Gebze Technical University

Department Of Computer Engineering

CSE 312 Spring 2023

Operating System

Homework 2

Due Date: 04.06.2023

Abdurrahman Bulut
1901042258

Introduction

In this project, I was asked to design and implement a simulated and simplified virtual memory management system in C/C++. My system was required to be able to perform simple integer array arithmetic with 2 different multiplication algorithms, an array summation algorithm, and 2 different search algorithms. I used threads to perform the arithmetic and search operations.

I was also required to design a page table structure that could be used to implement Second-Chance (SC), Least-Recently-Used (LRU), and Working Set Clock (WSClock) algorithms. I also needed to implement a program called operateArrays that could be used to run my virtual memory system.

I didn't implement a third part. And also I am not sure about multiplication and inverted parts..

Part 1

Page table

```
struct entry
{
    unsigned long recent_access_time;
    int lru_used;
    int holding_page;
    int reference_bit;
    int modified_bit;
    int is_present;
};

/* Virtual,Physical memory arrays */
struct entry *physical_page_array;
struct entry *virtual_page_array;
```

In expressing both virtual and physical memory accurately, I preferred utilizing two separate yet parallel data structures for this purpose. Firstly, there was a C integer arrangement intended for holding down Virtual Memory addresses whilst secondly there being Struct-like arrangements customized as per Page Table Entries expected to serve Virtual Memory purposes in this context. From an overall perspective on sizing scale issues relating with aforementioned arrangements; The first or rather previously stated data structure tends to come along constituting not less than

frame_size multiplied by assigned value representing allocated quantity on said Memory Spaces Physical and Virtual. Contrarily subsequential Customized Structure-centric arrangements tentatively might directly align towards allocated space quantity destined for said Memory Space areas in this context.

The struct entry had the following fields:

recent_access_time: The time at which the page was last accessed.
lru_used: Counter to create a time limit for least recently used algorithm.
holding_page: The page number of the page that is currently stored in the frame.
reference_bit: A flag indicating whether the page has been referenced since it was last brought into memory.
modified_bit: A flag indicating whether the page has been modified since it was last brought into memory.
is_present: A flag indicating whether the page is currently in memory.

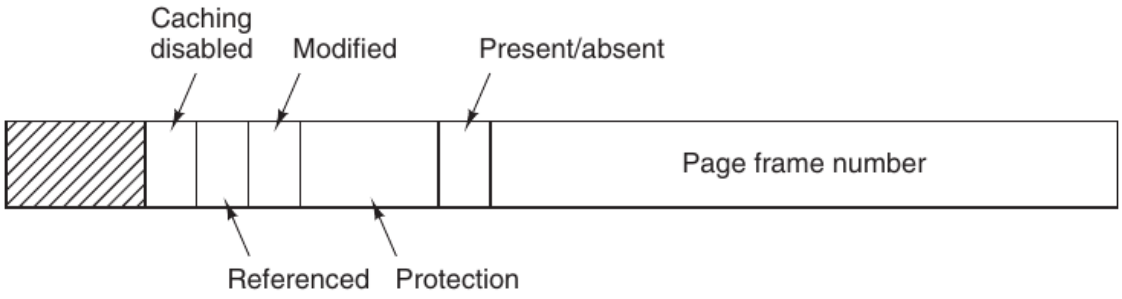


Figure 3-11. A typical page table entry.

Protection bits tell what kinds of access are permitted to a page. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page. For example, a protection bit of 001 would allow the page to be read, but not written to or executed. Modified bit keeps track of whether a page has been modified since it was last loaded into memory. When a page is written to, the hardware automatically sets the modified bit. This bit is used by the operating system to determine whether a page needs to be written back to disk when it is evicted from memory. Referenced bit keeps track of whether a page has been referenced since it was last loaded into memory. The referenced bit is set whenever a page is accessed, either for reading or for writing. This bit is used by the operating system to determine which pages are most likely to be used again in the near future. This information can be used to make better decisions about

which pages to evict from memory when there is not enough space to keep all of the pages that are currently in use.

```c
struct entry *physical_page_array;
struct entry *virtual_page_array;

struct statistics
{
    int reads;
    int writes;
    int disk_reads;
    int disk_writes;
    int page_misses;
    int page_replacement;
};

struct statistics statistics[8];

int second_chance_index[4], ws_clock_index[4];
int lru_counter;

int frame_size, num_physical, num_virtual, page_table_print_int;
char page_replacement[20], alloc_policy[20], disk_file_name[500];

int physical_mem_page_num, virtual_mem_page_num;
int frame_size_int, num_physical_int, num_virtual_int;

pthread_mutex_t mutex;
int thread_no[4];
pthread_t tid[4];

int memory_access;

int *virtual_memory;
int *physical_memory;

int *mult1arr;
int *mult2arr;
```

