System Programming

Lab-01

담당 교수	홍정규 교수님
제 출 일	2023. 12. 17
학 번	2018440017
이 름	김민석

1. Part 1 (배경이론)

1.1 가상주소와 물리주소의 차이

컴퓨터의 주소체계는 System에서 Process가 보는 Address Space가 무엇인가? 라는 질문을 통해 크게 Physical Address와 Virtual Address로 나눌 수 있다.

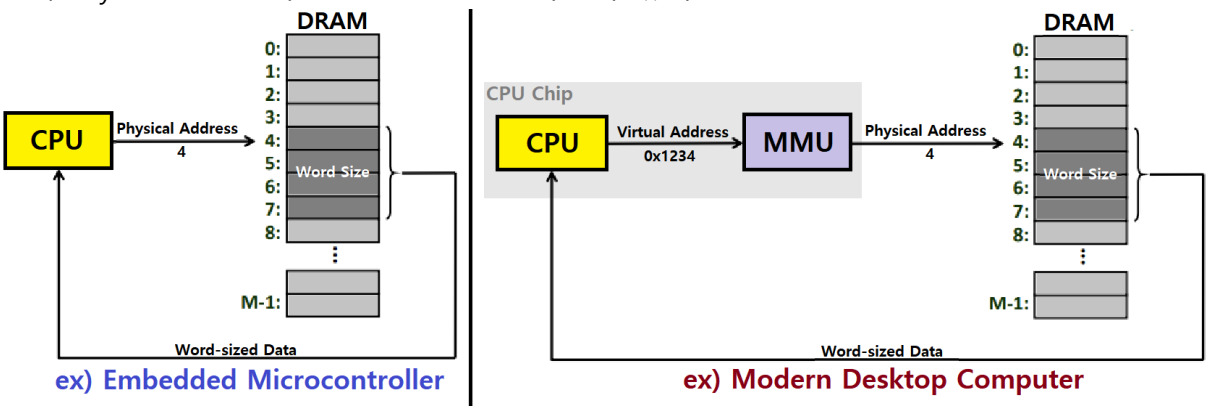

그림 1 – Physical Address & Virtual Address

- Physical Address : 실제 하드웨어에서 메모리의 주소를 나타내는 값이다. 즉, RAM에서 데이터가 저장되는 실제 위치를 나타낸다. 가상메모리를 지원하지 않는 Embedded Microcontroller같은 경우 CPU가 메인메모리에 접근할 때 Addressing하는 주소가 Physical Address이다.

- Virtual Address : 각 프로세스별로 실제 메모리의 위치와는 별개로 독립적으로 사용하는 주소이다. 모든 프로세스는 각자의 Virtual Address공간을 갖고있으며, 이는 MMU와 같은 하드웨어의 도움을 받아 Physical Address와 Mapping된다.

1.2 가상 주소 체계의 장점

- Memory Protection : 가상 주소 체계는 각 프로세스마다 가상 주소 공간을 갖고 있으며, 이러한 가상주소공간은 Isolated되어있다. 따라서 서로 다른 프로세스의 메모리영역을 침범할 수 없기 때문에 시스템의 안정성, 보안성이 높다고 할 수 있다.

- 편의성 : 프로세스가 각자의 가상주소공간을 갖고 이러한 가상주소가 실제주소와 mapping되기 때문에 개발자는 실제 물리적 메모리 위치를 고려하지 않고 프로그램을 개발할 수 있다.

- 효율성 : 가상 주소 체계를 사용시 물리메모리보다 큰 메모리공간을 사용할 수 있다. Process에게 물리메모리의 연속적인 큰 공간으로 할당할 필요가 없고 필요할 때 메모리에 Load하여 사용할 수 있기 때문에 효율적으로 사용할 수 있다.

