CSE 312
OPERATING SYSTEMS

HW2
REPORT

OGUZ MUTLU
1801042624

1.PROBLEM DEFINITION

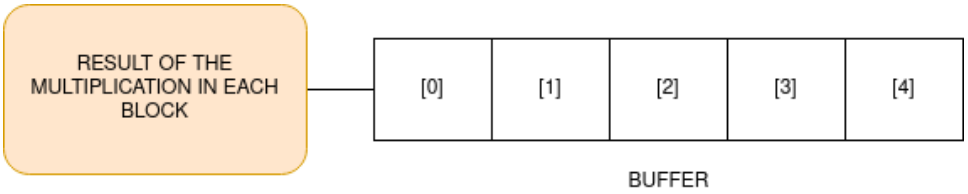
Your simplified system will do simple integer array arithmetic with 2 different multiplication algorithms (matrix (e.g.: 1000 by 3) - vector (e.g.: 3 by 1) Multiplication, vector (e.g.: 1 by1000)-vectorT Multiplication) and an array summation algorithm (Array Summation) 2 different search algorithms (linear search and binary search).

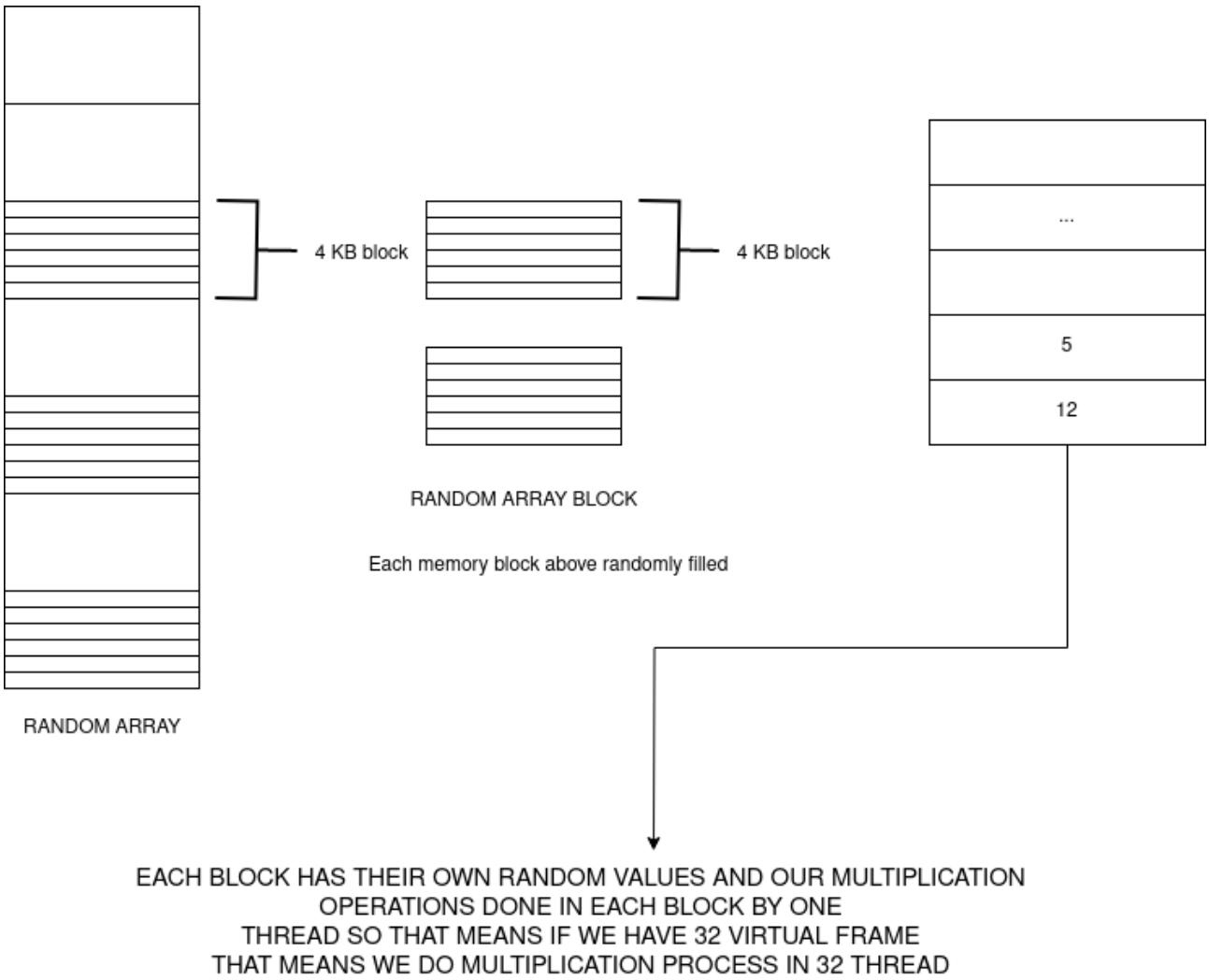
After filling the entire virtual memory with random integers, you will use C/C++ threads, one for each multiplication and summation operations. Each thread will work on a different part of the virtual array with equal size of the virtual memory.

