



**GTU Department of Computer Engineering
CSE 312 - Spring 2023
Homework 2 Report**

**Emirkan Burak Yılmaz
1901042659**

Contents

Contents	2
1 Part 1	3
1.1 Page Table Structure	3
1.2 Page Table Entry (PTE)	3
1.3 Page Replacement Algorithms	4
1.3.1 LRU	4
1.3.2 SC	4
1.3.3 WSClock	5
2 Part 2	5
2.1 Memory Management	5
2.2 Simulation	6
3 Part 3	7
3.1 Best Frame Size	7
3.1.1 Small Frame (512MB)	7
3.1.2 Medium Size Frame (1KB-4KB)	8
3.1.3 Large Frame (16KB - 32KB)	9
3.1.4 Conclusion	10
3.2 Best Page Replacement Algorithms	11
3.2.1 LRU	11
3.2.2 SC	11
3.2.3 WSClock	12
3.2.4 Conclusion	12

1 Part 1

1.1 Page Table Structure

The structure of the page table is designed in a way that given page replacement algorithms can work on the same structure. To achieve this, two data structure is used to keep the pages and handle the virtual to physical page mapping. The `std::unordered_map` is used to keep the mapping between physical page number and virtual page number. The `std::list` is used to keep the actual page table entries. The management of the page list enables to use different page replacement algorithm. Front of the list contains the least recently used page and this list can be treated as circular link list to implement WSClock algorithm.

```
You, now | 1 author (You)
class InvertedPageTable {
public:
    enum PageReplacementAlgo {SC, LRU, WSClock};

    InvertedPageTable(int capacity, int frameSize, PageReplacementAlgo replacement, unsigned int tau = 4);
    ~InvertedPageTable();

    long map(long virtualAddr, enum AccessMode mode);

    std::list<PTE *>::iterator findEntry(int virtualFrameID);

    int getSize();

    bool isFull();

    void printTable();

    bool addPage(PTE *entry);

    std::list<PTE *> getPageList();

    PTE *removePage(int virtualFrameID);

    PTE *evictPage(std::vector<PTE *> &modifiedPages);

    PTE *evictPage_SC();

    PTE *evictPage_LRU();

    PTE *evictPage_WSClock(std::vector<PTE *> &modifiedPages);

    void clearReferenceBit();

private:
    int capacity;
    int frameSize;
    unsigned long int sysclock;
    const unsigned int tau;
    PageReplacementAlgo replacement;
    std::unordered_map<int, int> pageTable;
    std::list<PTE *> pageList;
};
```

1.2 Page Table Entry (PTE)

PTE class contains information's related such as mapping between the virtual page number and physical page number and the modified/referenced bits. In addition to these the last reference time is kept tracking with the time field.

You, 4 hours ago | 1 author (You)

```
class PTE {
public:
    PTE(int virtualFrameID, int physicalFrameID);

    ~PTE();

    void reset(int virtualFrameID, int physicalFrameID);

    void setVirtualFrameID(int virtualFrameID);

    void setPhysicalFrameID(int physicalFrameID);

    void setReferenced(bool referenced);

    void setModified(bool modified);

    void setTime(unsigned long int time);

    bool getReferenced();

    bool getModified();

    int getPhysicalFrameID();

    int getVirtualFrameID();

    unsigned long int getTime();

private:
    bool referenced;
    bool modified;
    bool presentAbsent;
    int virtualFrameID;
    int physicalFrameID;
    unsigned long int time;
};
```

1.3 Page Replacement Algorithms

1.3.1 LRU

As it's mentioned earlier, the least recently used page can be found and removed easily by removing the front entry of the page list.

```
PTE *InvertedPageTable::evictPage_LRU()
{
    PTE *leastRecentPage = pageList.front();
    return removePage(leastRecentPage->getVirtualFrameID());
}
```

1.3.2 SC

For the implementation of the SC algorithm the page list is treated as circular link list and whole list traversed until finding the first unreferenced page.

```

PTE *InvertedPageTable::evictPage_SC()
{
    while (true) {
        PTE *oldestPage = pageList.front();
        /* Check the reference bit of the page */
        if (oldestPage->getReferenced()) {
            /* Page has been referenced, reset the reference bit */
            oldestPage->setReferenced(false);
            /* Move the page to the end of the list */
            pageList.pop_front();
            pageList.push_back(oldestPage);
        }
        else {
            /* Page can be evicted */
            return removePage(oldestPage->getVirtualFrameID());
        }
    }
}