To manage virtual and physical memory. I utilized two parallel data structures. For virtual memory I relied on a C integer array as well as a struct array for virtual memory page table entries. The C integer array was sized based on frame_size multiplied by the number of memory frames while the struct array was based solely on the number of memory frames. The overall memory system includes three main data structures: virtual memory consisting of an integer array and a structure, physical memory consisting of an integer array and a structure, and a disk which takes the form of a file. Virtual memory contains addresses equivalent to the total number of random integers with these addresses pointing to both physical memory and disk indexes. Physical memory and disk hold random values from virtual memory. If the is_present bit is not set to 1 in virtual memory it implies that this page or indexes value is unavailable in physical memory. Conversely. If the is_present bit equals 1. It signifies that the value exists in physical m

memory. Due to virtuals' superior size compared to physicals' capacity. There will be instances when physical storage may not suffice. Data will automatically shift over to using disk as secondary storage solution.

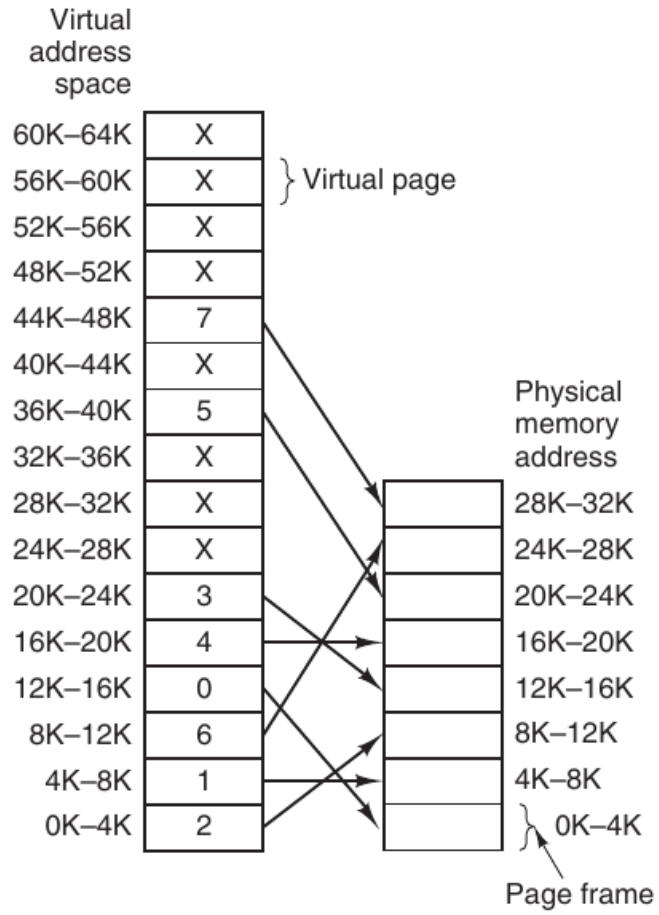
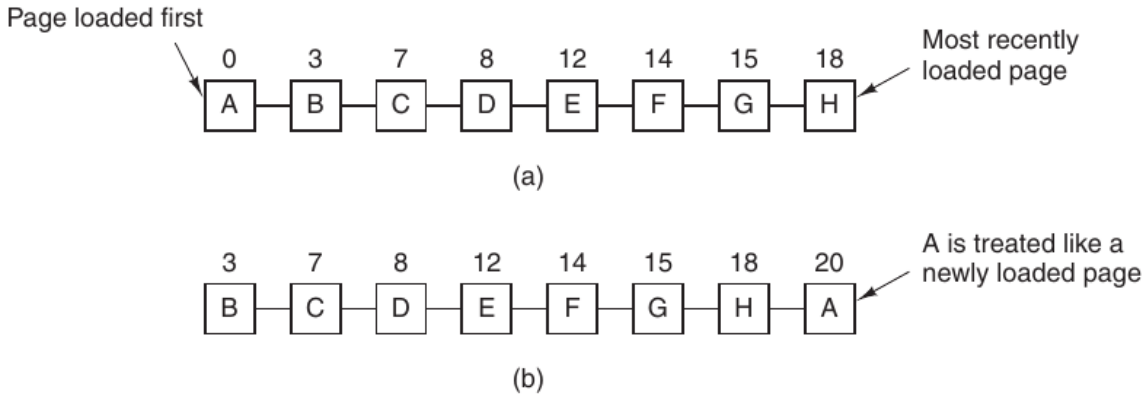