2.IMPLEMENTATION

2.1 SCENARIO CREATION

First, I randomly created an array of virtual memory frame size * frame_size. Then, I filled this array with random numbers. Each block represents a window. For example, if the virtual memory is divided into 16 blocks and each block is 4kb in size, I first partition the array into these blocks. Then, I fill each 4096-byte segment with random integers and create a 1024x4 array from this array. Then, I repeat the randomization process and multiply the result with a 4x1 vector. After obtaining the result, I take its transpose to obtain a single value. I then put this obtained value in a buffer.





The purpose of this buffer is to determine the corresponding virtual address space. Through these multiplication operations, I obtain a value that serves as my access address. If there are -1 values within the virtual page, it means that an unmapped value has been encountered. In this case, I go and read the .dat file and write the data to the physical memory. If the address is mapped, I increment the counter value that I keep as a reference bit in the address (simulating LRU using NRU). I continue reading until the end of the file.

2.2 PRODUCER-CONSUMER IMPLEMENTATION

While implementing this assignment, I attempted to adhere to the producer-consumer structure. My logic in the implementation was as follows: After performing mathematical calculations on the randomly filled arrays, I stored the results in another array. In this array, I took the modulus of the results based on the number of virtual frames. If there are 32 virtual frames, the values will range from 0 to 31. Considering the possibility of multiple occurrences of the same value, I found it more reasonable to use these values as an access ranking instead of placing them in the page table. For example, if the array is {5, 15, 31, 0, 2, 5, 5}, it means that the index 5 in the page table is accessed 5 times. Therefore, this array, which I defined as the buffer, is filled with the results of mathematical operations by the producer and then attempted to be emptied by the consumer. For synchronization purposes, I used **`pthread_mutex`**.

```
void* consumer_function(void* arg){

    int get_data = buffer[0];

    // for(int i = 0; i < pm.get_memory_size(); i++){
        cout<<"Reading "<<get_data<<" index from the virtual memory
        cout<<"Virtual Address Space: "<< vm.get_data(get_data)<<en

    // }


    pthread_mutex_lock(&mtx);

    buffer.erase(buffer.begin());

    // sort_buffer_linear_search(buffer);

    //choose algorithm
    int page_replacement_algorithm = 1; //LRU for 1


    cout<<"Physical Memory Data :" <<vm.get_value(1, get_data, pm)<

    pthread_mutex_unlock(&mtx);


}
```

```
void* matrix_vector_multiplication_thread(void* arg){
    vector<vector<int>> rand_numbers(page_size/window_size, vector

    vector<int> random_vector(page_size/window_size);

    srand(time(0));

    for(int i = 0; i < page_size/window_size; i++){

        for(int j = 0; j < window_size; j++){
            rand_numbers[i][j] = (rand()%1000);
        }

        random_vector[i] = (rand() % 1000);
    }



    vector<int> result_vector = matrix_vector(rand_numbers, random
    int transpose_vector_result =abs(matrixMultiplyWithTranspose(r


    pthread_mutex_lock(&mtx);
    counter++;
    // cout<<"Virtual Frames: "<<virtual_frames<<endl;
    buffer.push_back(transpose_vector_result%virtual_frames);
    pthread_mutex_unlock(&mtx);

    pthread_exit(NULL);
}
```

The locking is performed by the producer while performing operations, and it belongs to the producer, whereas the consumer threads utilize it when accessing the consumer side. You can see the producer consumer algorithm(below) and my own implementation(above). The reason why I did not keep a counter is because I am using a ready data structure.

2.3 LRU IMPLEMENTATION

```
int VirtualMemory ::LRU(int value, PhysicalMemory& pm, int address)
{

    int temp;
    int temp_address;
    int temp_vm_address;
    for(int i = 0; i < page_table_size; i++){

        if(i == 0){
            temp = page_table[i].counter;
            temp_address = page_table[i].mapping_address;
            temp_vm_address = i;
        }

        if(temp < page_table[i].counter){
            temp = page_table[i].counter;
            temp_address = page_table[i].mapping_address;
            temp_vm_address = i;
        }
    }
```

It requires a software counter associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the R bit, which is 0 or 1, is added to the counter. The counters roughly keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement. In the above code line I took counter in my page table entry and increase in each reach.

