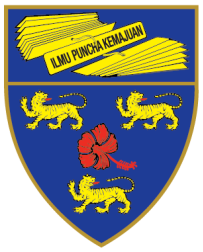


UNIVERSITI MALAYA

WIA2004 Operating Systems

Lab 6: Memory Management Techniques

Group : F5U

Occ: 10

Lecturer: Dr. Fazidah Othman

GROUP MEMBERS	MATRICS NUMBER
ZHANG ZHIYANG	S2193685
MEILIN LI	S2174975
HUMYRA TASMIA	S2176677
HUSNA NADIAH BINTI ROSTHAM	22060027
HUSNA BINTI IHSANUDDIN	U2102305

Table of Content

1.0 Introduction

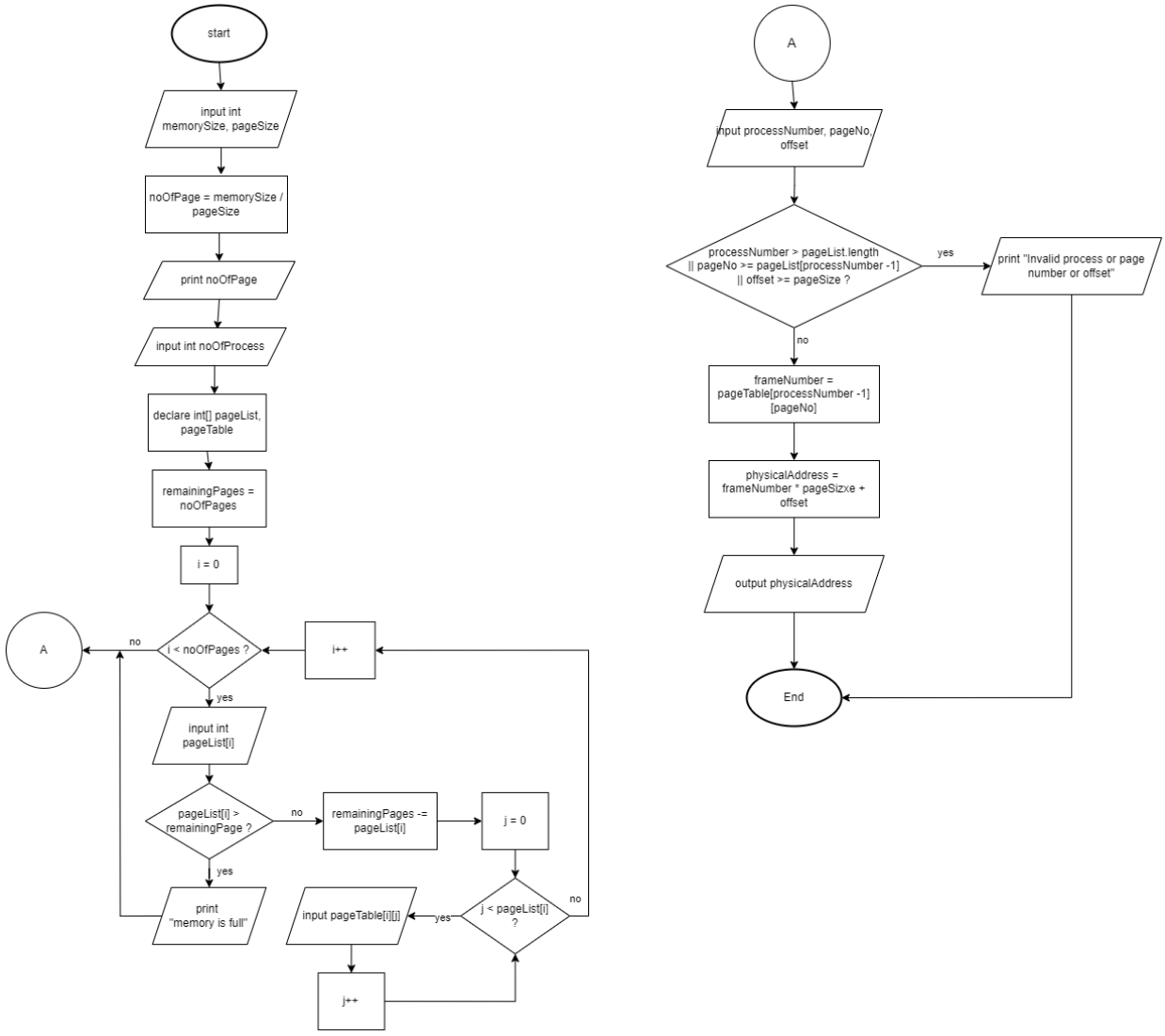
Paging is a memory management scheme used by operating systems to manage memory more efficiently. In paging, memory is divided into fixed-size blocks called "pages," typically ranging from 4KB to 64KB in size. Similarly, the logical memory space is divided into blocks of the same size called "frames." The size of a frame matches the size of a page.

