# CSE 312

# HW 2

Burak Kocausta

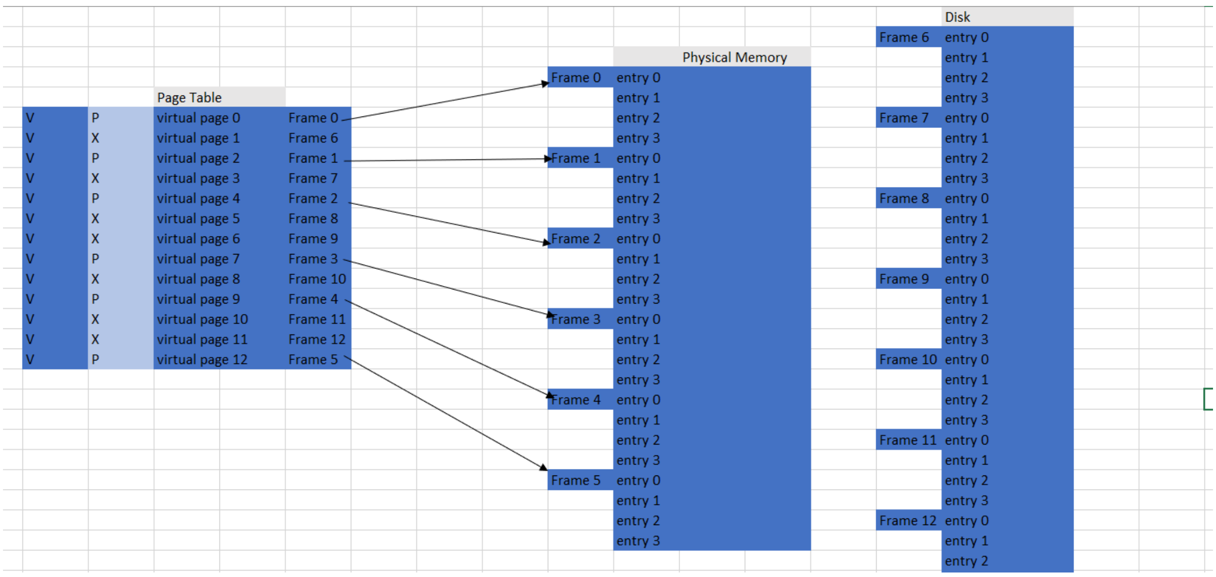
1901042605

## Contents



## 1- General Design

Firstly, I started with designing the virtual address, and physical address. I aimed to implement a memory manager, which simulates page table, physical memory, and uses files as disk.




- If page table has valid entry, it means that page frame is on the disk or physical memory. If present is set, it means that it is on the physical memory. I used .dat file for simulating disk. And whole image is managed by memory manager. It has all access to those structures.

```cpp
class virtualAddress
{
friend class MemoryManager;
friend class MMU;

private:
    size_t pageTableEntryNumber;
    size_t valueOffset;
    size_t pageFrameSize;  // as integer power of 2
    size_t pageTableSize;  // as integer power of 2
public:
    virtualAddress ();
    virtualAddress (size_t pageTableEntryNumber, size_t valueOffset, size_t pageFrameSize, size_t pageTableSiz
    ~virtualAddress ();

    // increment operator
    virtualAddress& operator++ ();
    virtualAddress operator++ (int);

    // method
    void copy (virtualAddress *address);

    void print () const;

};
```

```cpp
class physicalAddress
{
friend class MemoryManager;
friend class MMU;

private:
    size_t frameNumber;
    size_t valueOffset;
    size_t pageFrameSize;  // as integer power of 2
    size_t memorySize; // as integer power of 2

public:
    physicalAddress (const size_t *memorySize);
    physicalAddress (size_t frameNumber, size_t valueOffset, size_t pageFrameSize, size_t memorySize);
    ~physicalAddress ();

    // increment operator
    physicalAddress& operator++ ();
    physicalAddress operator++ (int);

    void print ();
};
```

- Memory manager gets virtual address, then translates it to physical address using MMU.

```cpp
bool MMU::addressTranslation (int processId, const virtualAddress *virtualAddress, physicalAddress *phy
{
    // check if processId is valid
    if (processId < 0)
        return false;

    // convert virtual address to physical address
    physicalAddress->frameNumber = *frameNumber;
    physicalAddress->valueOffset = virtualAddress->valueOffset;
    physicalAddress->pageFrameSize = pageFrameSize;

    return true;
}
MemoryManager::MemoryManager(int maxProcesses, size_t frameSize, size_t physicalMemorySize, size_t page
```

