

CSE 312 – Operating Systems

Homework 2 Report

Virtual memory management is a crucial component of modern computer systems, allowing efficient memory utilization and enabling processes to access a larger address space than the physical memory capacity.

In this project, I designed and implemented a page table structure to support three different page replacement algorithms: Second-Chance (SC), Least-Recently-Used (LRU), and Working Set Clock (WSClock).

• Part 1

Page table design

referenced_bit	modified_bit	present_absent_bit	page_frame_num
----------------	--------------	--------------------	----------------

The page table structure consists of an array of page table entries, both for the physical and virtual memory. Each page table entry contains the following fields:

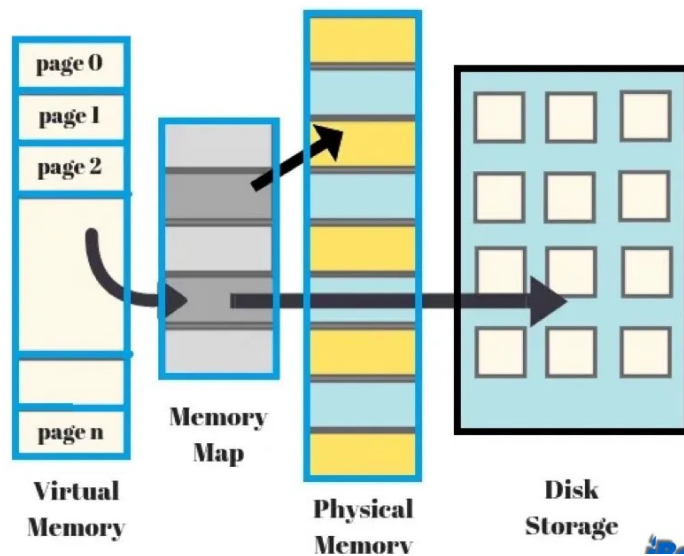
Referenced bit : This bit is used in page replacement algorithms to track whether a page has been recently accessed or referenced. It is set by the hardware whenever a page is accessed. The referenced bit is set whenever a page is referenced, either for reading or writing.

Modified bit : Also known as the "dirty bit," the modified bit is used to track whether a page has been modified or written to since it was last loaded into memory. It is set by the hardware whenever a write operation is performed on a page. The modified bit helps the operating system determine whether a modified page needs to be written back to the disk before it is replaced in memory.

Present/absent bit : The present/absent bit is used in virtual memory systems to indicate whether a page is currently present in physical memory or not.

Page frame num : The page frame number refers to the physical memory frame where a particular page is stored. In virtual memory systems, physical memory is divided into fixed-size frames, and each frame can hold one page. The page frame number is used to map a virtual page to its corresponding physical frame during address translation.

The image below illustrates the overall structure:



The implementation of the page table structure is done in C using a combination of arrays and structures. The main program initializes the statistics and creates the virtual and physical memory arrays, as well as the page table arrays. The program then proceeds to fill the virtual memory array with random integers and initializes the statistics.

```
struct PageTableEntry {  
    //int caching_disabled;  
    int referenced_bit;    //  
    int modified_bit;  
    //int protection_bit;  
    int present_absent_bit;  
    int page_frame_num;    //  
};
```

Page Replacement Algorithms

The program supports three page replacement algorithms: SC, LRU, and WSClock. These algorithms determine which pages should be evicted from physical memory when a page fault occurs. The algorithms keep track of page references and modify the page table entries accordingly.

Second-Chance (SC) :

The SC algorithm uses a "referenced bit" to track page references. When a page fault occurs, the algorithm scans the page table entries in a circular manner and evicts the first page with a referenced bit of 0. If all referenced bits are 1, the algorithm clears the referenced bits and starts scanning again.

Least-Recently-Used (LRU) :

The LRU algorithm maintains the order of page accesses by updating the page table entries at every array reference. Each page table entry is associated with a timestamp indicating the last access time. When a page fault occurs, the algorithm selects the page with the oldest timestamp for eviction.

Working Set Clock (WSClock) :