Here's how paging works:

1. **Page Table:** Each process has its own page table, which is maintained by the operating system. The page table keeps track of which pages of the process are currently in memory and where they are located.
2. **Address Translation:** When a program references a memory address, the CPU generates a virtual address. This virtual address is divided into two parts: a page number and an offset within the page. The page number is used to index into the page table to find the corresponding frame number in physical memory.
3. **Page Faults:** If the required page is not currently in memory (a "page fault" occurs), the operating system must load the required page from secondary storage (like a hard disk) into an available frame in physical memory. This process is called "page swapping."
4. **Replacement Policies:** If there are no free frames in physical memory when a page fault occurs, the operating system must choose a page to remove from memory to make room for the incoming page. This decision is based on various replacement policies like Least Recently Used (LRU), First-In-First-Out (FIFO), or others.
5. **Memory Protection:** Paging facilitates memory protection by assigning different permissions (read, write, execute) to different pages. The page table includes bits for each page to indicate whether the page is readable, writable, or executable. Unauthorized access attempts trigger exceptions, protecting the integrity of the system.

2.0 Methodology

Flowchart



Pseudocode

Start

Define main function:

1. Prompt the user to enter the memory size (memorySize)
2. Prompt the user to enter the page size (pageSize)
3. Set noOfPages = memorySize / pageSize
4. Print noOfPages
5. Read the number of processes from the user (noOfProcesses)
6. Initialize arrays pageList, pageTable
    a. pageList = array of integers with size noOfProcesses
    b. pageTable = 2D array of integers with size noOfProcesses
7. Call the allocatePages() method
8. Call the translateAddress() method

Define allocatePages() function:

1. Set remainingPages = noOfPages
2. Iterate for i = 0 to noOfProcesses - 1:
    a. Read the number of pages (pageList[i]) required for the current process
    b. Check if there are enough remaining pages in memory for the current process
        i. if pageList[i] > remainingPages:
        ii. print "Memory is Full for process ", (i + 1)
        iii. return false
    c. Update the remaining pages in memory, remainingPages -= pageList[i]
    d. Initialize the page table for the current process
        i. pageTable[i] = new array of integers with size pageList[i]
        ii. Loop through each page of the current process
        iii. Read the frame number (pageTable[i][j]) for each page
3. If all processes are allocated successfully, return true

Define translateAddress() function:

1. Read the processNumber, pageNo, and offset from the user
2. If input invalid (processNumber > length of pageList or pageNo >= pageList[processNumber - 1] or offset >= pageSize) print error message
3. Otherwise, perform address translation
4. frameNumber = pageTable[processNumber - 1][pageNo]
5. physicalAddress = frameNumber * pageSize + offset
6. print physicalAddress output

