PART1 REPORT

In this part, a page table structure is designed that makes it possible to implement Not-Recently Used(NRU), FIFO, Second-chance(SC), Least-Recently-Used(LRU), Working set clock(WSClock) algorithms.

Pyhsical memory(Figure 1) holds data as pages. Pyhsical memory is very fast but it is expensive. It must be used efficiently. Therefore, virtual memory addresses are used. In my design, there is a disk file covering all the virtual memory. Like physical memory, virtual memory is in the form of pages. Page table is used because virtual memory can be much larger than physical memory.

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Page table structure provides more memory to be used.

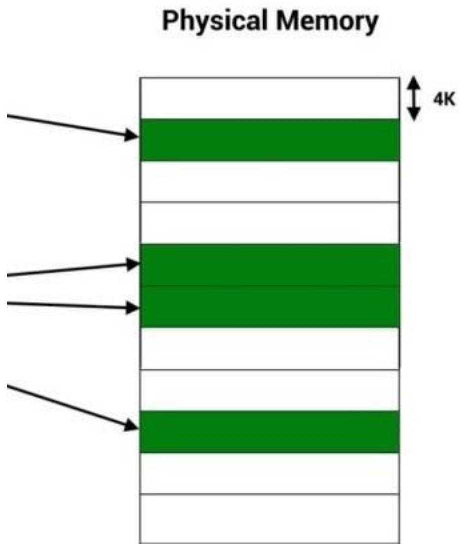

Figure 1

In part2, page table was implemented. Page table consists of many pages. A page structure has some integers(Figure 2). 4 integer occupies 16 bytes. So page table entry size becomes 16 bytes.

‘targetMemoryFrame’ is page frame number that pointed to physical memory page.

‘prAbBit’ is present/absent bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault. In this situation, page replacement algorithms are run. We will discuss page replacement.

‘modifyBit’ is modified bit. When a page is written to, automatically sets the ‘Modified’ bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified(value becomes 1), it must be written back to the disk. If it has not been modified(value becomes 0), it can just be abandoned, since the disk copy is still valid.

‘refBit’ is reference bit. It is set whenever a page is referenced, either for reading or for writing. Its value is used to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are far better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms.

```c

struct pageEntry{
    int targetMemoryFrame;
    int prAbBit;
    int modifyBit;
    int refBit;
};
struct pageEntry * pageTable;

```

Figure 2

In fact, programs are completely isolated from the memory structure. For example, memory frame is 3 bits, virtual frame 5 bits, frame size is 2 bits. Frame size becomes 4(2 over 2). Memory has 8 page(2 over 3), and each page holds 4 integers. Memory can holds 8 * 4 = 32 integers. But virtual memory 32 page(2 over 5) and each virtual page holds 4 integer. Programs only know that virtual memory 32 * 4 = 128 integers. In background this 128 number is replaced by 32 numbers. Let's look at this logic.

Programs only know get and set functions for accessing to the memory(part2, sortArray.c, 1000s lines). When the program tries to get the 50th integer, it is found which page the number is on. İndex / framesize gives us 12. It is looked 12th page of page table. If present/absent bit of this page is 1, there is not problem, this page already in physical memory. But present/absent bit is 0, page fault is ocurred. It must be replaced with a page with present/absent bit 1. Page replacement algorithms decided which page to replace. Each algorithm uses a different logic for replacement(implementation of pagereplacement algorithms in line 540). In addition, details of these are available in the part2 report.

If we continue from the example, page 5 let's make a logical choice and change it. The target address indicated by page 5 becomes the target address of page 12. All page 12 in virtual memory is moved to the target address in physical memory. Then the 3rd index of page 12 in physical memory is returned. However, it should not be forgotten that at this point, the data of the page being replaced will be lost. Modify bit takes effect at this point. If modify bit of page 5 is 1, it means something change, so the page in physical memory must be rewritten to disk before replacement. If it is 0, no change is necessary because the page in physical memory and virtual memory are exactly the same.

Figure 3 below shows this structure.

