OPERATING SYSTEM IMPORTANT TOPICS:

UNIT1:

1. Types of operating systems?
2. System calls
3. Services of operating system

UNIT2:

1. File attributes
2. File access methods
3. Directory Structure of file
4. Disk scheduling algorithm

UNIT3:

1. PCB

2. Threads

3. Context switching

4. Scheduling algorithms

UNIT4:

1. Paging
2. Virtual memory
3. Page replacement algorithm

UNIT5:

1. Critical section

2. Semaphore

3. Monitor

4. Producer Consumer problems

UNIT6:

1. Bankers algorithm
2. Resource allocation graph and DEADLOCK

**LRU**

import java.util.ArrayList;

public class LRU {

    // Driver method

    public static void main(String[] args) {

        int capacity = 4;

        int arr[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

        // To represent set of current pages.We use

        // an Arraylist

        ArrayList<Integer> s=new ArrayList<>(capacity);

        int count=0;

        int page\_faults=0;

        for(int i:arr)

        {

            // Insert it into set if not present

            // already which represents page fault

            if(!s.contains(i))

            {

            // Check if the set can hold equal pages

            if(s.size()==capacity)

            {

                s.remove(0);

                s.add(capacity-1,i);

            }

            else

                s.add(count,i);

                // Increment page faults

                page\_faults++;

                ++count;

            }

            else

            {

                // Remove the indexes page

                s.remove((Object)i);

                // insert the current page

                s.add(s.size(),i);

            }

        }

        System.out.println(page\_faults);

    }

}

**OPTIMAL**

#include <bits/stdc++.h>

using namespace std;

// Function to check whether a page exists

// in a frame or not

bool search(int key, vector<int>& fr)

{

for (int i = 0; i < fr.size(); i++)

if (fr[i] == key)

return true;

return false;

}

// Function to find the frame that will not be used

// recently in future after given index in pg[0..pn-1]

int predict(int pg[], vector<int>& fr, int pn, int index)

{

// Store the index of pages which are going

// to be used recently in future

int res = -1, farthest = index;

for (int i = 0; i < fr.size(); i++) {

int j;

for (j = index; j < pn; j++) {

if (fr[i] == pg[j]) {

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

// If a page is never referenced in future,

// return it.

if (j == pn)

return i;

}

// If all of the frames were not in future,

// return any of them, we return 0. Otherwise

// we return res.

return (res == -1) ? 0 : res;

}

void optimalPage(int pg[], int pn, int fn)

{

// Create an array for given number of

// frames and initialize it as empty.

vector<int> fr;

// Traverse through page reference array

// and check for miss and hit.

int hit = 0;

for (int i = 0; i < pn; i++) {

// Page found in a frame : HIT

if (search(pg[i], fr)) {

hit++;

continue;

}

// Page not found in a frame : MISS

// If there is space available in frames.

if (fr.size() < fn)

fr.push\_back(pg[i]);

// Find the page to be replaced.

else {

int j = predict(pg, fr, pn, i + 1);

fr[j] = pg[i];

}

}

cout << "No. of hits = " << hit << endl;

cout << "No. of misses = " << pn - hit << endl;

}

// Driver Function

int main()

{

int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };

int pn = sizeof(pg) / sizeof(pg[0]);

int fn = 4;

optimalPage(pg, pn, fn);

return 0;

}

**FIFO**

import java.util.HashSet;

import java.util.LinkedList;

import java.util.Queue;

class Test

{

    // Method to find page faults using FIFO

    static int pageFaults(int pages[], int n, int capacity)

    {

        // To represent set of current pages. We use

        // an unordered\_set so that we quickly check

        // if a page is present in set or not

        HashSet<Integer> s = new HashSet<>(capacity);

        // To store the pages in FIFO manner

        Queue<Integer> indexes = new LinkedList<>() ;

        // Start from initial page

        int page\_faults = 0;

        for (int i=0; i<n; i++)

        {

            // Check if the set can hold more pages

            if (s.size() < capacity)

            {

                // Insert it into set if not present

                // already which represents page fault

                if (!s.contains(pages[i]))

                {

                    s.add(pages[i]);

                    // increment page fault

                    page\_faults++;

                    // Push the current page into the queue

                    indexes.add(pages[i]);

                }

            }

            // If the set is full then need to perform FIFO

            // i.e. remove the first page of the queue from

            // set and queue both and insert the current page

            else

            {

                // Check if current page is not already

                // present in the set

                if (!s.contains(pages[i]))

                {

                    //Pop the first page from the queue

                    int val = indexes.peek();

                    indexes.poll();

                    // Remove the indexes page

                    s.remove(val);

                    // insert the current page

                    s.add(pages[i]);

                    // push the current page into

                    // the queue

                    indexes.add(pages[i]);

                    // Increment page faults

                    page\_faults++;

                }

            }

        }

        return page\_faults;

    }

    // Driver method

    public static void main(String args[])

    {

        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,

                        2, 3, 0, 3, 2};

        int capacity = 4;

        System.out.println(pageFaults(pages, pages.length, capacity));

    }

}

DISK SCHEDULING ALGORITHMS

* **Seek Time:**Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
* **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
* **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
* **Disk Access Time:** Disk Access Time is:

Disk Access Time = Seek Time +

Rotational Latency +

Transfer Time