- This is the translation, this function is called by memory manager.

```
class PageFrame
{
friend class MemoryManager;
private:
    size_t frameNumber;
    int processId;
    size_t frameSize;  // as integer power of 2
    size_t numValues;
    int *values;
    bool full;

public:

    PageFrame (size_t frameNumber, int processId, size_t frameSize);
    ~PageFrame ();

    // getters
    size_t getFrameNumber ();
    int getProcessId ();
    size_t getFrameSize ();
    size_t getNumValues ();

    // setters
    void setFrameNumber (size_t frameNumber);
    void setProcessId (int processId);
    void setFrameSize (size_t frameSize);
    void setNumValues (size_t numValues);

    // methods
    bool addValue (int value);
    int getValue (size_t index);
    bool setValue (size_t index, int value);

    bool isFull ();
    void print ();
};
```

- This is a page frame class, it is held in physical memory.

```cpp
class PhysicalMemory
{
private:
    size_t numFrames;
    size_t frameSize;  // as integer power of 2
    PageFrame **frames; // array of pointers to PageFrame objects
    size_t memorySize;
public:
    PhysicalMemory (size_t numFrames, size_t frameSize, size_t memorySize);
    ~PhysicalMemory ();

    // getters
    size_t getNumFrames ();
    size_t getFrameSize ();
    size_t getMemorySize ();

    // setters
    void setNumFrames (size_t numFrames);
    void setFrameSize (size_t frameSize);
    bool setFrame (PageFrame *frame);

    // methods
    bool addFrame (PageFrame *frame);
    PageFrame *getFrame (size_t index);
    void setFrame (int index, PageFrame *frame);
    bool setFrame (size_t index, PageFrame *frame);
    PageFrame* getFrameWithFrameNumber (size_t frameNumber);
    void print ();
};
```

- Physical memory holds page frames. It is an array of pointers.

```cpp
class PageTableEntry
{
friend class MemoryManager;
private:
    int pageNumber;
    size_t frameNumber;
    int processId;

    // valid means that the page is in the physical memory or disk
    bool valid;
    bool dirty;
    bool referenced;


    // present means that the page is in the physical memory
    bool present;
    bool modified;
    int protection;
    size_t pageFrameSize;  // as integer power of 2

    size_t diskLineNumber;

    // last time the page was referenced
    std::chrono::time_point<std::chrono::system_clock> lastReferenceTime;

public:
    PageTableEntry ();
    PageTableEntry (int pageNumber, size_t frameNumber, int processId, size_t pageFrameSize);
    ~PageTableEntry ();

    // getters
    int getPageNumber ();
    size_t getFrameNumber ();
    int getProcessId ();
    bool isValid ();
```

- Page table entry holds valid, dirty, referenced, present, modified, protection values. Also it holds page frame size information and disk line number. Disk line number means, if it is written on the disk, it finds with that value. Lastly, last reference time is used for least recently used algorithm.

```cpp
// virtual page table class
class PageTable
{
public:
    PageTable ();
    virtual ~PageTable ();

    // getters
    virtual int getNumProcesses () = 0;
    virtual size_t getNumEntries (int processId) = 0;
    virtual size_t getPageFrameSize () = 0;
    virtual size_t getPageTableSize () = 0;
    virtual PageTableType getType () = 0;
    virtual PageTableEntry *getEntry (int processId, int pageNumber) = 0;

    // setters
    virtual bool setEntry (int processId, int pageNumber, PageTableEntry *entry) = 0;

    // methods
    virtual void print () = 0;
    virtual bool addEntry (int processId, PageTableEntry *entry) = 0;
    // virtual bool addValue (int processId, int pageNumber, int value) = 0;
};
```

