NFU Page Replacement Algorithm
Project Report
CSE 325

Submitted To

MD. MAHIR ASHHAB

LECTURER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING,

EAST WEST UNIVERSITY

Submitted By

Tasmim Shajahan Mim

2021-3-60-078

DATE : 23/12/2023

Project description :

The provided C++ code implements the Not Frequently Used (NFU) page replacement algorithm in an interactive simulation. It models a memory system with multiple processes, each having a set of pages, and employs NFU logic to determine which pages to replace based on their reference history during simulated iterations.

Introduction :

The NFU (Not Frequently Used) algorithm is a page replacement algorithm used in operating systems to manage memory. It employs a simple and intuitive approach by maintaining a counter for each page, indicating the number of times the page has been referenced. When a page needs to be replaced due to a page fault, the algorithm selects the page with the lowest reference count, assuming that less frequently used pages are less likely to be accessed in the near future. The NFU algorithm is easy to implement but may exhibit suboptimal performance in certain scenarios, such as when a page is initially heavily referenced but becomes less important over time.

DSA :

1) Arrays/Vectors: The code utilizes vectors to represent the memory and pages, allowing for dynamic storage of process pages and their associated information.

2) Bitset: The std::bitset is used to represent the reference bits for each page. It simplifies the manipulation and storage of individual bits, especially in the context of tracking page references.
3) Random Number Generation: The rand() function is used for generating random numbers, particularly in initializing the reference bits and simulating page references.

While the code does not explicitly leverage complex data structures or sophisticated algorithms, it focuses on simulating the NFU page replacement algorithm and involves basic data structures and randomization techniques.

Features:

Variables:

The code utilizes several variables to manage the simulation. numProcesses represents the number of processes in the system, numPagesPerProcess denotes the number of pages each process has, and memorySize defines the total capacity of the memory in terms of the number of pages it can store. The memory variable is a vector representing the memory itself. Each element of this vector is a structure (Page) that stores information about a page, including its process ID, page ID, and a std::bitset<16> called referenceBits that keeps track of the page's reference history.

Features and Functions:

The code defines a structure called Page to encapsulate information related to a page, such as its process ID, page ID, and reference bits. The code includes initialization functions like initializeMemory() and initializeReferenceBits() to set up the initial state of the memory. There is a function named simulatePageReferences() responsible for simulating page references by updating the reference bits with random values. Additionally, there are functions related to the page replacement algorithm, including findLFUIndex() to find the index of the least frequently used page, updateReferenceBits() to simulate aging by shifting reference bits, and replaceLFUPage() to replace the least frequently used page with a new one. The displayMemoryStatus() function is used to print the current state of the memory, showing process and page information along with their reference bits.

Random Number Generation:

The code employs the rand() function for random number generation. It is used to initialize reference bits with random values and simulate page references by generating random bit positions.

User Interaction:

The code interacts with the user through standard input and output streams (cin and cout). The main loop, implemented in the runAlgorithm() function, executes the simulation iteratively, prompting the user to press 'q' to quit or any other key to simulate the next iteration.

Visual Representation of NFU Algorithm :