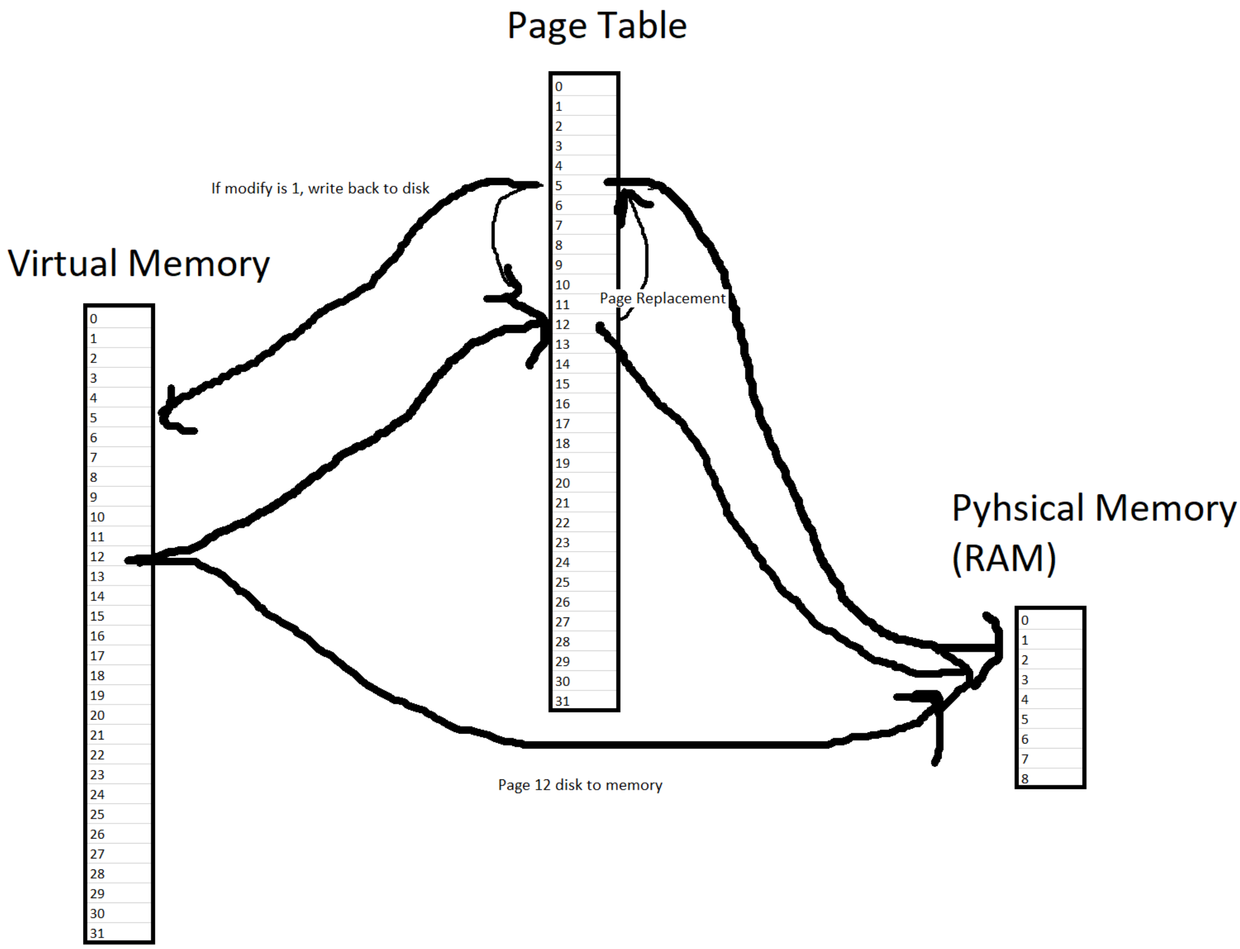

Figure 3

A similar operation is applied when an integer value in the memory is wanted to be changed(Figure 3). The set function sets the value by index. After the page operations are done and the right place is found, sets the value in index of target page.

What do this processes save us even if the disk is read and write?

In the example 50th number was not found in physical memory but if the program needs number 51, page 12 pre/abs bit is 1, so there is no operations required. That's why algorithms that determine which page to switch are so important(the least needed page should be replaced).

Part2 Report

In part2, disk and memory are created according to ‘sortArrays frameSize numPhysical numVirtual pageReplacement allocPolicy pageTablePrintInt diskFileName.dat’. pageReplacement is the page algorithm, allocPolicy is allocation policy(global or local), pageTablePrintInt is the interval of memory access after which the page table is printed on screen, numVirtual and numPhysical are the total number of page frames in the pyschical memory and virtual adress space.