- Page table class is a virtual class which will be derived by inverted page table, and regular page table class.

```
// regular page table class; it only has page table entries for one process
class RegularPageTable : public PageTable
{
friend class MemoryManager;

private:
    size_t numEntries;
    int processId;
    size_t pageFrameSize;  // as integer power of 2
    size_t pageTableSize;  // as integer power of 2
    PageTableType type;

    // page table entries
    PageTableEntry **entries;

public:
    RegularPageTable (int processId, size_t pageFrameSize, size_t pageTableSize);
    ~RegularPageTable ();

    // getters
    int getNumProcesses ();
    size_t getNumEntries (int processId);
    size_t getPageFrameSize ();
    size_t getPageTableSize ();
    PageTableType getType ();
    PageTableEntry *getEntry (int processId, int pageNumber);

    // setters
    bool setEntry (int processId, int pageNumber, PageTableEntry *entry);

    // methods
    void print ();
    bool addEntry (int processId, PageTableEntry *entry);
    bool addEntry (int processId, const size_t *pageFrameNumber, size_t *entryNumber);

    // bool addValue (int processId, int pageNumber, int value);
};
```

- Regular page table class is derived from the page table class, and it is managed by memory manager class. It is for one process, and it holds array of entry pointers. There are various set and get class.

```cpp
class InvertedPageTable : public PageTable
{
friend class MemoryManager;

private:
    int numProcesses;
    size_t pageFrameSize;  // as integer power of 2
    PageTableType type;
    size_t pageTableSize;  // as integer power of 2

    // page table entries for each process, <processId, list of entries>
    std::unordered_map<int, std::vector<PageTableEntry *>> *entries;

public:
    InvertedPageTable (size_t pageFrameSize, size_t pageTableSize);
    ~InvertedPageTable ();

    // getters
    int getNumProcesses ();
    size_t getNumEntries (int processId);
    size_t getPageFrameSize ();
    size_t getPageTableSize ();
    PageTableType getType ();
    PageTableEntry *getEntry (int processId, int pageNumber);

    // setters
    bool setEntry (int processId, int pageNumber, PageTableEntry *entry);

    // methods
    void print ();

    PageTableEntry *firstAvailableEntry (int processId);
    bool addEntry (int processId, PageTableEntry *entry);
    // bool addValue (int processId, int pageNumber, int value);
};
```

- I designed inverted table, but it is uncomplete. It is mapped as (n, entry) n is process id. Since in our simulation there is only one process, this table is just a simple linked list.

```cpp
class MMU
{
private:

    // different tlbs for each process
    std::unordered_map<int, std::vector<PageTableEntry *>> *tlbs;

    size_t pageFrameSize;  // as integer power of 2
    size_t tlbSize;

public:
    MMU (size_t pageFrameSize, size_t tlbSize);
    ~MMU ();

    // getters
    size_t getPageFrameSize ();
    size_t getTlbSize ();

    // setters
    void setPageFrameSize (size_t pageFrameSize);
    void setTlbSize (size_t tlbSize);

    // methods
    bool addEntry (int processId, PageTableEntry *entry);
    PageTableEntry *getEntry (int processId, int pageNumber);
    bool setEntry (int processId, int pageNumber, PageTableEntry *entry);

    bool addressTranslation (int processId, const virtualAddress *virtualAddress, physicalAddress
};
```

- This is an MMU, class I couldn't implement TLB but it was in my initial design for page replacement. I used MMU for address translation.

```cpp
class MemoryManager
{
private:
    PhysicalMemory *physicalMemory;

    // page tables for each process, if it is inverted, then there is only one page table
    PageTable **pageTables;

    // memory management unit
    MMU *mmu;

    int numProcesses;
    int maxProcesses;
    size_t maxPageFrameNumber;

    // input parameters
    std::string diskFileName;
    bool invertedFlag;
    size_t printCount;
    PageAlgorithm pageAlgorithm;
    size_t frameSize;  // as integer power of 2
    size_t physicalMemorySize;
    size_t pageTableSize;

    // statistics
    size_t numReads;
    size_t numWrites;
    size_t numPageMisses;
    size_t numPageReplacements;
    size_t numDiskWrites;
    size_t numDiskReads;
```

```
    // page algorithm variables
    size_t SCindex; // second chance index
    size_t WSCLOCKindex; // wsclock index
    // threshold for wsclock, it is compared with (current time - last reference time)
    std::chrono::duration<double> threshold;

    // estimated working set w()

ublic:
    MemoryManager (int maxProcesses, size_t frameSize, size_t physicalMemorySize, size
                    bool invertedFlag, size_t printCount, std::string diskFileName);
    ~MemoryManager ();

    // getters
    std::string getDiskFileName ();
    bool getInvertedFlag ();
    size_t getPrintCount ();
    PageAlgorithm getPageAlgorithm ();
    int getNumProcesses ();
    size_t getFrameSize ();
    size_t getPhysicalMemorySize ();
    size_t getPageTableSize ();
    size_t getPageMissCount ();
    // methods

    // // initalize the memory manager for a process
    // void initializeProcess (int processId);

    // allocate memory for a process, return true if successful
    bool allocateMemory (int processId, virtualAddress *virtualAddress);
    // bool allocateMemory (int processId, const size_t *size, virtualAddress *virtual
```

