**Paging Simulation Report**

**Overview**

This program simulates a paging system using the Aging Algorithm for page replacement. It takes a sequence of page references from a file and simulates how pages are loaded into memory frames. The program measures the number of page faults for different frame sizes and provides output accordingly.

**Key Variables**

* numFrames: The number of available page frames in memory.
* frames: A vector that holds the current pages loaded in memory.
* pageMap: A mapping between page IDs and their index in the frame list.
* age: An 8-bit aging counter used to determine which page should be replaced.
* pageFaults: A counter that keeps track of the number of page faults encountered.

**Data Structures**

* **Page Class**: Represents a page in memory with an ID and an aging counter.
* **AgingAlgorithm Class**: Implements the Aging Algorithm for page replacement.

**Algorithm Description**

**Reading the Input File**

* The program reads a sequence of page references from a file.
* These references represent requests made by processes for specific pages.

**First Loop: Updating Aging Counters**

* Each page's age counter is right-shifted (>> 1) to reduce its priority.
* If a page is accessed, its most significant bit (MSB) is set to 1.

**Second Loop: Checking and Handling Page Requests**

* If a page is already in memory, its age is updated.
* If a page is not in memory, a page fault occurs, and the least recently used page is replaced based on the aging counter.
* The replaced page’s ID is removed from pageMap, and the new page is inserted with the highest priority.

**Deadlock Prevention**

* Since this is a paging simulation, deadlocks do not occur as pages are replaced dynamically.
* The algorithm ensures that even if frames are full, an older page is replaced to accommodate new requests.

**Output and Results**

* The program runs the simulation for different frame sizes (e.g., 2, 4, 8, 16).
* It prints the number of page faults for each frame size, helping analyze performance.

**Main Function Execution**

1. Reads the page reference sequence from a file.
2. Iterates over different frame sizes.
3. Runs the Aging Algorithm for each frame size and tracks page faults.
4. Outputs the number of page faults for each scenario.

**Conclusion**

This paging simulation effectively demonstrates how the Aging Algorithm manages memory by replacing pages based on an aging mechanism. The program highlights the impact of varying frame sizes on page faults, helping understand memory performance in a paging system.

Source Code

#include <iostream>

#include <fstream>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <limits>

using namespace std;

// Class representing a single page

class Page {

public:

int id;

unsigned int age;

// Constructor initializing the page with an ID and resetting the age

Page(int id = -1) {

this->id = id;

this->age = 0;

}

};

// Class implementing the aging page replacement algorithm

class AgingAlgorithm {

private:

int numFrames; // Number of available frames

vector<Page> frames; // List of frames storing pages

unordered\_map<int, int> pageMap; // Mapping of page ID to frame index

public:

// Constructor initializing the algorithm with a given number of frames

AgingAlgorithm(int framesCount) : numFrames(framesCount), frames(framesCount) {}

// Checks if a page is already in memory

bool pageExists(int pageId) {

return pageMap.find(pageId) != pageMap.end();

}

// Updates the aging counter for each page by right-shifting its age

void updateAges() {

for (Page& frame : frames) {

frame.age >>= 1; // Shift age to the right, decreasing its priority

}

}

// Inserts a page into memory using the aging algorithm

void insertPage(int pageId, int& pageFaults) {

updateAges();

if (pageExists(pageId)) {

// If the page is already in memory, set the most significant bit

frames[pageMap[pageId]].age |= (1 << 7);

}

else {

// Page fault occurs since the page is not in memory

pageFaults++;

int victimIndex = 0;

// Find the page with the lowest age (least recently used)

for (int i = 1; i < numFrames; i++) {

if (frames[i].age < frames[victimIndex].age) {

victimIndex = i;

}

}

// Replace the victim page with the new page

pageMap.erase(frames[victimIndex].id);

frames[victimIndex] = Page(pageId);

frames[victimIndex].age = (1 << 7); // Assign highest priority to the new page

pageMap[pageId] = victimIndex;

}

}

};

// Reads page references from a file

void readPagesFromFile(const string& filename, vector<int>& pageReferences) {

ifstream file(filename);

if (!file) {

cerr << "Error opening file: " << filename << endl;

exit(1);

}

int pageId;

while (file >> pageId) {

pageReferences.push\_back(pageId);

}

file.close();

}

int main() {

string filename = "C:\\Users\\salla\\OneDrive\\Desktop\\Paging\_Sim\\paging\_text.txt";

vector<int> pageReferences;

readPagesFromFile(filename, pageReferences);

// Define different frame sizes to test the algorithm

vector<int> framesSizes = { 2, 4, 8, 16 };

// Simulate the aging algorithm for each frame size

for (int frames : framesSizes) {

AgingAlgorithm aging(frames);

int pageFaults = 0;

// Process each page reference

for (int pageId : pageReferences) {

aging.insertPage(pageId, pageFaults);

}

cout << "Page Faults with " << frames << " frames: " << pageFaults << endl;

}

return 0;

}