Set function(Figure 4) set a value by index and thread name. Each threads work on their own virtual part. If thread name is ‘fill’, set function directly writes value by index and update numDiskPageWrites statistics. If thread name is different, function looks at the page table. If present/absent bit is 1(means that there is a target memory frame in this page), the value in the memory is changed and modify bit is updated. If present/absent bit is 0(means that there is not a target memory frame in this page), pageReplacement function calls and outcoming pyhsical adress is occured. Anymore, target memory frame is specified. The value in the memory is changed. Modify bits and flag is updated. Flag is used for some statistics updates(Figure 4).

```c
void set(unsigned int index, int value, char * tName){          // set function sets a value to specifying index
    int pageReplaceFlag = 0;
    if( strcmp(tName, "fill") == 0){
        fseek(fp, sizeof(int) * index, SEEK_SET);               // if thread is fill, add value to disk directly
        fwrite(&value, sizeof(int), 1, fp);
        fillStatic.numDiskPageWrites = fillStatic.numDiskPageWrites + 1;
    }
    else if( strcmp(tName, "bubble") == 0  || strcmp(tName, "quick") == 0
            || strcmp(tName, "merge") == 0|| strcmp(tName, "index") == 0){

        struct pageEntry pageTemp = pageTable[index / frameSize];
        if( pageTemp.prAbBit == 0 ){        // if prAbbit is zero, call page replacement
            pageReplace(index, tName);
            physicalMemory[ ((pageTable[index / frameSize].targetMemoryFrame) * frameSize) + (index % frameSize)] = value;  // add to physical memory
            pageTable[index / frameSize].modifyBit = 1;
            pageReplaceFlag = 1;
        }
        else if( pageTemp.prAbBit == 1 ){   // if prAbbit is one, add directly to target adress
            pageTemp.refBit = 1;
            int ramFrame = pageTemp.targetMemoryFrame;
            int ramAdress = ( ramFrame * frameSize ) + (index % frameSize);
            physicalMemory[ramAdress] = value;
            pageTable[index / frameSize].modifyBit = 1;
        }
    }
    if( strcmp(tName, "fill") == 0 ){           // incremenet some statics according to page replacement is happened or not
        fillStatic.numWrites = fillStatic.numWrites + 1;
        if(pageReplaceFlag == 1){
            fillStatic.numPageReplace = fillStatic.numPageReplace + 1;
            fillStatic.numPageMiss = fillStatic.numPageMiss + 1;
        }
    }
    else if( strcmp(tName, "bubble") == 0 ){
        bubbleStatic.numWrites = bubbleStatic.numWrites + 1;
        if(pageReplaceFlag == 1){
            bubbleStatic.numPageReplace = bubbleStatic.numPageReplace + 1;
            bubbleStatic.numPageMiss = bubbleStatic.numPageMiss + 1;
        }
    }
    else if( strcmp(tName, "quick") == 0 ){
        quickStatic.numWrites = quickStatic.numWrites + 1;
        if(pageReplaceFlag == 1){
            quickStatic.numPageReplace = quickStatic.numPageReplace + 1;
            quickStatic.numPageMiss = quickStatic.numPageMiss + 1;
        }
    }
    else if( strcmp(tName, "merge") == 0 ){
        mergeStatic.numWrites = mergeStatic.numWrites + 1;
        if(pageReplaceFlag == 1){
```

Figure 4

Get function(Figure 5) get a value by its index. In fact, get and set functions are not very different from each other. Get function looks present/absent bit, too. If it is 1, directly gets value from memory. If it is 0, calls page replacement function and then reads value by index from memory, returns value. Function updates statistics according to thread name and flag. In addition, every time get and set functions are called, memory access variable is increased. It is used for print table.

```
int get(unsigned int index, char * tName){              // this function get a value by index
    int returnedValue;
    int pageReplaceFlag = 0;
    struct pageEntry pageTemp = pageTable[index / frameSize];
    if( pageTemp.prAbBit == 0 ){
        pageReplace(index, tName);          // page replace were happened
        returnedValue =  physicalMemory[ ((pageTable[index / frameSize].targetMemoryFrame) * frameSize) + (index % frameSize)];
        pageReplaceFlag = 1;
    }
    else if( pageTemp.prAbBit == 1 ){
        pageTemp.refBit = 1;
        int ramFrame = pageTemp.targetMemoryFrame;
        int ramAdress = ( ramFrame * frameSize ) + (index % frameSize);
        returnedValue = physicalMemory[ramAdress];      // directly access
    }
    if( strcmp(tName, "fill") == 0 ){           // some statics are necessary
        fillStatic.numReads = fillStatic.numReads + 1;
        if(pageReplaceFlag == 1){
            fillStatic.numPageReplace = fillStatic.numPageReplace + 1;
            fillStatic.numPageMiss = fillStatic.numPageMiss + 1;
        }
    }
    else if( strcmp(tName, "bubble") == 0 ){
        bubbleStatic.numReads = bubbleStatic.numReads + 1;
        if(pageReplaceFlag == 1){
            bubbleStatic.numPageReplace = bubbleStatic.numPageReplace + 1;
            bubbleStatic.numPageMiss = bubbleStatic.numPageMiss + 1;
        }
    }
    else if( strcmp(tName, "quick") == 0 ){
        quickStatic.numReads = quickStatic.numReads + 1;
        if(pageReplaceFlag == 1){
            quickStatic.numPageReplace = quickStatic.numPageReplace + 1;
            quickStatic.numPageMiss = quickStatic.numPageMiss + 1;
        }
    }
    else if( strcmp(tName, "merge") == 0 ){
        mergeStatic.numReads = mergeStatic.numReads + 1;
        if(pageReplaceFlag == 1){
            mergeStatic.numPageReplace = mergeStatic.numPageReplace + 1;
            mergeStatic.numPageMiss = mergeStatic.numPageMiss + 1;
        }
    }
    else if( strcmp(tName, "index") == 0 ){
        indexStatic.numReads = indexStatic.numReads + 1;
        if(pageReplaceFlag == 1){
            indexStatic.numPageReplace = indexStatic.numPageReplace + 1;
            indexStatic.numPageMiss = indexStatic.numPageMiss + 1;
```

Figure 5

Let’s look at page replacement algorithms. First of all, we should say that page replacement algorithms are replacing of pages but first pyhsical memory pages was must be in page table. Because page replacement is replacement of existing pages(this is done in main before the thread calling in main).