- It has several getters, I did not put setter for memory manager because, its variables should not be changed by outside. Before writing anything, memory space must be allocated beforehand. It takes virtual address as a parameter, and returns the allocated address.

```
    // allocate memory for a process, return true if successful
    bool allocateMemory (int processId, virtualAddress *virtualAddress);
    // bool allocateMemory (int processId, const size_t *size, virtualAddress *virtualAddress);

    // deallocate memory for a process, return true if successful
    // bool freeMemory (int processId, const size_t* free_size, const size_t *start_address);

    // // read a value from memory, return true if successful
    bool readMemory (int processId, const virtualAddress *address, int *value);

    // // write a value to memory, return true if successful
    bool writeMemory (int processId, const virtualAddress *address, int value);

    // page replacement algorithms
    bool secondChance1 (int processId, RegularPageTable *pageTable, PageFrame *frame, size_t* ent
    bool lru1 (int processId, RegularPageTable *pageTable, PageFrame *frame, size_t* entryNum, si
    bool wsclock1 (int processId, RegularPageTable *pageTable, PageFrame *frame, size_t* entryNum

    bool writePageFrameToDisk (PageFrame *frame, size_t *diskLine);
    bool overwriteAndReadFromDisk (PageFrame *newFrame, PageFrame *oldFrame, size_t *diskLine);

    // print the memory manager
    void printStatistics ();
    void resetStatistics ();
    void printPageTableInfo (int processId);
```

- This class have functionalities for allocating, writing, and reading. It has page replacement algorithm implementations. I held in duration in this class for WSClock algorithm. I defined it as 10 ms.

## 2- Part 1

```
// page replacement algorithms
bool secondChance1 (int processId, RegularPageTable *pageTable, PageFrame *frame, size_t* entryNum, size_t *diskPos, ReplaceMode mode)
bool lru1 (int processId, RegularPageTable *pageTable, PageFrame *frame, size_t* entryNum, size_t *diskPos, ReplaceMode mode);
bool wsclock1 (int processId, RegularPageTable *pageTable, PageFrame *frame, size_t* entryNum, size_t *diskPos, ReplaceMode mode);
```

- Page replacement algorithms are implemented in memory manager class. I added fields to page table entry, for those algorithms.

Second Chance: Present bit is enough for second chance algorithm. Also I added SCindex to memory manager, to remember where it is left off.

LRU: I added last time of use field to page table entry. I updated that field at every access to that frame.

WSCLOCK: I added threshold value to memory manager class, and it is used for checking difference with current time – last referenced.

```
if (entry->isReferenced())
{
    // set referenced bit to 0
    entry->setReferenced(false);
    // set the last reference time
    entry->lastReferenceTime = currentTime;
}
else
{
    bool cond = (currentTime - entry->lastReferenceTime) > this->threshold;

    if (cond)
    {
        // page to be replaced is found
        // set entry number
        *entryNumber = entry->getPageNumber();
        flag = true;

        WSCLOCKindex = (i + 1) % pageTable->getPageTableSize();
        break;
    }

    else
    {
        // check if it is the smallest duration
        if ((currentTime - entry->lastReferenceTime) < minDuration)
        {
            // set minDuration
            minDuration = currentTime - entry->lastReferenceTime;
            // set entry number
            *entryNumber = entry->getPageNumber();
            flag = true;
        }
    }
```

- Cases for wsclock.

```
bool flag = false;
// traverse all present entries in the page table
for (size_t i = 0; i < pageTable->getPageTableSize(); ++i)
{
    PageTableEntry* entry = pageTable->getEntry(processId, i);
    if (entry->isPresent())
    {
        // check if its time is less than minTime
        if (entry->lastReferenceTime < minTime)
        {
            // set minTime
            minTime = entry->lastReferenceTime;
            // set entry number
            *entryNumber = i;
            flag = true;
        }
    }
}
```

- Cases for determining page to be replaced in LRU. Choose the page, that is the oldest.

```
        {
            // check if it is referenced
            if (entry->isReferenced())
            {
                // set referenced bit to 0
                entry->setReferenced(false);

                // increment SCindex
                SCindex = (SCindex + 1) % pageTable->getPageTableSize();
```

- Second chance, case 1: it is referenced. If it is not referenced replace it.