1.3 Page Table & Page Table Entry & Page Fault

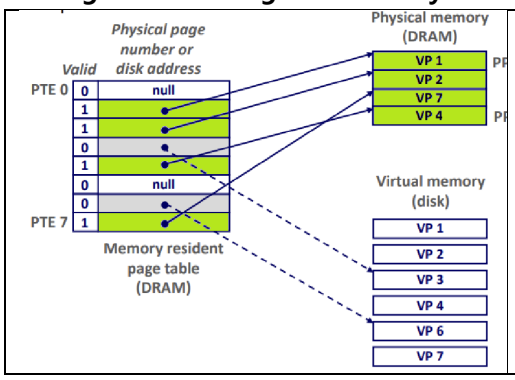

그림 2 – Page Table

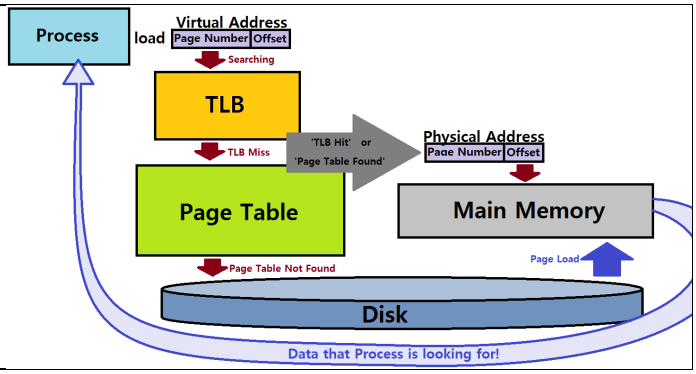

그림 3 – Architectue View

- Page Table : 앞서 설명한 Virtual Memory에서 Physical Memory로의 Mapping에 사용되는 자료구조로서 Process마다 DRAM에 존재한다.

- Page Table은 위의 그림2 과 같이 어떤 Virtual Page가 어떤 Physical Page에 Mapping되어있는지 확인할 때 사용된다. 이때 PTE는 Page Table에서 Index로서 활용된다고 할 수 있다.

- Page Fault : 그림 3을 참고하면, Virtual Address에서 Page Number를 통해 Page Table에 접근하였을 때 우리가 원하는 Target이 없는 상황을 Page Fault라고 한다. 즉 Main memory에 올라와있지 않고 Hard disk에 저장되어 있는 상태(valid bit가 0)를 의미한다.

1.4 Page Fault Hadler & Replacement Policy

- Page Fault Handler : Page Fault는 요청한 Page가 물리적 메모리에 Load되어 있지 않을 때 발생한다. 이러한 경우 Page Fault Handler가 동작하여 요청된 Page를 하드디스크와 같은 보조 기억장치로부터 메모리로 읽어오고 Page Table의 갱신이 필요하다. 또한, Page Fault Handler는 Kernel의 일부로 작동하며, Page Fault가 발생했을 때 즉시 실행된다.

- Replacement Policy : 위와 같이 Page Fault가 발생하였을 때, 이미 물리메모리가 가득찬 상황이라면 새로운 Page를 Load하기 위해 이미 존재하는 Page를 메모리에서 내쫓아야 하고, 이러한 Victim Page를 선택하는 정책을 Replacement Policy라고 한다. 대표적인 예로 Optimal, LRU, MRU 등이 존재한다.

Disk가 DRAM보다 10000배 정도 느리기 때문에 Replacement를 진행하는데 상당한 Overhead가 발생한다. 따라서 매우 정교한 Replacement Policy가 필요하다.

1.5 Signal & Signal Handler

	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

Signal : OS Kernel이 제공하는 Alert Mechanism으로 시스템에서 프로세스에게 어떠한 특정 이벤트가 발생하였음을 알리는 아주 작은 Message이며 1~30의 정수로 이루어진 ID로 구분된다. 또한 Signal은 다른 부수적인 정보를 포함하지 않으며 Signal ID와 도착정보 만을 포함한다.

Kernel이 Signal을 보내는 이유는 다음과 같다. 첫째, Division by Zero와 같은 상황을 감지(SIGFPE), 둘째, Child Process의 Termination을 감지(SIGCHLD), 셋째, 특정 Process가 다른 Process에 Kill System Call을 통해 Explicit Request를 보낸 경우. 즉, Signal은 커널이 직접 프로세스에게 보낼수도 있고, 다른 프로세스가 커널에게 시그널을 요청 할 수도 있다.

또한 Signal은 Pending 되지 않는다. 즉, Context Switching이 일어나기 직전에 Signal 도착여부를 확인할 뿐, 같은 Signal이 여러번 도착하였더라도 Block되어 한번만 인식할 수 있다.

Process가 Signal을 받았을 때 다음과 같은 동작을 수행할 수 있다.

- Ignore : 받은 Signal을 무시한다. (단, SIGKILL, SIGSTOP등은 ignore할 수 없다.)
- Terminate : 프로세스를 종료한다.
- Catch & Call sig_handler : 정의된 Default Action을 수행하지 않고, User Defined Function을 수행한다.