In fifo algorithm, there are global fifo variables for bubble, quick, merge, index, check(Figure 6). There are upper and lower limits according to thread names. Fifo must not turns in this limits. If given index exceeds the limits, it is a wrong access, prints an error message and return -1. If allocation policy is ‘global’, limits are extreme(0 and last page entry). If allocation policsy is ‘local’, limits are already specified. Function looks present/absent bit of fifo page. If pr/ab bit is 0, continues until pr/ab bit is 1. If modify bit is 1, it means page was changed. This page must write back to disk. Target page memory frame is taken by page table entry that contains index. Reads disk and write to physical memory. Fifo variable is increased(Figure 7).

```c
int writeToDiskFlag = 0;
if( replaceAlgorithm == 'F'){   // if replace al
    int lowerLimit;
    int upperLimit;
    int fifoCur;
    if( strcmp(sortAlgo, "bubble") == 0 ){
        lowerLimit = 0;
        upperLimit = virtualFrames / 4;
        fifoCur = fifoBubble;
    }
    else if( strcmp(sortAlgo, "quick") == 0 ){
        lowerLimit = virtualFrames / 4;
        upperLimit = virtualFrames / 2;
        fifoCur = fifoQucik;
    }
    else if( strcmp(sortAlgo, "merge") == 0 ){
        lowerLimit = virtualFrames / 2;
        upperLimit = (virtualFrames / 4) * 3;
        fifoCur = fifoMerge;
    }
    else if( strcmp(sortAlgo, "index") == 0){
        lowerLimit = (virtualFrames / 4) * 3;
        upperLimit = virtualFrames;
        fifoCur = fifoIndex;
    }
    else if( strcmp(sortAlgo, "check") == 0){
        lowerLimit = 0;
        upperLimit = virtualFrames;
        fifoCur = fifoCheck;
    }

    if(allocPoli == 'G'){
        lowerLimit = 0;
        upperLimit = virtualFrames;
    }

    int whichVirtualFrame = (index / frameSize);
    if(fifoCur == 0){
        fifoCur = lowerLimit;
    }

    if( whichVirtualFrame >= upperLimit ){
        printf("%s\n", "Wrong access!!");
        return -1;
    }
```

Figure 6

```c
fifoCur++;                                      // increment fifo
if(fifoCur >= upperLimit){
    fifoCur = lowerLimit;
}

if( pageTable[fifoCur].prAbBit == 0 ){          // if page is zero. find the e
    while(pageTable[fifoCur].prAbBit != 1){
        fifoCur++;
        if(fifoCur >= upperLimit){
            fifoCur = lowerLimit;
        }
        if( pageTable[fifoCur].prAbBit == 1 ){
            break;
        }
    }
}

if( pageTable[fifoCur].modifyBit == 1 ){        // if page is modify
    writeToDiskFlag = 1;
    int targetMemoryFrame = pageTable[fifoCur].targetMemoryFrame;
    for(int i = 0; i < frameSize; i++){
        int value = physicalMemory[targetMemoryFrame*frameSize + i];        //
        fseek(fp, sizeof(int) * ((fifoCur * frameSize) + i), SEEK_SET);
        fwrite(&value, sizeof(int), 1, fp);
    }
}

pageTable[fifoCur].prAbBit = 0;                         // set values
pageTable[fifoCur].modifyBit = 0;
int targetMemoryFrame = pageTable[fifoCur].targetMemoryFrame;
pageTable[whichVirtualFrame].prAbBit = 1;
pageTable[whichVirtualFrame].modifyBit = 0;
pageTable[whichVirtualFrame].targetMemoryFrame = targetMemoryFrame;

for(int j = 0; j < frameSize; j++){                     // page replacement, d
    int value;
    fseek(fp, sizeof(int) * ((whichVirtualFrame * frameSize) + j), SEEK_SET);
    fread(&value, sizeof(int), 1, fp);
    physicalMemory[targetMemoryFrame*frameSize + j] = value;
}

fifoCur++;
if(fifoCur == upperLimit){
    fifoCur = lowerLimit;
}
// end of FIFO
```