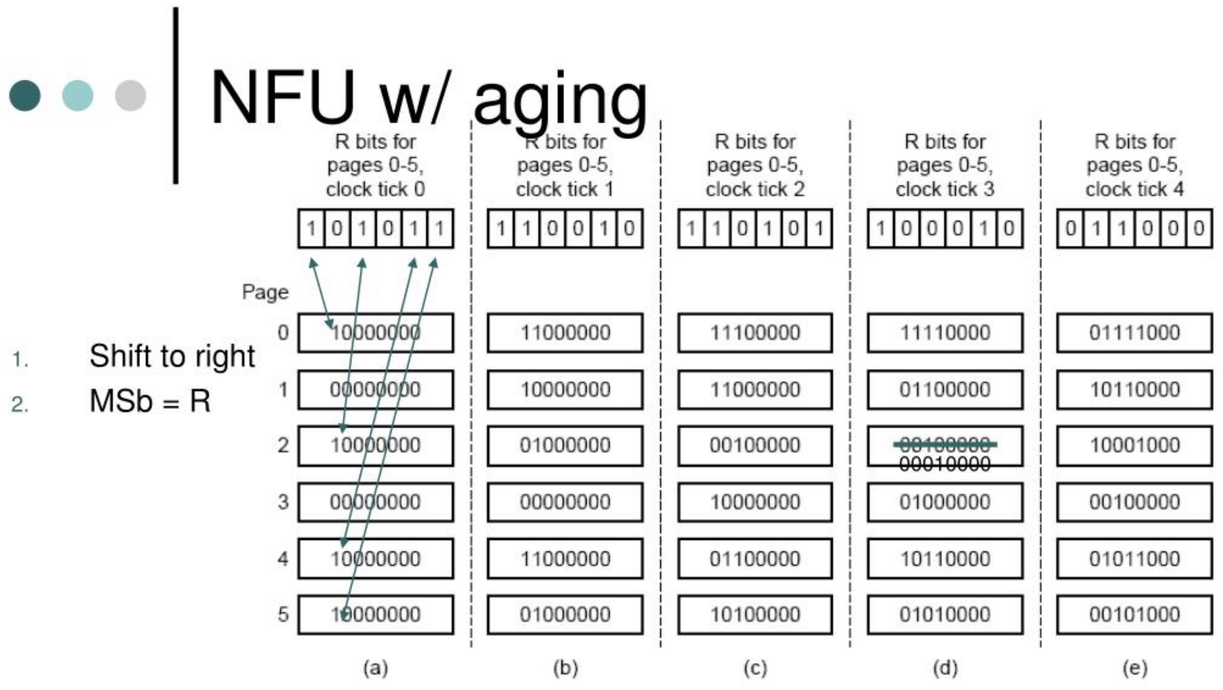


Fig. 4-19. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Source Code :

```
#include <iostream>
#include <vector>
#include <bitset>
#include <cstdlib>
#include <ctime>
#include <climits>
```

```
#include <limits>

using namespace std;

class nfuAlgorithm {
public:
                  nfuAlgorithm(int    numProcesses,    int
numPagesPerProcess, int memorySize)
        : numProcesses(numProcesses),
          numPagesPerProcess(numPagesPerProcess),
          memorySize(memorySize) {
        srand(static_cast<unsigned>(time(0)));
        initializeMemory();
        initializeReferenceBits();
    }

    struct Page {
        int processID;
        int pageID;
        bitset<16> referenceBits;
        int referenceCounter;

        Page(int process, int page)
                     :  processID(process),  pageID(page),
referenceBits(0), referenceCounter(0) {}
    };

    int numProcesses;
    int numPagesPerProcess;
    int memorySize;
    vector<Page> memory;
    void initializeMemory() {
        for (int i = 0; i < memorySize; ++i) {
            memory.push_back(Page(-1, -1));
        }
    }

    void initializeReferenceBits() {
                for (int i = 0; i < numProcesses *
numPagesPerProcess; ++i) {
            int numBitsToSet = rand() % 16 + 1;
            for (int j = 0; j < numBitsToSet; ++j) {
                int randomBit = rand() % 16;
                                             memory[i /
numPagesPerProcess].referenceBits.set(randomBit, 1);
            }
        }
    }

    void simulatePageReferences() {
        for (int i = 0; i < numProcesses; ++i) {
             for (int j = 0; j < numPagesPerProcess; ++j)
{
                 int numBitsToSet = rand() % (memorySize +
1);
                for (int k = 0; k < numBitsToSet; ++k) {
                    int randomBit = rand() % 16;

memory[i].referenceBits.set(randomBit, 1);
                }
                memory[i].referenceCounter++;
            }
        }
    }

    int findLFUIndex() const {
        int minCounter = INT_MAX;
        int lfuIndex = -1;

        for (int i = 0; i < memorySize; ++i) {
            int counter = memory[i].referenceCounter;
            if (counter < minCounter) {
                minCounter = counter;
                lfuIndex = i;
            }
        }

        return lfuIndex;
    }

    void updateReferenceBits() {
        for (int i = 0; i < memorySize; ++i) {
            memory[i].referenceBits >>= 1;
        }
    }

     void replaceLFUPage(int newProcessID, int newPageID,
int lfuIndex) {
        memory[lfuIndex].referenceCounter = 0;
        memory[lfuIndex] = Page(newProcessID, newPageID);
    }

    void displayMemoryStatus() const {
        cout << "Memory Status:\n";
        for (int i = 0; i < memorySize; ++i) {
            cout << "Page " << i << ": ";
            if (memory[i].processID != -1) {
                cout << "Process " << memory[i].processID
                     << ", Page " << memory[i].pageID;
            } else {
                cout << "Empty";
            }
            cout << ", Reference Bits: ";
            for (int j = 0; j < 16; ++j) {
                cout << memory[i].referenceBits[j];
            }
                           cout   <<   ",   Counter:   "   <<
memory[i].referenceCounter << "\n";
        }
        cout << "---------------------\n";
    }

    void runAlgorithm() {
        char userInput;
        do {
            simulatePageReferences();
            updateReferenceBits();
            int lfuIndex = findLFUIndex();
            int newProcessID = rand() % numProcesses;
            int newPageID = rand() % numPagesPerProcess;
                  replaceLFUPage(newProcessID,  newPageID,
lfuIndex);
            displayMemoryStatus();
              cout << "Press 'q' to quit or any other key
to simulate the next iteration: ";
            cin >> userInput;
```

```
cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
            cin.clear();
        } while (userInput != 'q');
    }
};

int main() {
    int numProcesses, numPagesPerProcess, memorySize;
    cout << "Enter the number of processes: ";
    cin >> numProcesses;
    cout << "Enter the number of pages per process: ";
    cin >> numPagesPerProcess;
    cout << "Enter the memory size: ";
    cin >> memorySize;

                nfuAlgorithm    nfuAlgorithm(numProcesses,
numPagesPerProcess, memorySize);
    nfuAlgorithm.runAlgorithm();

    return 0;
}
```