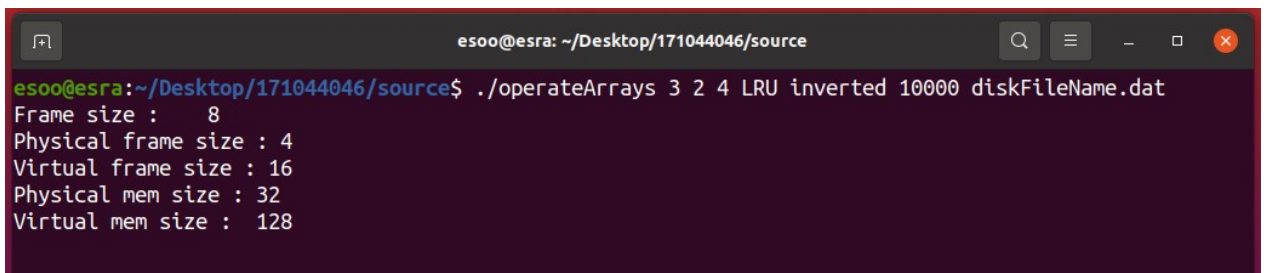
The WSClock algorithm is a modified version of the SC algorithm that considers the working set of each thread. It uses a clock hand to iterate through the page table entries and evicts the first page with a referenced bit of 0. Additionally, the algorithm adjusts the clock hand based on the working set of each thread.

• Part 2

I can summarize the program as follows:

- Firstly, I obtain the values specified in the assignment from the user and assign them to global variables. I fill my virtual memory and physical memory arrays with random numbers, and then I handle the matrix and search operations using threads.
- While doing these operations, I utilize two important functions called "get" and "set" functions. When performing get and set operations in these functions, I employ page replacement algorithms.
- Additionally, I update the statistics when necessary.

Below are the images of the code and outputs :



```
esoo@esra: ~/Desktop/171044046/source
esoo@esra:~/Desktop/171044046/source$ ./operateArrays 3 2 4 LRU inverted 10000 diskFileName.dat
Frame size :      8
Physical frame size : 4
Virtual frame size : 16
Physical mem size : 32
Virtual mem size : 128
```

Some variables and statistics struct :

```
// Virtual, Physical memory array and page tables
int* virtualMemory;
int* physicalMemory;
struct PageTableEntry* physical_page_array;
struct PageTableEntry* virtual_page_array;
```

```
// to hold statistic
struct Statistics {
    int num_reads;
    int num_writes;
    int num_page_misses;
    int num_page_replacements;
    int num_disk_page_writes;
    int num_disk_page_reads;
    int working_set;
};
```

set() function :

- Check the page table,
- If the page is not present, use a page replacement algorithm to place the page in memory,
- If a page replacement is necessary, read the data from the disk and place it in the target frame,
- Write the value to the target frame,
- Update the changes in the page table (referenced bit, modified bit, etc.).

some snippets from the code:

```
void set(int virtual_address, int value) {
    int virtual_page_number = virtual_address / frame_size;
    int offset = virtual_address % frame_size;

    virtualMemory[virtual_address] = value;
    // Check if the page is present in the physical memory
    if (virtual_page_array[virtual_page_number].present_absent_bit == 1) {
        int physical_frame_number = virtual_page_array[virtual_page_number].page_frame_num;
        int physical_address = physical_frame_number * frame_size + offset;

        // Write the value to the physical memory
        physicalMemory[physical_address] = value;

        // Set the modified bit of the page table entry
        virtual_page_array[virtual_page_number].modified_bit = 1;

        // Increment the write counter
        linear_search_static.num_writes++;

        //printf("Value set at virtual address %d: %d\n", virtual_address, value);
    }
    else {
```

get() function :

- Check the page table,
- If the page is not present, use a page replacement algorithm to place the page in memory,
- If a page replacement is needed, read the data from the disk and place it in the target frame,
- Read the page and return the relevant data.

some snippets from the code :

```
int get(int virtual_address) {
    int virtual_page_number = virtual_address / frame_size;
    int offset = virtual_address % frame_size;

    // Check if the page is present in the physical memory
    if (virtual_page_array[virtual_page_number].present_absent_bit == 1) {
        int physical_frame_number = virtual_page_array[virtual_page_number].page_frame_num;
        int physical_address = physical_frame_number * frame_size + offset;

        // Read the data from the physical memory
        int data = physicalMemory[physical_address];

        // Increment the read counter
        linear_search_static.num_reads++;

        //printf("Data at virtual address %d: %d\n", virtual_address, data);
        return data;
    }
    else {
        // Page fault occurred, need to fetch the page from disk
    }
}
```

Implementation of the page replacement algorithms

some code snippets:

```
int SC_page_replacement() {
    int victim_frame = -1;
    int i = 0;
    while (victim_frame == -1) {
        struct PageTableEntry* entry = &physical_page_array[i];
        if (entry->referenced_bit == 0) {
            if (entry->modified_bit == 1) {
                // Write the page back to the disk
                entry->modified_bit = 0;
                // Write to disk page writes statistic
                multiplication_static.num_disk_page_writes++;
            }
            victim_frame = i;
        } else {
            entry->referenced_bit = 0;
        }
        i = (i + 1) % physical_frame_size;
    }
    return victim_frame;
}
```

```

int LRU_page_replacement() {
    int victim_frame = -1;
    int min_reference = INT_MAX;
    for (int i = 0; i < physical_frame_size; i++) {
        struct PageTableEntry* entry = &physical_page_array[i];
        if (entry->referenced_bit < min_reference) {
            min_reference = entry->referenced_bit;
            victim_frame = i;
        }
    }
    if (victim_frame != -1) {
        if (physical_page_array[victim_frame].modified_bit == 1) {
            // Write the page back to the disk
            physical_page_array[victim_frame].modified_bit = 0;
            // Write to disk page writes statistic
            multiplication_static.num_disk_page_writes++;
        }
    }
    return victim_frame;
}

