CSE 312 HOMEWORK 2

VIRTUAL MEMORY MANAGEMENT SYSTEM

Çağrı Çaycı

1901042629

Page Table Entry

Valid Bit	Referenced Bit	Modified Bit	Old Integer	Frame Number Integer

Valid Bit: Valid bit is used as flag to indicate that frame number is valid or not.
Referenced Bit: Referenced bit is used to indicate the frame number is referenced. It used in both Second Chance algorithm and Least Recently Used algorithm.
Modified Bit: Modified bit is used to indicate the page is modified in the physical memory. In this way, when this page is changed, it is written to the disk first.
Old Integer: Old integer is used to control how recent the page is used. It is used in both Second Chance Algorithm and Least Recently Used algorithm.
Frame Number: Frame number indicates where the page is placed in physical memory.

```
typedef struct {
    bool valid = false;
    bool referenced = false;
    bool modified = false;
    int old = 0;
    unsigned int page_frame = -1;
}PageTableEntry;
```

Page Table

The page table consists of page table entries equal to the number of virtual pages. In this way, each virtual page has its own page table entries. It also provides to reach to correct physical memory page by using only indexes. For example, if the third page is needed, looking at the third entry of the page table is necessary to reach correct physical memory page.

```
class PageTable{
    private:
        PageTableEntry * pageTableEntries;
        PageReplacementMode mode;
        int counter = 1, pageFault = 0, pageReplacement = 0, read = 0, write = 0, readFromDisk = 0, writeToDisk = 0;
    public:
        PageTable(int size, PageReplacementMode mode){
            this->pageTableEntries = new PageTableEntry[size];
            this->mode = mode;
        }
        ~PageTable(){
            delete[] this->pageTableEntries;
        }
```

Physical Memory

Physical memory is kept as a 2D integer array in this project. It is filled by -1s to initialize it first.

Virtual Memory

Virtual memory is kept in a file in this project which is created and filled with random integers in the beginning of the program. Because the virtual memory size is bigger than the physical memory size, some operations (writing to the disk, reading from the disk) are applied when it is necessary.

PAGE FAULT CHECK

As it is mentioned above, to check the page is in physical memory or not, checking the validity of corresponding entry of page table is enough.

PAGE FAULT HANDLING

Encountering page fault does not mean that a page must be replaced in the physical memory. Because there can be available spaces in the physical memory to place the page. To verify this is the case, all the pages in the physical memory must be checked.

```
int properPage = -1;
for(int i = 0; i < physicalMemorySize && properPage == -1; i++){
    if(physicalMemory[i][0] == -1){ /* There is available spot in the physical memory. */
        properPage = i;
    }
}
```

Another possible case when handling page fault is that there is no available space in the physical memory to place the page. In this case, the solution is using page replacement algorithms. Page replacement algorithms are used to find the most proper page (least probable to need it again) to place the page.

```
if(properPage == -1){ /* There is no availabale spor in the physical memory. */
    pageReplacement++;
    if(mode == SC){ /* Second Chance. */
        properPage = secondChance(1);
    }else if(mode == LRU){ /* Least Recently Used. */
        properPage = leastRecentlyUsed();
    }else{ /* Working Set Clock. */
    }
}
```

SECOND CHANCE ALGORITHM

In second chance algorithm, all the pages are checked in the memory starting from first placed to the last placed. If the first placed page is not referenced (reference bit is 0), the proper page to replace is it. Otherwise, next pages (second produced, third produced, …, last produced) are checked sequentially. If all the pages are referenced, the first page is replaced.

```
int secondChance(int minOld){
    int pageReplaced, min = INT_MAX;
    for(int i = 0; i < virtualMemorySize; i++){
        if(this->pageTableEntries[i].valid && this->pageTableEntries[i].old < min && this->pageTableEntries[i].old >= minOld){
            min = this->pageTableEntries[i].old;
            pageReplaced = i;
        }
    }
    if(this->pageTableEntries[pageReplaced].referenced){
        this->pageTableEntries[pageReplaced].referenced = false;
        return secondChance(minOld + 1);
    }
    return pageTableEntries[pageReplaced].page_frame;
}
```