Figure 7

In last-recently used algorithm(Figure 8), there are global counters for LRU. There are upper and lower limits according to thread names, too. LRU counter specify the target index. If allocation policy is ‘global’, limits are 0 and last page entry, too. Function looks present/absent bit of fifo page. If pr/ab bit is 0, continues until pr/ab bit is 1. If modify bit is 1, it means page was changed. This page must write back to disk. Target page memory frame is taken by page table entry that contains index. Reads disk and write to physical memory. Different from FIFO, counter specify the last recently used index.

```c
        if(lruCur == 0){
            lruCur = upperLimit;
        }

        if( whichVirtualFrame >= upperLimit ){
            printf("%s\n", "Wrong access!!");
            return -1;
        }

        if( pageTable[lruCur].prAbBit == 0 ){
            while(pageTable[lruCur].prAbBit != 1){
                lruCur--;                       // go in reverse
                if(lruCur < lowerLimit){
                    lruCur = upperLimit;
                }
                if( pageTable[lruCur].prAbBit == 1 ){
                    break;
                }
            }
        }

        if( pageTable[lruCur].modifyBit == 1 ){         // if page is modified
            writeToDiskFlag = 1;
            int targetMemoryFrame = pageTable[lruCur].targetMemoryFrame;
            for(int i = 0; i < frameSize; i++){                 // back up to disk
                int value = physicalMemory[targetMemoryFrame*frameSize + i];
                fseek(fp, sizeof(int) * ((lruCur * frameSize) + i), SEEK_SET);
                fwrite(&value, sizeof(int), 1, fp);
            }
        }

        pageTable[lruCur].prAbBit = 0;
        pageTable[lruCur].modifyBit = 0;
        int targetMemoryFrame = pageTable[lruCur].targetMemoryFrame;
        pageTable[whichVirtualFrame].prAbBit = 1;
        pageTable[whichVirtualFrame].modifyBit = 0;
        pageTable[whichVirtualFrame].targetMemoryFrame = targetMemoryFrame;


        for(int j = 0; j < frameSize; j++){             // disk to ram
            int value;
            fseek(fp, sizeof(int) * ((whichVirtualFrame * frameSize) + j), SEEK_SET);
            fread(&value, sizeof(int), 1, fp);
            physicalMemory[targetMemoryFrame*frameSize + j] = value;
        }
        lruCur--;
        if(lruCur < lowerLimit){
            lruCur = upperLimit;
        }
        // end of LRU
    }
```

Figure 8

Second-chance(Figure 9) gives a second chance to the page to be replaced. When a page is set a second time, reference bit becomes 1 in my algorithm. If function try to replace this page, it is given a second chance, passed it. Because reference bit of this page is 1. The reference bit of this page trying to be changed becomes 0. Function find correct page that is reference bit 0, and replace them. Necessary updates take place. Reading and writing operations are occurred.

```
        if( pageTable[scCur].prAbBit == 1 && pageTable[scCur].refBit == 1 ){
            pageTable[scCur].refBit = 0;
            scCur++;
        }

        if( pageTable[scCur].prAbBit == 0 ){
            while(pageTable[scCur].prAbBit != 1){
                scCur++;
                if(scCur >= upperLimit){
                    scCur = lowerLimit;
                }
                if( pageTable[scCur].prAbBit == 1 ){
                    if( pageTable[scCur].refBit == 1 ){     // give a second chance
                        pageTable[scCur].refBit = 0;
                    }
                    else{
                        break;
                    }
                }
            }
        }
        if( pageTable[scCur].modifyBit == 1 ){      // if modify ram to disk.
            writeToDiskFlag = 1;
            int targetMemoryFrame = pageTable[scCur].targetMemoryFrame;

            for(int i = 0; i < frameSize; i++){             // ram to disk
                int value = physicalMemory[targetMemoryFrame*frameSize + i];
                fseek(fp, sizeof(int) * ((scCur * frameSize) + i), SEEK_SET);
                fwrite(&value, sizeof(int), 1, fp);
            }
        }

        pageTable[scCur].prAbBit = 0;
        pageTable[scCur].modifyBit = 0;
        int targetMemoryFrame = pageTable[scCur].targetMemoryFrame;
        pageTable[whichVirtualFrame].prAbBit = 1;
        pageTable[whichVirtualFrame].modifyBit = 0;
        pageTable[whichVirtualFrame].targetMemoryFrame = targetMemoryFrame;

        for(int j = 0; j < frameSize; j++){             // disk to ram
            int value;
            fseek(fp, sizeof(int) * ((whichVirtualFrame * frameSize) + j), SEEK_SET);
            fread(&value, sizeof(int), 1, fp);
            physicalMemory[targetMemoryFrame*frameSize + j] = value;
        }
        scCur++;
        if(scCur == upperLimit){
            scCur = lowerLimit;
        }
        // end of SECOND-CHANCE(SC)
    }
```