## 3- Part 2

- I used pthread library in c for parallelism. In this part, I performed the wanted computations with given frame values. I used alloc, read, and write. It might be too slow if virtual memory is huge, and it is filled with random integers. So, I commented out that filling part.

```
// virtual address pointers for matrix
virtualAddress** matrix = new virtualAddress*[matrixRow];
for (size_t i = 0; i < matrixRow; i++)
{
    matrix[i] = new virtualAddress[matrixCol];
    if (!matrix[i])
    {
        std::cerr << "Error: could not allocate memory" << std::endl;
        return;
    }
}
```

- I held virtual address pointers in order to access my memory. This is the example for matrix.

```
// allocate each virtual address in matrix
for (size_t i = 0; i < matrixRow; i++)
{
    for (size_t j = 0; j < matrixCol; j++)
    {
        if (!memoryManager->allocateMemory(0, &matrix[i][j]))
        {
            std::cerr << "Error: could not allocate memory" << std::endl;
            return;
        }
    }
}
```

- This is the memory allocation for matrix. I send the virtual address reference, and it is filled by memory manager. If there are page fault or anything all is handled by memory manager.

```cpp
    // memory full, page replacement
    this->numPageMisses++;
    if (pageAlgorithm == PageAlgorithm::SC)
    {
            size_t diskPos;
            // std::cout << "second chance" << std::endl;
            if(!secondChance1(processId, pageTable, frame, &entryNumber, &diskPos, ReplaceMode::APPEND))
                return false;

            // std::cout << "second chance done" << std::endl;
            // write to virtual address
            virtual_address->pageTableEntryNumber = entryNumber;
            virtual_address->valueOffset = 0;
            virtual_address->pageFrameSize = frameSize;
            virtual_address->pageTableSize = pageTableSize;

            this->numPageReplacements++;
            frame->addValue(0);
            // std::cout << "virtual address: " << std::endl;
            // virtual_address->print();
            // std::cout << std::endl;
            return true;
    }
```

- This is the part in memory allocation for page fault handling with second chance line: 1280

```cpp
std::cout << "\nGenerating matrix" << std::endl;
// fill matrix with random generated integers
for (size_t i = 0; i < matrixRow; i++)
{
    for (size_t j = 0; j < matrixCol; j++)
    {
        int val = distribution(generator);
        // if (!memoryManager->allocateMemory(0, &matrix[i][j]))
        // {
        //     std::cerr << "Error: could not allocate memory" << std::endl;
        //     return;
        // }
        if (!memoryManager->writeMemory(0, &matrix[i][j], val))
        {
            std::cerr << "Error: could not write to memory" << std::endl;
            return;
        }
    }
}
std::cout << "Matrix size: " << matrixRow << "x" << matrixCol << " generated" << std::endl;
```

- Writing random generated values to the memory using addresses.

```cpp
    pthread_t threads[numThreads];

    ThreadMatrixVectorMultiplyArgs args[numThreads];

    size_t start = 0;
    size_t end = 0;

    size_t step = matrixRow / numThreads;

    for (size_t i = 0; i < numThreads; i++)
    {
        start = i * step;
        end = (i + 1) * step;

        if (i == numThreads - 1)
        {
            // if it's the last thread, assign the remaining work
            end = matrixRow;
        }

        // std::cout << "start: " << start << " end: " << end << std::endl;
        args[i].matrix = matrix;
        args[i].vector = vector;
        args[i].result = result;
        args[i].matrixRow = matrixRow;
        args[i].matrixCol = matrixCol;
        args[i].vectorSize = vectorSize;
        args[i].start = start;
        args[i].end = end;

        pthread_create(&threads[i], NULL, threadMatrixVectorMultiply, (void*)&args[i]);
    }
```