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

The page table is a critical component of virtual memory management. It allows the CPU to translate virtual addresses to physical addresses, which is necessary for programs to access memory. The page fault is a mechanism that is used to handle situations where a page is not in memory. The MMU is a hardware device that is responsible for translating virtual addresses to physical addresses.

First, the value of the page or the address of the page is found in virtual memory. To access the value of the address found, it is first checked if it exists in physical memory. If it does, it becomes a “hit” and that value is returned directly. If not found, this will be a “page fault”. In this case, since the value sought in physical memory cannot be found, the page containing the value sought from the secondary storage, i.e. disk, must be brought to physical memory. The best page to replace with page replacement algorithms is found and replaced with a new one. After the page is changed, the value called from physical memory is taken and returned. During this exchange, of course, the values of the bits in the page table are updated and changed in the access times and the held page variables. In other words, if the 30th page is held in the 0th page of physical memory, for example, the holding page parameter changes since the 10th page will be held after the exchange. Time, reference bit, modified bit, first_index, and is_present bits are updated as appropriate. All this is done in get and set functions.

Second Chance Page Replacement Algorithm (307 - 365)



Second Chance is a page replacement algorithm that considers both old and new pages. It works by maintaining a queue of pages, with older pages at the tail and newer pages at the head. When a page is accessed, its reference bit is set to 1. When a page needs to be replaced, the algorithm checks the reference bits of all the pages in the queue. If the reference bit for a page is 1, it means that the page has been recently accessed and is therefore more likely to be used again soon. In this case, the page is moved to the head of the queue, giving it a "second chance" to avoid being replaced. If the reference bit for a page is 0, it means that the page has not been used recently and is therefore less likely to be used again soon. In this case, the page is replaced and its contents are loaded from disk.

I implemented this algorithm by maintaining a sc_index counter. I increment the sc_index counter every time a page is accessed. When a page needed to be replaced, I checked the

reference bits of all the pages in the queue, starting with the page at the current sc_index. If the reference bit for a page was 1, I reset the reference bit to 0 and moved the page to the head of the queue. If the reference bit for a page was 0, I replaced the page and loaded its contents from disk.

LRU Page Replacement Algorithm (378 - 440)

The Least Recently Used (LRU) algorithm selects the page that has been least recently used for replacement. It does this by keeping track of the last time each page was used.

To implement the LRU algorithm, I created a linked list of all the pages in physical memory. Each page in the linked list has a reference to the page that was last used before it. When a page needs to be replaced, I start at the head of the linked list and follow the references until I find a page that has not been used recently. I then replace that page with the new page.

The LRU algorithm is a good choice for page replacement because it is likely to select pages that are not needed anymore. This can help to improve the performance of the system by reducing the number of page faults.