Figure 9

In not recently used(NRU), there are cases(Figure 11). Cases contains reference and pre/abs bits. Cases are examined 0 from 3. When case is provided, passed other cases. Reference bit about page usage. Modify bit about modifying of memory page. timerReset value increases every not recently used operation(Figure 10). If its value is 100, it is an reset timer interrupt(in my design, i prefer this structure). resetRefBits function is called, and this function resets all reference bits of pages in page table. When correct page is found, necessary updates and replacement are done.

```
    else if( replaceAlgorithm == 'N' ){         // NOT RECENTLY USED
        int lowerLimit;
        int upperLimit;
        int nruIndex = -1;

        timerReset++;                           // timerReset variable is used for reset reference bits as  a timer.
        if( timerReset == 100){
            timerReset = 0;
            resetRefBits();
        }

        if( strcmp(sortAlgo, "bubble") == 0 ){
            lowerLimit = 0;
            upperLimit = virtualFrames / 4;
        }
        else if( strcmp(sortAlgo, "quick") == 0 ){
            lowerLimit = virtualFrames / 4;
            upperLimit = virtualFrames / 2;
        }
        else if( strcmp(sortAlgo, "merge") == 0 ){
            lowerLimit = virtualFrames / 2;
            upperLimit = (virtualFrames / 4) * 3;
        }
        else if( strcmp(sortAlgo, "index") == 0){
            lowerLimit = (virtualFrames / 4) * 3;
            upperLimit = virtualFrames;
        }
        else if( strcmp(sortAlgo, "check") == 0){
            lowerLimit = 0;
            upperLimit = virtualFrames;
        }

        if(allocPoli == 'G'){
            lowerLimit = 0;
            upperLimit = virtualFrames;
        }

```

Figure 10

```
        }
        for(int i = lowerLimit; i < upperLimit; i++){
            if( pageTable[i].prAbBit == 1 && pageTable[i].refBit == 0 && pageTable[i].modifyBit == 0 ){          // CASE 0
                nruIndex = i;
                break;
            }
        }
        if(nruIndex == -1){
            for(int i = lowerLimit; i < upperLimit; i++){
                if( pageTable[i].prAbBit == 1 && pageTable[i].refBit == 0 && pageTable[i].modifyBit == 1 ){      // CASE 1
                    nruIndex = i;
                    break;
                }
            }
        }

        if( nruIndex == -1 ){
            for(int i = lowerLimit; i < upperLimit; i++){
                if( pageTable[i].prAbBit == 1 && pageTable[i].refBit == 1 && pageTable[i].modifyBit == 0 ){      // CASE 2
                    nruIndex = i;
                    break;
                }
            }
        }

        if( nruIndex == -1 ){
            for(int i = lowerLimit; i < upperLimit; i++){
                if( pageTable[i].prAbBit == 1 && pageTable[i].refBit == 1 && pageTable[i].modifyBit == 1 ){      // CASE 3
                    nruIndex = i;
                    break;
                }
            }
        }
        if( pageTable[nruIndex].modifyBit == 1 ){       // if modify ram to disk.
            writeToDiskFlag = 1;
            int targetMemoryFrame = pageTable[nruIndex].targetMemoryFrame;
            for(int i = 0; i < frameSize; i++){             // ram to disk
                int value = physicalMemory[targetMemoryFrame*frameSize + i];
                fseek(fp, sizeof(int) * ((nruIndex * frameSize) + i), SEEK_SET);
                fwrite(&value, sizeof(int), 1, fp);
            }
        }

        pageTable[nruIndex].prAbBit = 0;
        pageTable[nruIndex].modifyBit = 0;
        pageTable[nruIndex].refBit = 0;
        int targetMemoryFrame = pageTable[nruIndex].targetMemoryFrame;
        pageTable[whichVirtualFrame].prAbBit = 1;
        pageTable[whichVirtualFrame].modifyBit = 0;
        pageTable[whichVirtualFrame].targetMemoryFrame = targetMemoryFrame;
```

Figure 11

WSClock algorithm(Figure 12) moves clockwise. Like second chance, it is examined reference bit. There is not clock that reset the bits. This operation is provided by itself. Accessing of a page a second time changes reference and modify bits. When correct page is found, necessary updates and replacement are done.

```
        }
        if( pageTable[wsCur].prAbBit == 1 && pageTable[wsCur].refBit == 1 ){
            pageTable[wsCur].refBit = 0;
            wsCur++;
        }
        if( pageTable[wsCur].prAbBit == 0 ){
            while(pageTable[wsCur].prAbBit != 1){
                wsCur++;
                if(wsCur >= upperLimit){
                    wsCur = lowerLimit;
                }
                if( pageTable[wsCur].prAbBit == 1 ){
                    if( pageTable[wsCur].refBit == 1 ){
                        pageTable[wsCur].refBit = 0;
                    }
                    else{
                        break;
                    }
                }
            }
        }
        if( pageTable[wsCur].modifyBit == 1 ){      // if modify ram to disk.
            writeToDiskFlag = 1;
            int targetMemoryFrame = pageTable[wsCur].targetMemoryFrame;
            for(int i = 0; i < frameSize; i++){          // ram to disk
                int value = physicalMemory[targetMemoryFrame*frameSize + i];
                fseek(fp, sizeof(int) * ((wsCur * frameSize) + i), SEEK_SET);
                fwrite(&value, sizeof(int), 1, fp);
            }
        }

        pageTable[wsCur].prAbBit = 0;
        pageTable[wsCur].modifyBit = 0;
        pageTable[wsCur].refBit = 0;
        int targetMemoryFrame = pageTable[wsCur].targetMemoryFrame;
        pageTable[whichVirtualFrame].prAbBit = 1;
        pageTable[whichVirtualFrame].modifyBit = 0;
        pageTable[whichVirtualFrame].targetMemoryFrame = targetMemoryFrame;
        for(int j = 0; j < frameSize; j++){            // disk to ram
            int value;
            fseek(fp, sizeof(int) * ((whichVirtualFrame * frameSize) + j), SEEK_SET);
            fread(&value, sizeof(int), 1, fp);
            physicalMemory[targetMemoryFrame*frameSize + j] = value;
        }
        wsCur++;
        if(wsCur == upperLimit){
            wsCur = lowerLimit;
        }
        // end of WSClock
    }
```