```

1.3.3 WSClock

For the implementation of the SC algorithm the page list is treated as circular link list and a counter named sysclock is used. To evict a page, the page list traversed and the first page which last reference time is less than τ (constant class field) is removed. During the traverse, the refence and modified bits of the pages are cleared, and the required disk write operations are implemented.

```

PTE *InvertedPageTable::evictPage_WSClock(std::vector<PTE *> &modifiedPages)
{
    while (true) {
        PTE *currentPage = pageList.front();

        if (currentPage->getModified() || currentPage->getReferenced()) {
            /* Set the current clock time and set the R and M bit to 0 */
            if (currentPage->getModified()) {
                /* Clear the modified bit, and indicate that this page should write back to the disk */
                currentPage->setModified(false);
                modifiedPages.push_back(currentPage);
            }

            currentPage->setReferenced(false);

            currentPage->setTime(sysclock);

            /* Move the page to the end of the list (circular linked list for clock algorithm) */
            pageList.pop_front();
            pageList.push_back(currentPage);
        }
        else if (sysclock - currentPage->getTime() > tau) {
            /* check if the page's age to see if it's older than the tau */
            /* Page can be evicted */
            return removePage(currentPage->getVirtualFrameID());
        }
        /* With each condition check, the clock and arrow (front) are advanced */
        ++sysclock;
    }
}

```

2 Part 2

2.1 Memory Management

The physical memory is represented as integer array and kept in MemoryManagement class. It's the abstraction of physical memory and disk. The memory management is handling with the page table

and disk file by read write operations. To create a physical memory with the virtual address space the parameters such as size of the page frame and the number of physical and virtual frames are needs to be given. The MemoryManagent class has public functions read and write for the usage of the memory.

```
You, 4 hours ago | 1 author (You)
class MemoryManagement
{
public:
    MemoryManagement(
        unsigned long int frameSize,
        unsigned long int numPhysicalFrames,
        unsigned long int numVirtualFrames,
        InvertedPageTable::PageReplacementAlgo replacement,
        const char *diskFileName);

    ~MemoryManagement();

    int getVirtualMemSize();

    int getPhysicalMemSize();

    int getFrameSize();

    int read(long virtualAddr);

    int write(long virtualAddr, int data);

    void printStats();

private:
    unsigned long int frameSize;
    unsigned long int numVirtualFrames;
    unsigned long int numPhysicalFrames;
    const char *diskFileName;
    int *physicalMem;
    InvertedPageTable pageTable;
    pthread_mutex_t mutex;
    MemStats stats;

    long access(long virtualAddr, AccessMode mode);

    long diskPosition(int virtualFrameID);

    long memoryPosition(int physicalFrameID);

    int getFrameID(long addr);

    int getFrameOffset(long addr);

    void initDisk();

    void readDisk(PTE *entry);

    void writeDisk(PTE *entry);
};
```

2.2 Simulation

The virtual memory simulation covers all the virtual space by running simulation function on different virtual memory portions. First the virtual memory is filled randomly by using write function from the MemoryManagement class. After that each portion of the memory is given to a new thread with the

simulation function. With this way simulation function can be executed on different portion of the virtual memory in parallel. Each simulation function makes matrix-vector and vector-transpose multiplications in parallel, and the resulting matrix and vectors are mutually summed, and the two-search algorithm linear and binary searches is applied on the resulting array. The execution statistics are given below.

```
ebylmz@ebylmz: ~/cse/Operating-Systems/hw/hw02/src
~
~
~
29      VF: 998 -> PF: 24      M: 1      R: 1
~
~
30      VF: 1018 -> PF: 23     M: 1      R: 1
~
~
31      VF: 999 -> PF: 8       M: 1      R: 1
~
~
~
target: -24      result: -1
target: 75       result: 106
target: 34       result: 327
#####
# of physical frames: 32
# of virtual frames: 1024
Physical memory size: 131072
Virtual memory size: 4194304
#####
# of Memory and Disk Operations
Read      : 8459506
Write     : 6266649
Page Miss : 3033
Page Replacement: 3001
Disk Page Read : 3033
Disk Page Write : 1872
ebylmz@ebylmz:~/cse/Operating-Systems/hw/hw02/src$ make runMediumFrame
```

3 Part 3

3.1 Best Frame Size

3.1.1 Small Frame (512MB)

Using small page frame is requires many pages replacement due to the frequent page misses.

```
ebylmz@ebylmz: ~/cse/Operating-Systems/hw/hw02/src
~
~
~
29      VF: 7997 -> PF: 27     M: 0      R: 1
~
~
30      VF: 7998 -> PF: 12     M: 0      R: 1
~
~
31      VF: 7999 -> PF: 31     M: 0      R: 1
~
~
~
target: -24      result: -1
target: 75       result: 106
target: 34       result: 327
#####
# of physical frames: 32
# of virtual frames: 8192
Physical memory size: 16384
Virtual memory size: 4194304
#####
# of Memory and Disk Operations
Read      : 8459164
Write     : 6266649
Page Miss : 30968
Page Replacement: 30936
Disk Page Read : 30968
Disk Page Write : 13390
ebylmz@ebylmz:~/cse/Operating-Systems/hw/hw02/src$ make runSmallFrame
```