LRU Page Replacement Algorithm (443 - 505)

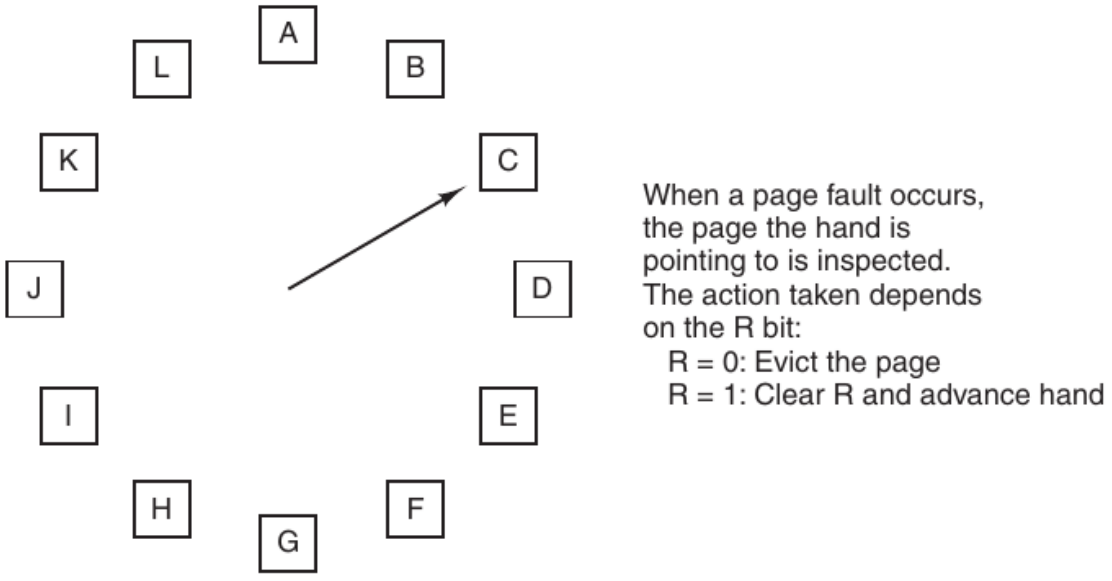


Figure 3-16. The clock page replacement algorithm.

The WSClock (Working Set Clock) algorithm is a page replacement algorithm that uses a circular list or clock to track the usage of pages in memory. When a page fault occurs, the WSClock algorithm searches the list for the oldest page that has not been used recently. If the oldest page has been used recently, the WSClock algorithm sets the page's reference bit to 0 and delays its eviction to the next round. This gives the page a chance to be used again before being replaced. The WSClock algorithm is a good choice for page replacement because it is likely to select pages that are not needed anymore.

Part 2

setValue: This function sets a value at a specific index in the memory. If called with the third parameter (tName) as "fill", it fills the memory. Otherwise, it replaces the line given as an index with the provided value. After each operation, it checks if it's time to print the page table (based on the variable memory_access) and does so if needed.

get_direct_disk: This function reads and returns a line directly from the disk using the given index.

get_disk: This function is similar to get_direct_disk, but it uses the value at the given index in virtual_memory to determine which line to read from the disk.

get: This function is a bit more complex. It retrieves the value at a given virtual memory index, handling several different scenarios based on the value of the tName parameter. If the data is in physical memory, it gets returned directly. If not, the function uses a page replacement algorithm (Second Chance, Least Recently Used, or Working Set Clock) to free up space in physical memory and then retrieves the data from the disk. It also handles various statistics, such as counting disk reads, disk writes, page replacements, and page misses for different operations.

get_page_in_disk: This function performs a page replacement by moving the content of a page from the disk to physical memory.

is_in_physical_mem: It checks whether the index is in the physical memory or not. It returns the index if it is in the physical memory, else it returns -1.

get_value_from_physical_mem: It retrieves the value at a specific index from the physical memory if it exists. If the index does not exist in the physical memory or if the value is negative, it will return -1 and print an error message.

free_memories(): This function frees up the memory allocated to various arrays used in the program.

mult1: This function performs matrix-vector multiplication. The input is a matrix and a vector, both represented as 1-D arrays. The result is a new array representing the multiplication result.

mult2: This function performs a multiplication of two vectors where one vector is transposed. The result is a new array representing the multiplication result.

linear_search: It performs a linear search on the array starting from start_index till start_index+size to find the target value. If the target is found, it prints the index of the target value, otherwise it prints a message indicating that the target value is not found.

bubble_sort: This function sorts an array using the bubble sort algorithm starting from the start_index for a length of size.

binary_search: This function performs a binary search on a sorted array starting from start_index to end_index to find the target value. If the target is found, it returns the index, otherwise it returns -1.

arr_sum: This function calculates the sum of all elements in the array that are created with mıltiplication operations until it encounters a zero. The sum is then printed to the console.

