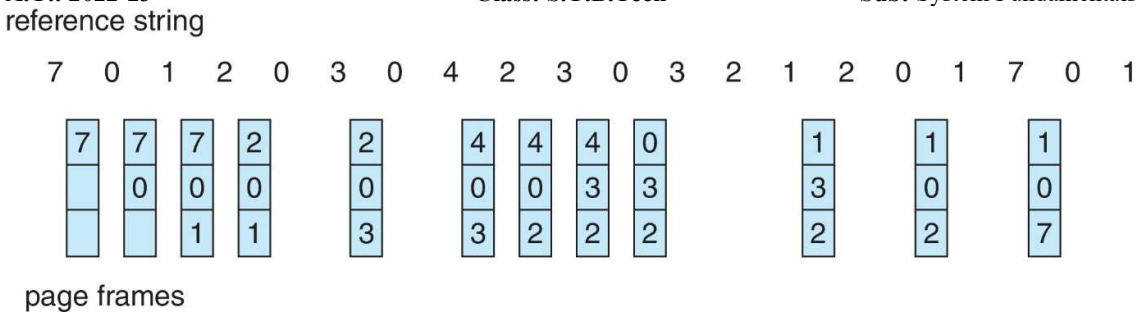
First-in, first-out

The simplest page-replacement algorithm is a FIFO algorithm. The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little book-keeping on the part of the operating system. The idea is obvious from the name – the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences Bélády's anomaly. FIFO page replacement algorithm is used by the VAX/VMS operating system, with some modifications.



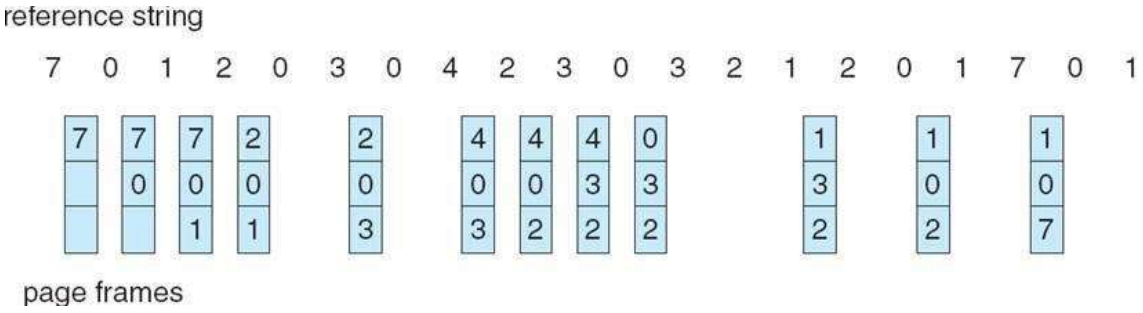
OPT

when a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future. For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds.
This algorithm cannot be implemented in the general purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used, except when all software that will run on a system is either known beforehand and is amenable to the static analysis of its memory reference patterns, or only a class of applications allowing run-time analysis. Despite this limitation, algorithms exist[citation needed] that can offer near-optimal performance — the operating system keeps track of all pages referenced by the program, and it uses those data to decide which pages to swap in and out on subsequent runs. This algorithm can offer near-optimal performance, but not on the first run of a program, and only if the program's memory reference pattern is relatively consistent each time it runs.
Analysis of the paging problem has also been done in the field of online algorithms. Efficiency of randomized online algorithms for the paging problem is measured using amortized analysis.



LRU

The least recently used page (LRU) replacement algorithm keeps track of page usage over a short period of time. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory, it is rather expensive to implement in practice. There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.



Lab Assignments to complete in this session

1. Consider page reference string 1, 3, 0, 3, 5, 6 with 3-page frames. Find number of page faults using FIFO find hit ratio.

```
[21] m=int(input("Enter no. of frames:"))
     n=int(input("Enter no. of pages to be entered:"))
     page=[]
     for i in range(n):
         x=int(input("Enter page:"))
         page.append(x)
```

```
Enter no. of frames:3
Enter no. of pages to be entered:6
Enter page:1
Enter page:3
Enter page:0
Enter page:3
Enter page:5
Enter page:6
```

```
[22] def fifo(pages, m):
         cache = []
         faults = 0
         hits = 0

         for page in pages:
             if page in cache:
                 hits += 1
             else:
                 if len(cache) == n:
                     cache.pop(0)
                 cache.append(page)
                 faults += 1

         hit_ratio = hits / len(pages)
         return faults, hit_ratio


     faults, hit_ratio = fifo(page, m)

     print(f"Number of page faults: {faults}")
     print("Number of hits:",n-faults)
     print(f"Hit ratio: {hit_ratio}")
```

```
Number of page faults: 5
Number of hits: 1
Hit ratio: 0.16666666666666666
```

2. Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4-page frame. Find number of page fault using LRU.

```
[32] m=int(input("Enter no. of frames:"))
     page=[7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]

     Enter no. of frames:4
```

```
def lru(pages, m):
    cache = []
    faults = 0
    hits = 0

    for page in pages:
        if page in cache:
            hits += 1
            cache.remove(page)
        else:
            if len(cache) == n:
                cache.pop(0)
            faults += 1
        cache.append(page)

    hit_ratio = hits / len(pages)
    return faults, hit_ratio

faults, hit_ratio = lru(page, m)

print(f"Number of page faults: {faults}")
print("Number of hits:",len(page)-faults)
print(f"Hit ratio: {hit_ratio}")
```

```
Number of page faults: 6
Number of hits: 7
Hit ratio: 0.5384615384615384
```

3. Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4-page frame. Find number of page fault using Optimal.

```
[33] m=int(input("Enter no. of frames:"))
     page=[7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]


     Enter no. of frames:4
```

```
def optimal_page_faults(pages, m):
    cache = []
    faults = 0
    hits = 0

    for i, page in enumerate(pages):
        if page in cache:
            hits += 1
        else:
            if len(cache) == n:
                # find the page that will not be used for the longest time
                future = {p: pages[i+1:].index(p) if p in pages[i+1:] else n for p in cache}
                page_to_remove = max(future, key=future.get)
                cache.remove(page_to_remove)
            cache.append(page)
            faults += 1

    hit_ratio = hits / len(pages)
    return faults, hit_ratio


faults, hit_ratio = optimal_page_faults(page, m)

print(f"Number of page faults: {faults}")
print("Number of hits:",len(page)-faults)
print(f"Hit ratio: {hit_ratio}")
```

```
Number of page faults: 6
Number of hits: 7
Hit ratio: 0.5384615384615384
```