- This is the matrix multiplication function, it uses threads for that operation. As it can be seen, different parts of matrix is computed by different threads.

```
for (size_t i = 0; i < numThreads; i++)
{
    start = i * step;
    end = (i + 1) * step;

    if (i == numThreads - 1)
    {
        // if it's the last thread, assign the remaining work
        end = summationSize;
    }

    args[i].start = start;
    args[i].end = end;
    args[i].result1 = result1;
    args[i].result2 = result2;
    args[i].summation = summation;
    args[i].vectorSize = vectorSize;
    args[i].matrixRow = matrixRow;

    pthread_create(&threads[i], NULL, threadSummationMatrixVector, (void*)&args[i]);
}

// wait for threads to finish
for (size_t i = 0; i < numThreads; i++)
{
    pthread_join(threads[i], NULL);
}
```

- This is the summation that is calculated with threads again.

```
// copy vector to transposed vector
for (size_t i = 0; i < vectorSize; i++)
{
    int val = -1;
    if (!memoryManager->readMemory(0, &vector[i], &val))
    {
        std::cerr << "Error: could not read memory" << std::endl;
        return false;
    }
    if (!memoryManager->writeMemory(0, &transposedVector[i], val))
    {
        std::cerr << "Error: could not write to memory" << std::endl;
        return false;
    }
}
```

- Copying the vector value to its transpose, they have different memory regions in memory. It makes read and write.

```
bool binaryAndLinearSearch (virtualAddress* vector, size_t vectorSize, int searchVal, int *index1, int *index2)
{

    // make binary search and linear search in parallel

    // sort the vector

    merge_sort(vector, 0, vectorSize - 1);

    // create threads

    pthread_t threads[2];

    // binary search
    ThreadBinarySearchArgs args1;
    args1.vector = vector;
    args1.vectorSize = vectorSize;
    args1.searchVal = searchVal;
    args1.index = index1;

    // liner search
    ThreadLinearSearchArgs args2;
    args2.vector = vector;
    args2.vectorSize = vectorSize;
    args2.searchVal = searchVal;
    args2.index = index2;

    pthread_create(&threads[0], NULL, threadBinarySearch, (void*)&args1);
    pthread_create(&threads[1], NULL, threadLinearSearch, (void*)&args2);

    // wait for threads to finish
    for (size_t i = 0; i < 2; i++)
    {
        pthread_join(threads[i], NULL);
    }
```

- This is binary and linear search, but I sorted the array before doing it. Again searches are done in parallel.

- For **backing store**, as I said earlier, I stored the unused page frames on disk. If a page is in memory, it is not stored in disk, it is written to disk when it needs to be replaced.

```
Process Id: 0
Frame Number: 1009
Frame Size: 128
Number of Values: 128
Full: 1
355
402
167
409
446
385
385
347
168
101
183
15
270
424
196
30
246
166
198
218
417
87
241
304
305
270
460
343
482
13
55
```

- For example, page frame 0 is stored in disk like that, if it is referenced again it will be retrieved.

```
bool writePageFrameToDisk (PageFrame *frame, size_t *diskLine);
bool overwriteAndReadFromDisk (PageFrame *newFrame, PageFrame *oldFrame, size_t *diskLine);
```

- I used those functions for writing, and reading from disk. In the page table entry, there is a field for holding disk offset. If page from the disk is referenced it is accessed by that field. All the values etc., will be read from the disk.

Example run: (random fill is not commented out)

```
g++ -Wall -Wextra -std=c++11 -pthread -o part3 obj/memoryManagement.o obj/part3.o obj/util.o
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse312/homework assignments/HW2/hw2$ ./operateArrays 7 4 10 LRU 10000 diskFileName.dat
Test 1
frame size: 128
# physical frames: 16
# virtual frames: 1024
page algorithm: LRU
disk file name: diskFileName.dat

Filling virtual memory with random values

Page Table Info
Number of Pages in Memory: 16
Number of Pages in Disk: 102

Number of Reads: 0
Number of Writes: 9999
Number of Page Misses: 102
Number of Page Replacements: 102
Number of Disk Reads: 0
Number of Disk Writes: 102


Page Table Info
Number of Pages in Memory: 16
Number of Pages in Disk: 180

Number of Reads: 0
Number of Writes: 19999
Number of Page Misses: 180
Number of Page Replacements: 180
Number of Disk Reads: 0
Number of Disk Writes: 180
```

```
Page Table Info
Number of Pages in Memory: 16
Number of Pages in Disk: 258

Number of Reads: 0
Number of Writes: 29999
Number of Page Misses: 258
Number of Page Replacements: 258
Number of Disk Reads: 0
Number of Disk Writes: 258


Page Table Info
Number of Pages in Memory: 16
Number of Pages in Disk: 336

Number of Reads: 0
Number of Writes: 39999
Number of Page Misses: 336
Number of Page Replacements: 336
Number of Disk Reads: 0
Number of Disk Writes: 336
```