Signal Handler Installation

```
handler_t *signal(int signum, handler_t *handler)
```

- Signum : 처리하고자하는 signal번호.
- Handler : SIG_IGN(ignore), SIG_DFL(default), address of user defined signal handler.

위와 같은 함수를 활용하여 Signal에 대한 Handler를 Installation할 수 있다.

2. Part2 (소스코드 작성)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <time.h>
#include <math.h>

#define ADDR_SIZE 27 //주어진 시스템은 27-bit 주소를 사용함
#define NUM_FRAMES 64 //주어진 시스템은 64 개의 물리 프레임이 있음
#define MAX_LINE_SIZE 20000

int page_size = -1; //한 페이지의 크기(Byte)
int num_pages = -1; //각 프로세스의 페이지 개수

// 페이지 테이블
int* pageTable = NULL;

// 물리적 메모리 페이지(프레임)
int physicalMemory[NUM_FRAMES];

// 페이지 폴트 및 페이지 교체 횟수
unsigned int pageFaults = 0;


// 프로세스의 종료 구분 플래그
int processDone = 0;
// 프로세스를 종료하지 않기 위한 플래그
int contFlag = 0;

///////////////// FOR LRU //////////////////////
typedef struct node { //LRU 캐시를 위한 노드
    int page; // page entry 를 저장
    struct node *prev;
    struct node *next;
} node;

node *cache = NULL; // cache 를 저장할 연결리스트 선언

void init_cache() {cache = NULL;} // cache 를 초기화

node *new_node(int data) { // 새로운 노드를 생성하는 함수
    node *n = (node *)malloc(sizeof(node)); // 노드를 동적할당
    n->page = data;
    n->next = NULL;
    n->prev = NULL;
    return (n);
}
```

```
void delete_cache(node *del) { // 노드를 삭제하는 함수
    if (cache == NULL || del == NULL)
        return;
    if (cache == del) // 삭제할 함수가 head
        cache = del->next;
    if (del->next != NULL) // next 가 null 이 아닐 때
        del->next->prev = del->prev;
    if (del->prev != NULL) // prev 가 null 이 아닐 때
        del->prev->next = del->next;
    free(del); // 삭제할 노드를 free
}

void insert_cache(int data) { // 최신의 data 를 head 에 넣는 함수
    node *ptr = cache; // head 를 pointing 하는 포인터
    node *n = new_node(data); // 삽입할 새로운 노드 생성
    while (ptr != NULL) {
        if (ptr->page == data) { // 일치하는 데이터(page entry)가 존재한다면
            delete_cache(ptr); // 이미 존재하는 노드를 최신으로 갱신하기 위한
삭제
            break;
        }
        ptr = ptr->next;
    }
    n->prev = NULL;
    n->next = cache; // 새로운 노드를 연결리스트의 제일 앞에 삽입

    if (cache != NULL)
        cache->prev = n;
    cache = n; // 헤드노드를 새로운 노드로 변경
}

void printCache() {
    node *current = cache;
    printf("LRU Cache: ");
    while (current != NULL) {
        printf("%d ", current->page);
        current = current->next;
    }
    printf("\n");
}

////////////////////////////////////////////////////

//TODO-2. 입력된 페이지 오프셋을 통해 페이지 크기 및 페이지 개수를 계산
void calculatePageInfo(int page_bits) {
    page_size = (int)pow(2,page_bits); // offset 개수의 bit 로 나타낼 수 있는
경우의 수 만큼 page_size 를 설정
    num_pages = (int)pow(2, ADDR_SIZE - page_bits); // page 의 개수 = (Logical
address 의 전체 경우의 수) / (page size)
```

```
}

// 가상 주소에서 페이지 번호 추출
int getPageNumber(int virtualAddress) {
    return virtualAddress / page_size;
}