End

3.0 Implementation

3.1 Coding

```java
public class OSLAB6 {

    private static Scanner scanner = new Scanner(source: System.in);

    public static void main(String[] args) {
        int memorySize = readInteger(prompt: "Enter the memory size: ");
        int pageSize = readInteger(prompt: "Enter the page size: ");
        int noOfPages = memorySize / pageSize;
        System.out.println("The number of pages available in memory are: " + noOfPages);

        int noOfProcesses = readInteger(prompt: "\nEnter number of processes: ");
        int[] pageList = new int[noOfProcesses];
        int[][] pageTable = new int[noOfProcesses][];

        allocatePages(noOfProcesses, noOfPages, pageList, pageTable);

        boolean continueTranslation = true;
        while (continueTranslation) {
            System.out.println(x: "\n-----Enter Logical Address to find Physical Address-----");
            translateAddress(pageList, pageTable, pageSize);

            continueTranslation = loopresponse();
        }

        scanner.close();
    }


    // Prompts the user with a message and returns the integer input.

    private static int readInteger(String prompt) {
        System.out.print(s: prompt);
        return scanner.nextInt();
    }


    // Allocates pages based on the number of processes and available pages.

    private static boolean allocatePages(int noOfProcesses, int noOfPages, int[] pageList, int[][] pageTable) {
        int remainingPages = noOfPages;
        for (int i = 0; i < noOfProcesses; i++) {
            pageList[i] = readInteger("\nEnter number of pages required for p[" + (i + 1) + "]: ");
            if (pageList[i] > remainingPages) {
                System.out.println("\nMemory is Full for process " + (i + 1));
                return false;  // Memory is full, but continue to allow address translation
```

```
                System.out.println("\nMemory is Full for process " + (i + 1));
                return false;  // Memory is full, but continue to allow address translation
            }
            remainingPages -= pageList[i];
            pageTable[i] = new int[pageList[i]];
            for (int j = 0; j < pageList[i]; j++) {
                pageTable[i][j] = readInteger("Enter frame number for page " + j + ": ");
            }
        }
        return true;
    }


    // Translates logical address to physical address and prints the result

    private static void translateAddress(int[] pageList, int[][] pageTable, int pageSize) {
        int processNumber = readInteger(prompt: "Enter process number: ");
        int pageNo = readInteger(prompt: "Enter page number: ");
        int offset = readInteger(prompt: "Enter offset: ");

        if (processNumber > pageList.length || pageNo >= pageList[processNumber - 1] || offset >= pageSize) {
            System.out.println(x: "\nInvalid process or page number or offset");
        } else {
            int frameNumber = pageTable[processNumber - 1][pageNo];
            int physicalAddress = frameNumber * pageSize + offset;
            System.out.println("The physical address is: " + physicalAddress);
        }
    }


    // Asks the user if they want to continue finding physical addresses

    private static boolean loopresponse() {
        System.out.print(s: "\nEnter 1 to continue finding physical addresses, 0 to stop: ");
        int response = scanner.nextInt();
        return response == 1;
    }
}

```

3.2 Sample output

```
run:
Enter the memory size: 500
Enter the page size: 50
The number of pages available in memory are: 10

Enter number of processes: 3

Enter number of pages required for p[1]: 4
Enter frame number for page 0: 8
Enter frame number for page 1: 6
Enter frame number for page 2: 9
Enter frame number for page 3: 5

Enter number of pages required for p[2]: 5
Enter frame number for page 0: 1
Enter frame number for page 1: 4
Enter frame number for page 2: 5
Enter frame number for page 3: 7
Enter frame number for page 4: 3

Enter number of pages required for p[3]: 2

Memory is Full for process 3

-----Enter Logical Address to find Physical Address-----
Enter process number: 2
Enter page number: 3
Enter offset: 29
The physical address is: 379

Enter 1 to continue finding physical addresses, 0 to stop: 0
BUILD SUCCESSFUL (total time: 1 minute 10 seconds)
```

4.0 Conclusion

In conclusion, paging is a vital memory management technique that offers several benefits, including simplified memory allocation, efficient memory utilization, and robust memory protection through page-level permissions. Despite the overhead introduced by maintaining page tables and handling page faults, paging significantly enhances system performance by enabling efficient memory management and ensuring the integrity and security of processes.

5.0 References

- Paging in Operating Systems: What it Is & How it Works (phoenixnap.com)

- Modern operating systems : Tanenbaum, Andrew S., 1944- : Free Download, Borrow, and Streaming : Internet Archive https://archive.org/details/modernoperatings0000tane