Figure 12

Allocation policy issues were told. If policy is ‘local’, threads work only their memory part. There are 4 sorting algorithms. Each threads can access a quarter area. Upper and lower limits are specified for each threads. If policy is ‘global’, threads can work whole memory but sort their area again. Lower limit should 0 and upper limit should last frame address for global policy.

Other issue is backing store. It means if a page is removal, where is gone. In my design, disk is exactly virtual memory. The page table contains as many elements as there are virtual frames. For example, we have 32(2 over 5) virtual frames, page table entry size is 32. If page 10 should write back to disk, it is written to virtual frame 10(necessary place is 10 * frame-size). However, this is done only if the page's modify bit is 1. If modify bit is 0, means there are no changes on this page. No need to write back to disk because the data is the same between physical memory and virtual memory(disk).

PART 3 REPORT

We have a physical memory that can hold 64K integers and a virtual memory that can hold 1M integers. Our task is finding the optimal page size for each sorting algorithm. In this part, virtual memory was used as a integer C array like physical memory(Figure 13). Reason of this, program takes long with this parameters. FIFO algorithm was used as page replacement algorithm.

```c
void * bubbleThreadFun(void *var);                    // sorting thread that called sorting, get, set functions
void * quickThreadFun(void * var);
void * mergeThreadFun(void * var);
void * indexThreadFun(void * var);

int pageReplace(int index, char * sortAlgo);          // page replace function

struct pageEntry{                                     // page entry
    int targetMemoryFrame;
    int prAbBit;
    int modifyBit;
    int refBit;
};
struct pageEntry * pageTable;                         // page table that holds page entries

int * physicalMemory;
int * virtualMemory;     // virtual memory is a integer array, too
int frameSize, memoryFrames, virtualFrames;
int fifoBubble = 0, fifoQucik = 0, fifoMerge = 0, fifoIndex = 0, fifoCheck = 0;
int pageReplacementBubble = 0, pageReplacementQuick = 0, pageReplacementMerge = 0, pageReplacementIndex = 0;

pthread_mutex_t lock;                                     // mutex for threads

int main(int argc, char const *argv[])
{
    int tempBubble = 999999;        // some temporaries
    int tempQuick = 999999;
    int tempMerge = 999999;
    int tempIndex = 999999;

    int smallestFrameSizeBubble;    // this variables gives smallest frame size after loop
    int smallestFrameSizeQuick;
    int smallestFrameSizeMerge;
    int smallestFrameSizeIndex;

    frameSize = 1 * 512;            // frame size increases with 512 byte
    memoryFrames = (64) * 1024 / frameSize;         // initial values
    virtualFrames = (1024) * 1024/ frameSize;
```

Figure 13

Optimal page size is the one that causes the smallest number of page replacements. Brute-force is used for the finding smallest one. Each time different page size(increase by 512 byte) is tried and find smallest. Loop continues while number of memory frame(64K / frame-size) is 4, because there are 4 sorting algorithms and each one has 1 page at least(in my design). Figure 14-15 are below.

```c
while( memoryFrames >= 4 ){         // loop until memory frame numbers lower than 4.(because we have 4 sorting algorithms)
    int totalMemoryIntegers = frameSize * memoryFrames;     // memory has frame * memoryframes integers
    physicalMemory = (int*) malloc(sizeof(int) * totalMemoryIntegers);
    for(int i = 0; i < totalMemoryIntegers; i++){
        physicalMemory[i] = 0;                              // initialize memory by zero
    }

    int totalVirtualIntegers = frameSize * virtualFrames;
    virtualMemory = (int*) malloc(sizeof(int) * totalVirtualIntegers);
    srand(1000);                                        // create same random numbers
    for(int i = 0; i < totalVirtualIntegers; i++){
        int randValue = rand();
        set(i, randValue, "fill");                      // set to virtual disk random numbers with "fill" parameter
    }

    printf("%s\n", "Please Wait......");
    pageTable = (struct pageEntry *) malloc(sizeof(struct pageEntry) * virtualFrames);      // page elements are changing according to virtual frames

    for(int i = 0; i < virtualFrames; i++){             // page table initialize.
        struct pageEntry ex;
        ex.targetMemoryFrame = 0;
        ex.prAbBit = 0;
        ex.modifyBit = 0;
        ex.refBit = 0;
        pageTable[i] = ex;
    }
    for(int i = 0; i < memoryFrames; i++){
        pageTable[i].prAbBit = 1;
        pageTable[i].targetMemoryFrame = i;

        for(int j = 0; j < frameSize; j++){                 // virtual memory to physical memory
            int value = virtualMemory[i * frameSize + j];
            physicalMemory[i*frameSize + j] = value;
        }
    }
```