//TODO-4. 이 함수는 교체 정책(policy)을 선택 후,
//알고리즘에 따른 교체될 페이지(victimPage)를 지정.
//교체될 페이지가 존재했던 물리 프레임 번호(frameNumber)를 반환(return).
int doPageReplacement(char policy) {
    int victimPage = -1; //교체될(evictee) 페이지 번호
    int frameNumber = -1; //페이지 교체를 통해 사용가능한 프레임 번호(return
value)

    static int defaultVictim = 0;// 샘플 교체정책에 사용되는 변수
    switch (policy) {
    case 'd': //샘플: 기본(default) 교체 정책
    case 'D': //순차교체: 교체될 페이지 엔트리 번호를 순차적으로 증가시킴
        while (1) { //유효한(물리프레임에 저장된) 페이지를 순차적으로 찾음
            if (pageTable[defaultVictim] != -1) {
                break;
            }
            defaultVictim = (defaultVictim + 1) % num_pages;
        }
        victimPage = defaultVictim;
        break;
    case 'r':
    case 'R': //TODO-4-1: 교체 페이지를 임의(random)로 선정
        while (1) {
            victimPage = rand() % num_pages; // random 으로 victimPage 를
설정함
            if (pageTable[victimPage] != -1) break; // 유효한 페이지면 탈출
        }
        break;
    case 'a':
    case 'A': // LRU 방식의 Policy 구현
        for(node *ptr = cache; ptr !=NULL; ptr = ptr->next) { // cache 에서
제일 마지막 노드(가장 오래전에 사용한 노드)를 찾는 반복문
            if (ptr->next == NULL){
                victimPage = ptr->page; // 찾았다면 해당 pageentry 를
victimPage 로 선정
                delete_cache(ptr); // 해당 page 를 LRU cache 에서 삭제
            }
        }
        break;
```

```
    default:
        printf("ERROR: 정의되지 않은 페이지 교체 정책\n");
        exit(1);
        break;
    }

    frameNumber = pageTable[victimPage]; //교체된 페이지를 통해 사용 가능해진 물리 프레임 번호
    pageTable[victimPage] = -1;  //교체된 페이지는 더 이상 물리 메모리에 있지 않음을 기록

    return frameNumber;
}

// 페이지 폴트 처리
void handlePageFault(int pageNumber, char policy) {
    int frameNumber = -1; //페이지 폴트 시 사용할 물리 페이지 번호(index)

    // 사용하지 않는 프레임을 순차적으로 할당함
    // (p.s. 실제 시스템은 이런식으로 순차할당하지 않습니다.)
    static int nextFrameNumber = 0;
    static int frameFull = -1;      //모든 물리프레임이 사용된 경우 1, 아닌 경우 -1

    //물리 프레임에 여유가 있는 경우
    if (frameFull == -1) {
        frameNumber = nextFrameNumber++;
        //모든 물리 프레임이 사용된 경우, 이를 마크함
        if(nextFrameNumber == NUM_FRAMES)
            frameFull = 1;
    }
    //모든 물리 프레임이 사용중. 기존 페이지를 교체해야 함
    else {
        //TODO-4. 페이지 교체 알고리즘을 통해 교체될 페이지가 저장된 물리 프레임 번호를 구함
        frameNumber = doPageReplacement(policy);
    }

    // 페이지 테이블 업데이트
    pageTable[pageNumber] = frameNumber;

    printf("페이지 폴트 발생: 페이지 %d 를 프레임 %d 로 로드\n", pageNumber, frameNumber);
    pageFaults++;
}


//TODO-1. 'Ctrl+C' 키보드 인터럽트(SIGINT) 발생 시 처리 루틴
void myHandler() {
```

```
    printf("\n(Interrupt) 현재까지 발생한 페이지폴트의 수: %d\n", pageFaults);

    //모든 작업이 완료되었을 시, 시그널 발생 시 동작:
    if (processDone == 1) { // 전체 process 가 종료되었다면
        contFlag = 1; //contFlag 를 1 로 변화시켜 main 함수에서의 while 문의
동작을 멈춤
        printf("2018440017 / 김민석(Minseok-Kim)\n\n"); // 학번/이름을 출력
    }
    signal(SIGINT, myHandler);
}

int main(int argc, char* argv[]) {
    //TODO-1. SIGINT 시그널 발생시 핸들러 myHandler 구현 및 등록(install)
    signal(SIGINT, myHandler);

    srand(time(NULL));
    if (argc <= 2) {
        printf("please input the parameter! ex)./test 13 d\n");
        printf("1st parameter: page offset in bits\n2nd parameter:
replacement policy\n");
        return -1;
    }
    init_cache(); // LRU cache 를 초기화함

    int page_bits = atoi(argv[1]);  //입력받은 페이지 오프셋(offset) 크기
    char policy = argv[2][0];       //입력받은 페이지 교체 정책