it continues like that

```
Generating vector
Vector size: 3 generated

Generating matrix
Matrix size: 1000x3 generated

Calculating M * V

Page Table Info
Number of Pages in Memory: 16
Number of Pages in Disk: 1008

Number of Reads: 805
Number of Writes: 129194
Number of Page Misses: 1058
Number of Page Replacements: 1058
Number of Disk Reads: 50
Number of Disk Writes: 1058
```

```
V * M is calculated

Calculating V * V^t

V * V^t is calculated

Calculating V * M + V * V^t

V * M + V * V^t is calculated
Search value: 239043

Making binary and linear search in parallel

Page Table Info
Number of Pages in Memory: 16
Number of Pages in Disk: 1008

Number of Reads: 8927
Number of Writes: 131072
Number of Page Misses: 1192
Number of Page Replacements: 1192
Number of Disk Reads: 184
Number of Disk Writes: 1192
```

```
Page Table Info
Number of Pages in Memory: 16
Number of Pages in Disk: 1008

Number of Reads: 18927
Number of Writes: 131072
Number of Page Misses: 1192
Number of Page Replacements: 1192
Number of Disk Reads: 184
Number of Disk Writes: 1192

Binary search result: 350
Linear search result: 350

Number of Reads: 24761
Number of Writes: 131072
Number of Page Misses: 1192
Number of Page Replacements: 1192
Number of Disk Reads: 184
Number of Disk Writes: 1192
```

(result statistics)

## 4- Part 3

- I did it for merge and quick sort, generally greater frame size is worked better. But I cannot properly test it. I tried with smaller values. Those are the page miss numbers.

```cpp
// assign maximum value of size_t
size_t minPageMisses = std::numeric_limits<size_t>::max();

std::cout << "Starting tests..." << std::endl;

std::cout << "Testing with merge sort" << std::endl;
// try frame sizes 3 up to 20, for merge sort
for (size_t i = 3; i <= 7; i++)
{
    std::cout << "Testing with frame size: " << i << std::endl;
    memoryManager = new MemoryManager (1, i, physicalMemorySize, virtualMemorySize, PageAlgorithm::SC, inverted, printCount, diskFileName)

    size_t result = merge_sort();
    if (result < minPageMisses)
    {
        minPageMisses = result;
        frameSize = i;
    }
}
delete memoryManager;
```

```cpp
// assign maximum value of size_t
minPageMisses = std::numeric_limits<size_t>::max();

std::cout << "Testing with quick sort" << std::endl;
// try frame sizes 3 up to 20, for quick sort
for (size_t i = 3; i <= 7; i++)
{
    std::cout << "Testing with frame size: " << i << std::endl;
    memoryManager = new MemoryManager (1, i, physicalMemorySize, virtualMemorySize, PageAlgorithm::SC, inverted, printCount, diskFileName)

    size_t result = quick_sort();
    if (result < minPageMisses)
    {
        minPageMisses = result;
        frameSize = i;
    }
}
delete memoryManager;

std::cout << "Best frame size for quick sort: " << frameSize << std::endl;
```

```
Starting tests...
Testing with merge sort
Testing with frame size: 3
Merge sort page misses: 3766
Testing with frame size: 4
Merge sort page misses: 2412
Testing with frame size: 5
Merge sort page misses: 1460
Testing with frame size: 6
Merge sort page misses: 568
Testing with frame size: 7
Merge sort page misses: 25
Best frame size for merge sort: 7
Testing with quick sort
Testing with frame size: 3
Quick sort page misses: 7141
Testing with frame size: 4
Quick sort page misses: 6196
Testing with frame size: 5
Quick sort page misses: 4560
Testing with frame size: 6
Quick sort page misses: 2738
Testing with frame size: 7
Quick sort page misses: 31
Best frame size for quick sort: 7
```