Figure 14

```c

        pthread_t thread_idQuick;
        pthread_create(&thread_idQuick, NULL, quickThreadFun, (void *)&quarter);

        pthread_t thread_idMerge;
        pthread_create(&thread_idMerge, NULL, mergeThreadFun, (void *)&quarter);

        pthread_t thread_idIndex;
        pthread_create(&thread_idIndex, NULL, indexThreadFun, (void *)&quarter);


        pthread_join(thread_idBubble, NULL);
        pthread_join(thread_idQuick, NULL);
        pthread_join(thread_idMerge, NULL);
        pthread_join(thread_idIndex, NULL);

        pthread_mutex_destroy(&lock);                           // destroy the mutex

        printf("FOR FRAME SIZE : %d BYTE \n", frameSize);
        printf("pageReplacementBubble %d \n", pageReplacementBubble);
        printf("pageReplacementQuick %d \n", pageReplacementQuick);
        printf("pageReplacementMerge %d \n", pageReplacementMerge);
        printf("pageReplacementIndex %d \n", pageReplacementIndex);

        if( pageReplacementBubble <= tempBubble ){
            tempBubble = pageReplacementBubble;
            pageReplacementBubble = 0;
            smallestFrameSizeBubble = frameSize;
        }
        if( pageReplacementQuick <= tempQuick ){
            tempQuick = pageReplacementQuick;
            pageReplacementQuick = 0;
            smallestFrameSizeQuick = frameSize;
        }
        if( pageReplacementMerge <= tempMerge ){
            tempMerge = pageReplacementMerge;
            pageReplacementMerge = 0;
            smallestFrameSizeMerge = frameSize;
        }
        if( pageReplacementIndex <= tempIndex ){
            tempIndex = pageReplacementIndex;
            pageReplacementIndex = 0;
            smallestFrameSizeIndex = frameSize;
        }
        frameSize += 512;

        memoryFrames = (64) * 1024 / frameSize; // (* 1024)
        virtualFrames = (1024) * 1024/ frameSize; // (* 1024)
    }

    printf("Optimal Page Size for Bubble %d KB\n", smallestFrameSizeBubble / 1024);
```

Figure 15

System is same. 4 sorting thread works their area. Differently, the arrays where the elements brought from memory are kept sorted by quick sort. It doesn't block the system and doesn't change much. Some functions, such as bubble sort, run very slowly and with these parameters, the program takes too long to terminate. Because of this reason, with quick sort, only the temporary array is sorted.

```c
void * bubbleThreadFun (void * var)         // bubble thread function
{
    pthread_mutex_lock(&lock);              // lock the mutex
    int *myVar = (int *) var;
    int quarter = *myVar;                   // get parameter
    int arrBubble[quarter];
    int j = 0;
    for(int i = 0; i < quarter; i++){
        arrBubble[j] = get(i, "bubble");    // get first quarter of integers
        j++;
    }
    quickSort(arrBubble, quarter);      // sort them
    j = 0;
    for(int i = 0; i < quarter; i++){
        set(i, arrBubble[j], "bubble");     // set back elements
        j++;
    }
    pthread_mutex_unlock(&lock);            // unlock the mutex
    pthread_exit(NULL);
}
void * quickThreadFun (void * var)          // the same operations were done for quick thread function
{
    pthread_mutex_lock(&lock);
    int *myVar = (int *) var;
    int quarter = *myVar;
    int arrQuick[quarter];
    int j = 0;
    for(int i = quarter; i < quarter * 2; i++){
        arrQuick[j] = get(i, "quick");
        j++;
    }
    quickSort(arrQuick, quarter);
    j = 0;
    for(int i = quarter; i < quarter * 2; i++){
        set(i, arrQuick[j], "quick");
        j++;
    }
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}
void * mergeThreadFun (void * var)          // the same operations were done for merge thread function
{
    pthread_mutex_lock(&lock);
    int *myVar = (int *) var;
    int quarter = *myVar;
    int arrMerge[quarter];
    int j = 0;
    for(int i = quarter * 2; i < quarter * 3; i++){
        arrMerge[j] = get(i, "merge");
        j++;
    }
    quickSort(arrMerge, quarter);
```

Figure 16