    //TODO-2. 입력정보를 바탕으로 페이지 크기 및 페이지 개수 계산
    calculatePageInfo(page_bits);

    printf("입력된 페이지 별 크기: %dBytes\n 프로세스의 페이지 개수: %d 개\n 페이지
교체 알고리즘: %c\n",
        page_size, num_pages, policy);


    pageTable = (int*)malloc(num_pages * sizeof(int)); // page 의 개수만큼
pagetable 을 생성

    for (int i = 0; i < num_pages; i++)
        pageTable[i] = -1;

    for (int i = 0; i < NUM_FRAMES; i++)
        physicalMemory[i] = 0;


    // 파일 읽기
    const char* filename = "input.txt";
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        perror("파일 열기 오류");
```

```
        return EXIT_FAILURE;
    }
    // 파일 내 데이터: 가상 메모리 주소
    // 모든 메모리 주소에 대해서
    int lineNumber = 0;
    while (!feof(file)) {
        char line[MAX_LINE_SIZE];
        fgets(line, MAX_LINE_SIZE, file);
        int address;
        sscanf(line, "%d", &address);

        // 가상 주소에서 페이지 번호(pageNumber)를 얻음
        int pageNumber = getPageNumber(address);

        // pageTable 함수는 페이지 폴트 시 -1 값을 반환함
        int frameNumber = pageTable[pageNumber];
        if (frameNumber == -1) { //page fault
            handlePageFault(pageNumber, policy); //페이지 폴트 핸들러
            frameNumber = pageTable[pageNumber];
        }
        insert_cache(pageNumber); // cache 정보를 업데이트. (이미 존재하는
page 라면 최신으로 갱신, 새로운 page 라면 삽입->victim page 는 pagefault
handler 에서 이미 삭제)
        //해당 물리 프레임을 접근하고 접근 횟수를 셈
        physicalMemory[frameNumber]++;

        lineNumber++;
        // usleep(1000); //매 페이지 접근 처리 후 0.001 초간 멈춤
        //이 delay 는 프로세스 수행 중, signal 발생 처리과정을 확인하기 위함이며,
        //구현을 수행하는 도중에는 주석처리하여, 빠르게 결과확인을 하기 바랍니다.
    }

    fclose(file);
    free(pageTable);
    // 작업 수행 완료. Alarm 시그널을 기다림.
    processDone = 1;
    printf("프로세스가 완료되었습니다. 종료 신호를 기다리는 중...\n");
    while (contFlag == 0){};

    // 결과 출력
    printf("\n---물리 프레임 별 접근 횟수----\n");
    for (int i = 0; i < NUM_FRAMES; i++) {
        printf("[%03d]frame: %d\n", i, physicalMemory[i]);
    }
    printf("----------\n 페이지 폴트 횟수: %d\n", pageFaults);

    return 0;
}
```

3. Part3 (분석 및 이해)

3.1 코드 설명

```
void calculatePageInfo(int page_bits);
```

입력한 offset bit에 따라 page_size와 num_pages의 값을 결정하는 함수이다.
Offsetbit의 값이 m이라면 offsetbit로 표현가능한 경우의 수는 2^m이다. 따라서 각각의 page에 저장되는 주소의 개수가 2^m개이고. Page의 크기가 2^mbytes임을 알 수 있다.
이번 Lab에 주어진 시스템 체계는 27bit 시스템을 활용한다. 따라서 전체 주소공간의 개수는 2^{27}, pagesize를 위에서 구했기 때문에 나누어 주면 page의 개수는 2^{27-m}임을 알 수 있다.

```
int getPageNumber(int virtualAddress);
```

virtualAdress를 입력받아 해당 address가 속하는 PageNumber 즉 PTE값을 얻는 함수이다.
virtualAddress를 Page 개수만큼 나눈 몫을 반환한다.

```
void myHandler();
```

문제의 조건에 따라 SIGINT 시그널이 입력되었을 때 처리하기위한 sig_handler 함수이다.
기본적으로 현재 발생한 PageFault 횟수를 출력하도록 설정되어 있으며,
Input.txt파일을 모두 읽었을 때 processDone 변수가 1로 설정되면 학번을 출력하고 contFlag를 1로 설정하여 main함수의 while Loop가 종료되도록 설정하였다.