```

```

int WSClock_page_replacement() {
    int victim_frame = -1;
    static int hand = 0;
    int i = hand;
    while (victim_frame == -1) {
        struct PageTableEntry* entry = &physical_page_array[i];
        if (entry->referenced_bit == 0) {
            if (entry->modified_bit == 1) {
                // Write the page back to the disk
                entry->modified_bit = 0;
                // Write to disk page writes statistic
                multiplication_static.num_disk_page_writes++;
            }
            victim_frame = i;
        } else {
            entry->referenced_bit = 0;
        }
        i = (i + 1) % physical_frame_size;
        if (i == hand) {
            // All frames are referenced, reset the reference bits
            for (int j = 0; j < physical_frame_size; j++) {
                physical_page_array[j].referenced_bit = 0;
            }
        }
    }
    hand = (i + 1) % physical_frame_size;
    return victim_frame;
}

```



```
esoo@esra: ~/Desktop/171044046/source
esoo@esra:~/Desktop/171044046/source$ ./operateArrays 3 2 4 LRU inverted 10000 diskFileName.dat
Frame size : 8
Physical frame size : 4
Virtual frame size : 16
Physical mem size : 32
Virtual mem size : 128

-----
>> Page table has 16 elements.

- Entry 0 -> 656 , 84 , 321 , 461 , 996 , 767 , 527 , 395 ,
- Entry 1 -> 585 , 845 , 967 , 893 , 917 , 323 , 629 , 193 ,
- Entry 2 -> 674 , 313 , 822 , 811 , 561 , 205 , 404 , 314 ,
- Entry 3 -> 870 , 925 , 313 , 985 , 60 , 413 , 649 , 406 ,
- Entry 4 -> 497 , 661 , 558 , 183 , 427 , 84 , 268 , 702 ,
- Entry 5 -> 929 , 234 , 285 , 535 , 248 , 914 , 729 , 922 ,
- Entry 6 -> 917 , 550 , 732 , 477 , 445 , 136 , 482 , 314 ,
- Entry 7 -> 751 , 485 , 989 , 501 , 898 , 638 , 908 , 395 ,
- Entry 8 -> 298 , 465 , 268 , 725 , 549 , 536 , 117 , 168 ,
- Entry 9 -> 461 , 402 , 703 , 709 , 315 , 121 , 320 , 923 ,
- Entry 10 -> 671 , 51 , 89 , 806 , 878 , 571 , 120 , 318 ,
- Entry 11 -> 55 , 108 , 820 , 644 , 436 , 417 , 729 , 734 ,
- Entry 12 -> 882 , 997 , 149 , 120 , 222 , 266 , 288 , 683 ,
- Entry 13 -> 668 , 682 , 391 , 673 , 803 , 711 , 595 , 164 ,
- Entry 14 -> 453 , 685 , 970 , 20 , 946 , 780 , 29 , 692 ,
- Entry 15 -> 579 , 539 , 335 , 14 , 956 , 63 , 439 , 837 ,

-----

>>>> After operations <<<<

-----
>> Page table has 16 elements.

- Entry 0 -> 432304 , 7308 , 104004 , 213904 , 995004 , 590590 , 279310 , 157210 ,
- Entry 1 -> 343980 , 716560 , 937990 , 800128 , 843640 , 105298 , 397528 , 37828 ,
- Entry 2 -> 456298 , 98908 , 678150 , 660154 , 316404 , 42640 , 164428 , 99538 ,
- Entry 3 -> 759510 , 858400 , 98908 , 973180 , 3780 , 171808 , 423148 , 166054 ,
- Entry 4 -> 248500 , 438904 , 313038 , 34038 , 183610 , 7308 , 72628 , 494910 ,
- Entry 5 -> 865828 , 55458 , 82080 , 287830 , 62248 , 838138 , 533628 , 852850 ,
- Entry 6 -> 843640 , 304150 , 538020 , 228960 , 199360 , 18904 , 233770 , 99538 ,
- Entry 7 -> 566254 , 236680 , 981088 , 252504 , 809098 , 408958 , 827188 , 157210 ,
- Entry 8 -> 89698 , 217620 , 72628 , 527800 , 303048 , 288904 , 14040 , 28728 ,
- Entry 9 -> 213904 , 162810 , 496318 , 504808 , 100170 , 15004 , 103360 , 854698 ,
- Entry 10 -> 452254 , 2754 , 8188 , 652054 , 773518 , 327754 , 14760 , 102078 ,
- Entry 11 -> 3190 , 11988 , 674860 , 416668 , 191404 , 175140 , 533628 , 540958 ,
- Entry 12 -> 780570 , 997000 , 22648 , 14760 , 49950 , 71554 , 83808 , 468538 ,
- Entry 13 -> 448228 , 467170 , 154054 , 454948 , 647218 , 507654 , 355810 , 27388 ,
- Entry 14 -> 206568 , 471280 , 943810 , 460 , 897754 , 610740 , 928 , 480940 ,
- Entry 15 -> 336978 , 292138 , 113230 , 238 , 916804 , 4158 , 194038 , 703080 ,

-----
>> Page table has 16 elements.

- Entry 0
Referenced Bit : 0
Modified Bit : 1
Present/Absent Bit : 1
Page Frame Number : 0

- Entry 1
Referenced Bit : 0
Modified Bit : 1
Present/Absent Bit : 1
Page Frame Number : 0

- Entry 2
Referenced Bit : 0
Modified Bit : 1
Present/Absent Bit : 1
Page Frame Number : 0

- Entry 3
Referenced Bit : 0
Modified Bit : 1
Present/Absent Bit : 1
Page Frame Number : 0

- Entry 4
Referenced Bit : 0
Modified Bit : 1
Present/Absent Bit : 1
Page Frame Number : 0

- Entry 5
Referenced Bit : 0
Modified Bit : 1
Present/Absent Bit : 1
Page Frame Number : 0

- Entry 6
Referenced Bit : 0
Modified Bit : 1
Present/Absent Bit : 1
Page Frame Number : 0
```



```

- Entry 7
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 8
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 9
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 10
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 11
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 12
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 13
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 14
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

- Entry 15
Referenced Bit :      0
Modified Bit :       1
Present/Absent Bit :  1
Page Frame Number :   0