3.1.2 Medium Size Frame (1KB-4KB)

When we select our frame size around 1KB to 4 KB, the number of page miss becomes less compared to small page frame.

```
ebylmz@ebylmz: ~/cse/Operating-Systems/hw/hw02/src
30      VF: 4062 -> PF: 29      M: 0      R: 1
~~~~~
31      VF: 4082 -> PF: 7       M: 0      R: 1
~~~~~
target: -24      result: -1
target: 75       result: 151
target: 34       result: 244
target: -24      result: -1
target: 75       result: 291
target: 34       result: 430
#####
# of physical frames: 32
# of virtual frames: 4096
Physical memory size: 32768
Virtual memory size: 4194304
#####
# of Memory and Disk Operations
Read      : 8458885
Write     : 6266649
Page Miss : 15277
Page Replacement: 15245
Disk Page Read : 15277
Disk Page Write : 6593
ebylmz@ebylmz:~/cse/Operating-Systems/hw/hw02/src$ make runX
```

1KB Page Frame

```
ebylmz@ebylmz: ~/cse/Operating-Systems/hw/hw02/src
~~~~~
29      VF: 998 -> PF: 24      M: 1      R: 1
~~~~~
30      VF: 1018 -> PF: 23     M: 1      R: 1
~~~~~
31      VF: 999 -> PF: 8       M: 1      R: 1
~~~~~
target: -24      result: -1
target: 75       result: 106
target: 34       result: 327
#####
# of physical frames: 32
# of virtual frames: 1024
Physical memory size: 131072
Virtual memory size: 4194304
#####
# of Memory and Disk Operations
Read      : 8459506
Write     : 6266649
Page Miss : 3033
Page Replacement: 3001
Disk Page Read : 3033
Disk Page Write : 1872
ebylmz@ebylmz:~/cse/Operating-Systems/hw/hw02/src$ make runMediumFrame
```

4KB Page Frame

Using larger frame resulted in having even smaller number of page miss.

16KB Frame

32KB Page Frame

3.1.4 Conclusion

In summary, in our single process simulation, the number of page misses decreases as the size of the page frame is increases. However, with larger page frames, there is a higher chance of internal fragmentation. When a smaller amount of data is stored in a large page frame, the remaining space within the page frame is wasted. This can lead to inefficient memory utilization, especially if most of the allocated memory does not fully utilize the page frames.



3.2 Best Page Replacement Algorithms

3.2.1 LRU

```
ebylmz@ebylmz: ~/cse/Operating-Systems/hw/hw02/src
30      VF: 1005 -> PF: 6      M: 1      R: 1
~~~~~
31      VF: 1020 -> PF: 5      M: 1      R: 1
~~~~~
target: -24      result: -1
target: -24      result: -1
target: 75       result: 105
target: 75       result: 291
target: 34       result: 437
target: 34       result: 430
#####
# of physical frames: 32
# of virtual frames: 1024
Physical memory size: 131072
Virtual memory size: 4194304
#####
# of Memory and Disk Operations
Read      : 8459091
Write     : 6266649
Page Miss : 3040
Page Replacement: 3008
Disk Page Read : 3040
Disk Page Write : 1880
ebylmz@ebylmz:~/cse/Operating-Systems/hw/hw02/src$ make runLRU
```

3.2.2 SC

```
ebylmz@ebylmz: ~/cse/Operating-Systems/hw/hw02/src
~~~~~
29      VF: 1008 -> PF: 27     M: 1      R: 1
~~~~~
30      VF: 1018 -> PF: 13     M: 1      R: 1
~~~~~
31      VF: 1009 -> PF: 10     M: 1      R: 1
~~~~~
target: -24      result: -1
target: 75       result: 104
target: 34       result: 213
#####
# of physical frames: 32
# of virtual frames: 1024
Physical memory size: 131072
Virtual memory size: 4194304
#####
# of Memory and Disk Operations
Read      : 8459126
Write     : 6266649
Page Miss : 2962
Page Replacement: 2930
Disk Page Read : 2962
Disk Page Write : 1862
ebylmz@ebylmz:~/cse/Operating-Systems/hw/hw02/src$ make runSC
```

3.2.3 WSClock

```
ebylmz@ebylmz: ~/cse/Operating-Systems/hw/hw02/src
~~~~~
29      VF: 1021 -> PF: 4      M: 1    R: 0
~~~~~
30      VF: 1018 -> PF: 13     M: 1    R: 0
~~~~~
31      VF: 1014 -> PF: 2      M: 1    R: 1
~~~~~
target: -24      result: -1
target: 75       result: 151
target: 34       result: 244
#####
# of physical frames: 32
# of virtual frames: 1024
Physical memory size: 131072
Virtual memory size: 4194304
#####
# of Memory and Disk Operations
Read       : 8458798
Write      : 6266649
Page Miss   : 3030
Page Replacement: 2998
Disk Page Read : 3030
Disk Page Write : 1932
ebylmz@ebylmz:~/cse/Operating-Systems/hw/hw02/src$
```

3.2.4 Conclusion

According to simulation results Second Change page replacement algorithm resulted in least number of pages miss and page replacement.