After page fault occurs and page replacement operation done I reset all the counters to the zero, and when page accessed I increase counters. You can see the code in the below. After mapping operation done, I remove older map from the virtual memory, and replace new one after that reset all counter values to the 0 **(Since hardware interrupt doesnot handled I use NRU for simulating the LRU)**

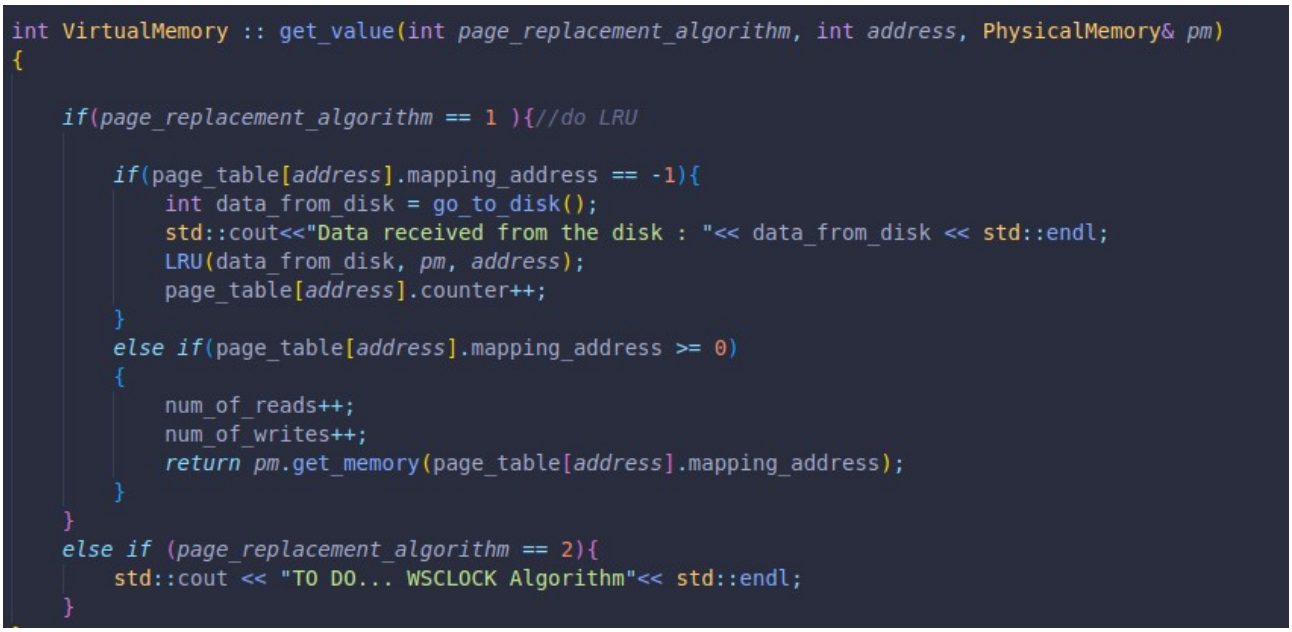
```
pm.set_memory(page_table[temp_vm_address].mapping_address, value);
page_table[temp_vm_address].counter = 0;
page_table[address].mapping_address = page_table[temp_vm_address].mapping_address;
page_table[temp_vm_address].mapping_address = -1;

for(int i = 0; i < page_table_size; i++){
    page_table[i].counter = 0;
}
```

The explanation of the code below would be as follows:

Firstly, we have a conditional statement for the page_replacement_algorithm variable that we intend to get from the user. We inform the user that they should enter 1 for LRU and 2 for WSCLOCK.

If we want to find the mapping address to access, we check the provided integer value in the address variable against the index in the struct. If we find a value of -1, it means that this value has not been used or mapped before. Using LRU, we find the lowest counter value and write it from the disk to the physical memory, taking it from the current cursor position.



```
int VirtualMemory :: get_value(int page_replacement_algorithm, int address, PhysicalMemory& pm)
{

    if(page_replacement_algorithm == 1 ){//do LRU

        if(page_table[address].mapping_address == -1){
            int data_from_disk = go_to_disk();
            std::cout<<"Data received from the disk : "<< data_from_disk << std::endl;
            LRU(data_from_disk, pm, address);
            page_table[address].counter++;
        }
        else if(page_table[address].mapping_address >= 0)
        {
            num_of_reads++;
            num_of_writes++;
            return pm.get_memory(page_table[address].mapping_address);
        }
    }
    else if (page_replacement_algorithm == 2){
        std::cout << "TO DO... WSCLOCK Algorithm"<< std::endl;
    }
```

RESULTS

In the results section, I derived the following result: First, it checks whether there is a value present at index 183. If it doesn't exist, it visits the disk file and retrieves the value, which is then loaded into the physical memory.