The approach of this solution in the project is like the algorithm. The old field is added to the page table entry fields to decide which page is placed first. All the pages are checked in the page table to find the oldest page in the physical memory. After the oldest page is found, the reference bit of the page is checked. If the oldest page has referenced, the function is called for next oldest page in the page table. At the end of the function, the corresponding page in the physical memory index is returned.

LEAST RECENTLY USED ALGORITHM

In the least recently used algorithm, clock is used to find oldest page which is loaded into physical memory. Contrary to the second chance algorithm, when a page is referenced, its clock is updated as it was created recently. So, the pages which are the placed least recent and are not used recently are the more probable to replaced.

```
int leastRecentlyUsed(){
    int pageReplaced, min = INT_MAX;
    for(int i = 0; i < virtualMemorySize; i++){
        if(this->pageTableEntries[i].valid && this->pageTableEntries[i].old < min){
            min = this->pageTableEntries[i].old;
            pageReplaced = i;
        }
    }
    return pageTableEntries[pageReplaced].page_frame;
}
```

The approach of this solution in the project is like the algorithm. All the pages are checked in the page table to find the oldest page in the physical memory. After the oldest page is found, the index of the page is returned. Addition to the function, when a page is referenced, the old field of the page is updated in the paging function.

WRITING TO THE DISK

If a page in the physical memory is updated during the program, the modified bit of the page must be set as 1. In this way, when the page replacement algorithm in the page table gives that page to be placed, the data of the page must be written to the disk to avoid data loss.

```
void writePageToDisk(char * filename, int pageNumberInDisk, int pageNumberInMemory){
    FILE * fp = fopen(filename, "r");
    char c;
    if(fp == NULL){
        printf("An error occured while opening the disk.\n");
        return;
    }
    FILE * temp = fopen("temp.dat", "w");
    if(temp == NULL){
        printf("An error occured while opening a temp file.\n");
    }
    for(int i = 0; i < pageNumberInDisk; i++){
        while((c = fgetc(fp)) != '\n'){
            fputc(c, temp);
        }
        fputc('\n', temp);
    }
    for(int i = 0; i < frameSize; i++){
        fprintf(temp, "%d ", physicalMemory[pageNumberInMemory][i]);
    }
    fprintf(temp, "\n");

    while(fgetc(fp) != '\n'){

    }

    while((c = fgetc(fp)) != EOF){
        fputc(c, temp);
    }
    fclose(fp);
    fclose(temp);

    remove(filename);

    rename("temp.dat", filename);

}
```

This approach is provided as the function above. Starting from the first page to the last page is written to the temp file except the page which will be updated. Instead of the old data of the page, new version of the page is got from the physical memory and written to the temp file. After all these are completed, the old disk file is removed, and the new temp file is renamed.

READING FROM THE DISK

As mentioned above, physical memory is not large enough to keep all virtual memory. So, when a page is needed in physical memory, it must be read from the disk. In this way, programs can be run even though there is not enough physical memory.

```
void readPageFromDisk(char * filename, int pageNumberInDisk, int pageNumberInMemory){
    FILE * fp = fopen(filename, "r");
    if(fp == NULL){
        printf("An error occured while opening the disk.\n");
        return;
    }
    for(int i = 0; i < pageNumberInDisk; i++){
        while(fgetc(fp) != '\n'){

        }
    }

    for(int i = 0; i < frameSize; i++){
        int number;
        fscanf(fp, "%d", &number);
        physicalMemory[pageNumberInMemory][i] = number;
    }
    fclose(fp);
}
```

This approach is provided as the function above. Corresponding page in the disk is read and placed into physical memory.

RESULTS