```

```

-----
>>>> Program Statistics <<<<
Multiplication Statistics
Number of reads : 64
Number of writes : 64
Number of page misses : 0
Number of page replacements : 0
Number of disk page writes : 0
Number of disk page reads : 0
Estimated working set function w for each thread as a table : 0

Summation Statistics
Number of reads : 64
Number of writes : 64
Number of page misses : 0
Number of page replacements : 0
Number of disk page writes : 0
Number of disk page reads : 0
Estimated working set function w for each thread as a table : 128

Linear Search Statistics
Number of reads : 128
Number of writes : 128
Number of page misses : 129
Number of page replacements : 1
Number of disk page writes : 0
Number of disk page reads : 0
Estimated working set function w for each thread as a table : 128

Binary Search Statistics
Number of reads : 128
Number of writes : 0
Number of page misses : 128
Number of page replacements : 0
Number of disk page writes : 0
Number of disk page reads : 0
Estimated working set function w for each thread as a table : 128

```

```

esoo@esra:~/Desktop/171044046/source$ █

```

Backing Store:

My implementation includes a backing store, represented by the disk file "diskFileName.dat." When a page fault occurs and a page needs to be brought in from the disk, the corresponding data is read from or written to the disk file. This ensures that the system can handle a larger amount of data than what can be accommodated in physical memory.

• Part 3

- Physical memory, limited to 16K integers, it provides faster access for programs, resulting in quicker execution times. However, exceeding its capacity may lead to performance issues like disk swapping.
- Virtual memory with a larger capacity of 128K integers, virtual memory acts as an extension of physical memory by utilizing secondary storage. It allows running larger programs and handling more extensive datasets. Accessing data from virtual memory is slower due to disk I/O operations.
- Memory management side : The operating system's memory management unit handles the translation between virtual and physical addresses using techniques like paging or segmentation (like our homework topic). It maintains a page table or segment table to map virtual addresses to physical addresses (our page table struct).
- Overall, virtual memory provides more extensive memory utilization but with slower access times. Efficient memory management is essential to handle memory demands effectively and optimize system performance.
- What is actually important here is the efficient functioning of page replacement algorithms; if they work effectively, the system will be fast.

(I did not write the algorithm that determines the optimal page size.)

Some problems:

- I don't think the disk read and write operations are being done correctly.
- The algorithms have been implemented, but I believe I'm not updating the values in the page table correctly when using the algorithms as I expected.
- If the user provides high frame sizes, the search functions give errors. That's why I commented out that part and sent it, but it doesn't cause any issues when the size is not too high.