## 6 – Test Results (statistics)

```
cse312@ubuntu:~/Desktop/cse312/hw2/hw2$ ./operateArrays 6 3 7 LRU 10000 diskFile
Name.dat
Test 1
frame size: 64
# physical frames: 8
# virtual frames: 128
page algorithm: LRU
disk file name: diskFileName.dat

Filling virtual memory with random values

Generating vector
Vector size: 3 generated

Generating matrix
Matrix size: 1000x3 generated

Calculating M * V

Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 3272
Number of Writes: 6727
Number of Page Misses: 232
Number of Page Replacements: 232
Number of Disk Reads: 112
Number of Disk Writes: 232
```

```
V * M is calculated

Calculating V * V^t

V * V^t is calculated

Calculating V * M + V * V^t

V * M + V * V^t is calculated
Search value: 111674

Making binary and linear search in parallel

Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 11807
Number of Writes: 8192
Number of Page Misses: 329
Number of Page Replacements: 329
Number of Disk Reads: 209
Number of Disk Writes: 329


Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120
```

```
Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 21807
Number of Writes: 8192
Number of Page Misses: 409
Number of Page Replacements: 409
Number of Disk Reads: 289
Number of Disk Writes: 409

Binary search result: 378
Linear search result: 378

Number of Reads: 24812
Number of Writes: 8192
Number of Page Misses: 1178
Number of Page Replacements: 1178
Number of Disk Reads: 1058
Number of Disk Writes: 1178
```

( LRU results)

```
cse312@ubuntu:~/Desktop/cse312/hw2/hw2$ ./operateArrays 6 3 7 SC 10000 diskFileName.dat
Test 1
frame size: 64
# physical frames: 8
# virtual frames: 128
page algorithm: SC
disk file name: diskFileName.dat

Filling virtual memory with random values

Generating vector
Vector size: 3 generated

Generating matrix
Matrix size: 1000x3 generated

Calculating M * V

Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 3272
Number of Writes: 6727
Number of Page Misses: 217
Number of Page Replacements: 217
Number of Disk Reads: 97
Number of Disk Writes: 217
```

```
V * M is calculated

Calculating V * V^t

V * V^t is calculated

Calculating V * M + V * V^t

V * M + V * V^t is calculated
Search value: 165230

Making binary and linear search in parallel

Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 11807
Number of Writes: 8192
Number of Page Misses: 332
Number of Page Replacements: 332
Number of Disk Reads: 212
Number of Disk Writes: 332
```

```
Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 21807
Number of Writes: 8192
Number of Page Misses: 471
Number of Page Replacements: 471
Number of Disk Reads: 351
Number of Disk Writes: 471

Binary search result: 735
Linear search result: 735

Number of Reads: 25178
Number of Writes: 8192
Number of Page Misses: 1558
Number of Page Replacements: 1558
Number of Disk Reads: 1438
Number of Disk Writes: 1558
```

- Second chance results.

```
cse312@ubuntu:~/Desktop/cse312/hw2/hw2$ ./ope
Test 1
frame size: 64
# physical frames: 8
# virtual frames: 128
page algorithm: WSCLOCK
disk file name: diskFileName.dat

Filling virtual memory with random values

Generating vector
Vector size: 3 generated

Generating matrix
Matrix size: 1000x3 generated

Calculating M * V

Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 3270
Number of Writes: 6729
Number of Page Misses: 227
Number of Page Replacements: 227
Number of Disk Reads: 107
Number of Disk Writes: 227
```

```
Calculating M * V

Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 3270
Number of Writes: 6729
Number of Page Misses: 227
Number of Page Replacements: 227
Number of Disk Reads: 107
Number of Disk Writes: 227


V * M is calculated

Calculating V * V^t

V * V^t is calculated

Calculating V * M + V * V^t

V * M + V * V^t is calculated
Search value: 196336

Making binary and linear search in parallel

Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 11807
Number of Writes: 8192
Number of Page Misses: 355
Number of Page Replacements: 355
Number of Disk Reads: 235
Number of Disk Writes: 355
```

```
Page Table Info
Number of Pages in Memory: 8
Number of Pages in Disk: 120

Number of Reads: 21807
Number of Writes: 8192
Number of Page Misses: 509
Number of Page Replacements: 509
Number of Disk Reads: 389
Number of Disk Writes: 509

Binary search result: 425
Linear search result: 425

Number of Reads: 24939
Number of Writes: 8192
Number of Page Misses: 1420
Number of Page Replacements: 1420
Number of Disk Reads: 1300
Number of Disk Writes: 1420
```

- WSCLOCK results.

LRU gave the best results.