The thread_func() function is made to function in its own thread. It accepts a pointer input that it interprets as a pointer to an integer that represents the "quarter" of the operation that the thread should execute. Different tasks, such as some multiplication operations, sorting and searching, and finally computing and publishing the total of particular results, are assigned to each quarter. The structure of the function is as follows:

Quarter 0: If the quarter value is 0, the function multiplies the first quarter of the virtual_memory array with a matrix and a vector and estimates the time it takes to do so (mult1).

Quarter 1: If the quarter parameter is set to 1, the function multiplies the first quarter of the virtual_memory array by two vectors (mult2) and determines the time required to complete the operation.

Quarter 2: If the quarter value is 2, the function initially uses the bubble_sort function to sort a section of the virtual_memory array. Then, it does binary searches in the sorted section of the virtual_memory array for specific target values, printing whether or not each target value is there. Additionally, it calculates how long binary search operations take. Following that, it repeats linear searches for the same target values in the array and clocks the length of time it takes.

Quarter 3: If the third quarter is selected, the function computes and prints the sums of the outcomes of the earlier mult1 and mult2 actions. It measures how long these operations take as well. The function does not produce a result; instead, it returns NULL since it is intended to have side effects (such as modifying the system's state or outputting output). Each thread that calls this function will be given a "quarter" and will work in parallel with the other threads to complete the tasks related to that quarter.

Tests

```
bulut@bulut:~/Desktop/os_hw2$ make
make: Nothing to be done for 'all'.
bulut@bulut:~/Desktop/os_hw2$ ./main 5 2 4 LRU local 10000 disk.dat
System Info:
Frame Size: 5 (160 integers)
Number of Physical Frames: 2 (10 integers)
Number of Virtual Frames: 4 (20 integers)
Page Replacement Algorithm: LRU
Allocation Policy: local
Page Table Print Interval: 10000
Disk File Name: disk.dat

5 is not in the this part of the virtual memory. binary search
7 is not in the this part of the virtual memory. binary search
9 is not in the this part of the virtual memory. binary search
11 is not in the this part of the virtual memory. binary search
13 is not in the this part of the virtual memory. binary search

Multiplication 1 result: Sum: 1000742

Multiplication 2 result: Sum: 0
Target value 1745795202 not found in the array
Target value 7 not found in the array
Target value 247538188 not found in the array
Target value 11 not found in the array
Target value 13 not found in the array
index:0    ----    value:766020790
index:1    ----    value:1182770779
index:2    ----    value:1333893513
index:3    ----    value:173226398
index:4    ----    value:1071903604
index:5    ----    value:1702255141
index:6    ----    value:2121871803
```

main.cpp | disk.dat | makefile

disk.dat

```
0766020790
1182770779
1333893513
0173226398
1071903604
1702255141
2121871803
2124051570
0983886268
1364009855
1991873138
0779257283
1653856994
1570801147
0147856433
0203709553
1220495216
1564659500
1538721691
1742383952
1756404237
1634722517
1712204171
1223105346
1405486716
0644531718
1511256847
0165749840
1133612654
0559409836
1302352056
1899633444
1742180616
```

```
index:388    ----    value:1046469520
index:387    ----    value:1745795202
index:386    ----    value:114802036
index:385    ----    value:691206573
index:384    ----    value:842291272

*FILL STATISTICS*
Number of reads: 0
Number of writes: 640
Number of page misses: 0
Number of page replacements: 0
Number of disk writes: 384
Number of disk reads: 0

*MULT1 STATISTICS*
Number of reads: 0
Number of writes: 0
Number of page misses: 0
Number of page replacements: 0
Number of disk writes: 0
Number of disk reads: 252

*MULT2 STATISTICS*
Number of reads: 0
Number of writes: 0
Number of page misses: 0
Number of page replacements: 0
Number of disk writes: 0
Number of disk reads: 256

*ARRSUM STATISTICS*
Number of reads: 0
Number of writes: 0
Number of page misses: 0
Number of page replacements: 0
Number of disk writes: 0
Number of disk reads: 0

*LINEAR STATISTICS*
Number of reads: 0
Number of writes: 0
Number of page misses: 0
Number of page replacements: 0
```