<u>Code Processing :</u>

The C++ code implements the Not Frequently Used (NFU) page replacement algorithm in an interactive simulation. The nfuAlgorithm class represents the algorithm, and it includes functions for initializing memory, simulating page references, finding the least frequently used page, updating reference bits, and replacing pages accordingly. The simulation occurs in a loop, prompting the user to press 'q' to quit or any other key to simulate the next iteration. During each iteration, the code generates random page references, updates reference bits, identifies the least frequently used page, replaces it with a new page, and displays the current memory status. User input and output are facilitated through standard input and output streams. The algorithm's logic involves tracking reference history using bitsets and selecting pages based on their least frequently used status. The program provides a visual representation of the memory status, allowing users to observe the behavior of the NFU algorithm over multiple iterations.

Code Explanation :

Header Files:

The code begins by including necessary header files. It uses <iostream> for input and output operations, <vector> to work with dynamic arrays, <bitset> for efficient handling of binary information, <cstdlib> and <ctime> for random number generation, and <climits> for using the constant INT_MAX. Additionally, <limits> is included for handling input buffer clearing.

Main Function:

In the main function, the user is prompted to input the number of processes, the number of pages per process, and the size of the memory. An instance of the nfuAlgorithm class is then created with these user inputs. The runAlgorithm function is called on this instance, initiating the simulation. The simulation iterates until the user decides to quit by pressing 'q'. For each iteration, the code simulates page references, updates reference bits, identifies the least frequently used page, replaces it with a new page, and displays the current memory status. The user is prompted to continue the simulation or exit.

nfuAlgorithm Class - Initialization Functions:

The nfuAlgorithm class includes a constructor that initializes essential variables such as the number of processes, pages per process, and memory size. It also initializes the random number generator seed and calls two functions: initializeMemory to set up the initial state of the memory, and initializeReferenceBits to randomly assign reference bits for each page.

nfuAlgorithm Class - Memory Initialization:

The initializeMemory function populates the memory vector with instances of the Page structure, representing each page in the simulated memory. Each page is initially set to be empty (processID and pageID are -1).

nfuAlgorithm Class - Reference Bits Initialization:

The initializeReferenceBits function initializes the reference bits for each page. It generates a random number of bits to set for each page, and random positions within the 16-bit reference bits are chosen to be set to 1, simulating the initial reference history.

nfuAlgorithm Class - Page Reference Simulation:

The simulatePageReferences function simulates page references for each iteration of the algorithm. It randomly selects a number of bits to set for each page and updates the reference bits accordingly, mimicking the referencing behavior of pages in a real-world scenario.

nfuAlgorithm Class - Find LFU Index:

The findLFUIndex function identifies the index of the least frequently used page in the memory. It iterates through the pages, counts the number of set bits in the reference bits, and returns the index of the page with the minimum count.

nfuAlgorithm Class - Update Reference Bits:

The updateReferenceBits function simulates the aging of pages by right-shifting their reference bits. This operation reflects the decrease in significance of past references over time.

nfuAlgorithm Class - Counter mechanism :

The counter in the code keeps track of how often each page is referenced during the simulation. For each page, the referenceCounter is incremented in the simulatePageReferences function, reflecting the number of references the page has received. After selecting a page for replacement, its counter is reset to zero in the replaceLFUPage function.

nfuAlgorithm Class - Replace LFU Page:

The replaceLFUPage function replaces the least frequently used page with a new page. It takes the process ID and page ID of the new page, finds the LFU index using findLFUIndex, and updates the memory accordingly.

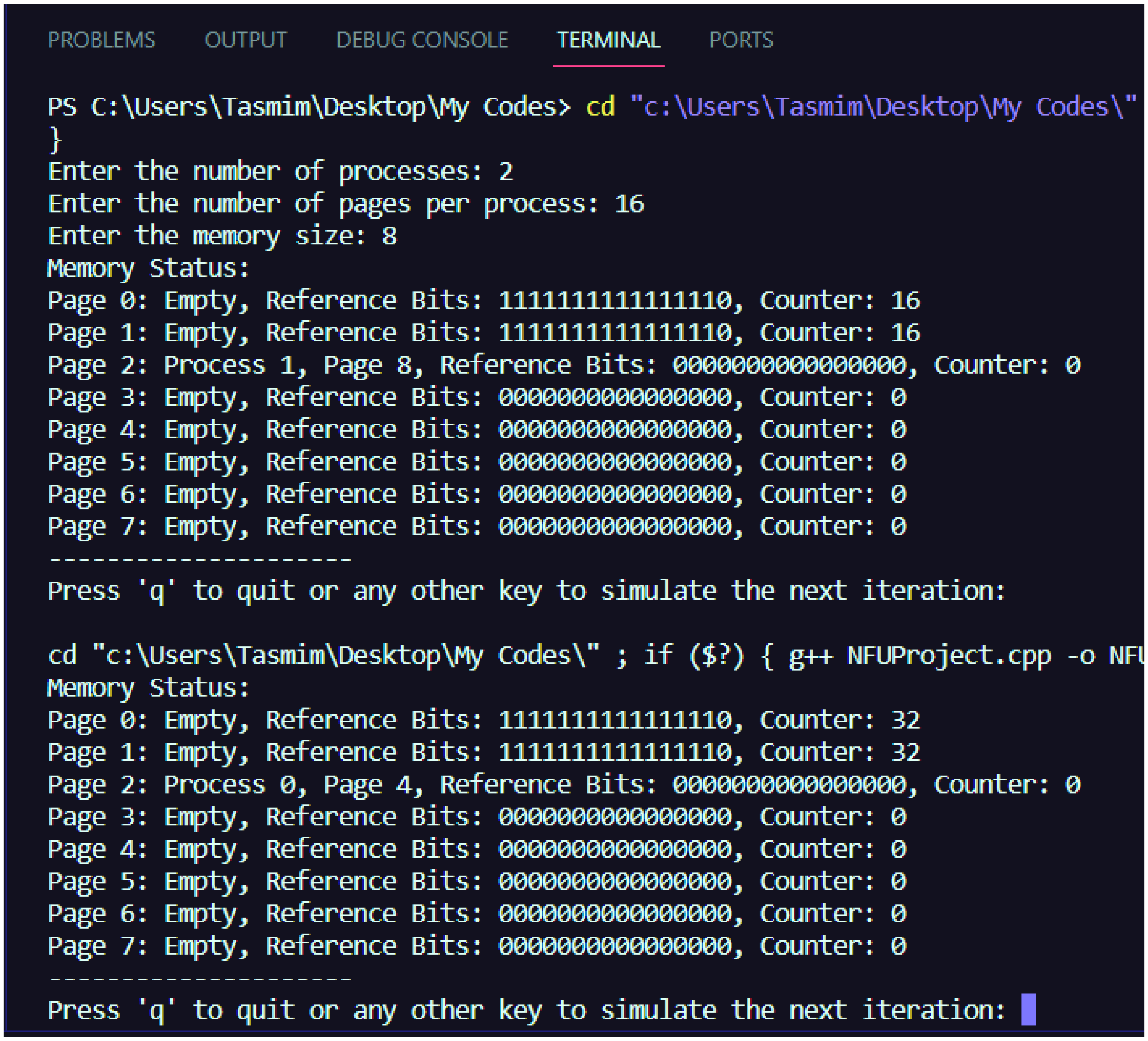
nfuAlgorithm Class - Display Memory Status:

The displayMemoryStatus function prints the current status of the memory. It shows whether each page is empty or occupied, along with its process ID, page ID, and reference bits.

nfuAlgorithm Class - Run Algorithm:

The runAlgorithm function executes the main simulation loop. It repeatedly calls the simulatePageReferences, updateReferenceBits, findLFUIndex, and replaceLFUPage functions, displaying the memory status after each iteration. The user is prompted to continue or exit the simulation based on their input.

<u>Output :</u>



```
PS C:\Users\Tasmim\Desktop\My Codes> cd "c:\Users\Tasmim\Desktop\My Codes\"
}
Enter the number of processes: 2
Enter the number of pages per process: 16
Enter the memory size: 8
Memory Status:
Page 0: Empty, Reference Bits: 1111111111111110, Counter: 16
Page 1: Empty, Reference Bits: 1111111111111110, Counter: 16
Page 2: Process 1, Page 8, Reference Bits: 0000000000000000, Counter: 0
Page 3: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 4: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 5: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 6: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 7: Empty, Reference Bits: 0000000000000000, Counter: 0
---------------------
Press 'q' to quit or any other key to simulate the next iteration:

cd "c:\Users\Tasmim\Desktop\My Codes\" ; if ($?) { g++ NFUProject.cpp -o NF
Memory Status:
Page 0: Empty, Reference Bits: 1111111111111110, Counter: 32
Page 1: Empty, Reference Bits: 1111111111111110, Counter: 32
Page 2: Process 0, Page 4, Reference Bits: 0000000000000000, Counter: 0
Page 3: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 4: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 5: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 6: Empty, Reference Bits: 0000000000000000, Counter: 0
Page 7: Empty, Reference Bits: 0000000000000000, Counter: 0
---------------------
Press 'q' to quit or any other key to simulate the next iteration:
```

Output Explanation :

The output represents the initial state of the memory in a simulated environment using the NFU (Not Frequently Used) page replacement algorithm. The user has inputted that there are two processes, each having 16 pages, and the total memory size is set to 8 pages. In this initial configuration, the memory is predominantly empty, with only one page occupied (Page 2) by Process 1, and its reference bits are all zeros, indicating that it has not been referenced in the initial simulation. The other pages are marked as empty, and their reference bits are all set to 1, representing their initial random reference history. The output reflects the state before any simulated page references, providing a snapshot of the memory's initial composition and reference bit patterns for potential page replacements in subsequent iterations.

Conclusion :

In conclusion, the C++ code successfully implements the Not Frequently Used (NFU) page replacement algorithm in a simulated memory environment. The code employs object-oriented principles, encapsulating the algorithm's logic within the nfuAlgorithm class. It facilitates user interaction by allowing input for the number of processes, pages per process, and memory size, and then proceeds to simulate the NFU algorithm iteratively. The simulation accurately reflects the dynamic nature of page references, aging of pages, identification of the least frequently used page, and replacement of pages based on the algorithm's logic. The user is presented with a clear visualization of the memory status after each iteration, enhancing understanding and providing insights into the NFU page replacement strategy. Overall, the code demonstrates a structured and functional implementation of the NFU algorithm, offering a valuable tool for studying and observing its behavior in a controlled, simulated environment.

References :

1. Andrew S. Tanenbaum - Modern Operating Systems.
2. Operating System Concepts Book by Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin
3. https://www.github.com