```
int doPageReplacement(char policy);
```

정책에 따라 주어진 replacement policy를 수행하는 함수이다.
Skeleton code에서 default로 구현된 순차교체 방식을 살펴보면 Pagetable을 순차적으로 순회하면서 -1값이 아닌 테이블 즉 memory에 로드가 되어있는 첫번째 page를 찾은 후 그 페이지를 victimpage로 지정한다.

다음으로 Random방식은 while Loop를 돌면서 rand()함수를 활용하여 난수를 생성한뒤 page개수로 나눈 나머지에 해당하는 page가 메모리에 로드되어있다면 (Page Table의 값이 -1이 아니라면) 해당 page를 victimpage로 지정한다.

마지막으로 LRU(Last Recently Used)방식은 가장 오래전에 로드된 Page를 victimPage로 지정하는 방식이다. Data에 Load될 때 LRU Cache라는 연결리스트에 해당 페이지 번호가 순서대로 저장되도록 하였고, Fault가 발생하면 가장 오래된 노드를 삭제하고 해당 페이지를 victimpage로 지정한다.

이렇게 각각의 알고리즘에 의해 victimpage가 지정되면 pagetable[victim] = -1 즉, 메모리에서 쫓아내고, framenumber = pagetable[victim] 기존에 저장되어있던 주소를 반환한다.

`void handlePageFault(int pageNumber, char policy);`
Pagefault가 발생하였을 때 처리하고 pagefault가 발생했음을 출력하는 함수이다. 우선, 물리 프레임에 공간이 있는경우 page를 frame에 순차적으로 할당한다. 모든 물리 프레임이 사용된 경우 `int doPageReplacement(char policy);` 함수를 호출하여 입력된 policy에 따른 victimpage를 찾아 framenumber를 리턴받고, 해당 page를 물리 프레임에 로드한다.

`int main(int argc, char* argv[]);`
1. `signal(SIGINT, myHandler);` 함수를 활용하여 SIGINT 에 대해 handler 함수를 지정한다. 2. argc (입력된 인자의 개수) <= 2 즉 offset bit, policy 중 입력이 안된것이 있다면 오류를 출력한다. 3. 입력된 offsetbit 와 policy 를 저장한다. 이때 argv 는 문자열이기 때문에 atoi 함수를 활용하여 적절히 int 형으로 형변환을 시행한다. 4. 위에서 소개한 calculate~~ 함수를 활용하여 offsetbit 에 따른 page_size, num_pages 값을 갱신한다. 5. int 형 Pagetable 배열을 num_pages(page 의 개수)만큼 동적할당한다. 6. 요청할 주소목록이 적혀있는 input.txt 파일을 open 한다. 7. while 문을 돌면서 파일의 끝을 만날 때 까지 한줄씩 virtualAddress 를 input.txt 파일에서 읽는다. 8. 입력받은 address 와 getPageNumber 를 활용해서 PTE 를 얻는다. 9. 해당 page 가 memory 에 로드되어있는지 확인하고 pagefault 가 발생한다면 handlepagefault 함수를 활용하여 victimpage 를 지정한다. 10. 모두 읽었다면 file 을 close 하고 pagetable 을 free 한다. 11. SIGINT 가 발생할 때 까지 대기하다가 SIGINT 가 입력되면 프레임별 접근 횟수, pageFault 발생 횟수를 출력하고 프로그램을 종료한다.

3.2 Page_size 및 Replacement Policy에 따른 변화

Page_size에 따른 변화, Replacement Policy(d)			
Offset bit	Page_size[Bytes]	Num_Pages	Fault 횟수
13	8192	16384	565
14	16384	8192	526
15	32768	4096	472
18	262144	512	132
20	1048576	128	46

- Offset bit가 증가함에 따라 Page_size가 2배씩 증가함을 알 수 있다. 원인은 앞서 설명한 바와 같이 offsetbit로 표현가능한 경우의 수가 2^m이기 때문이다.
- Page_size가 2배씩 증가함에 따라 Page의 개수는 2배씩 감소함을 알 수 있다. 원인은 앞서 설명한 바와 같이 page의 개수는 2^{27-m}이기 때문이다.
- 문제 조건이 Frame의 개수가 고정적이기 때문에 page수가 감소하고 Page의 크기가 커질수록 더 많은 주소를 담고 있기 때문에 Fault횟수가 감소함을 알 수 있다.

	Larger Page_size	Smaller Page_size
Table Size	작아짐	커짐
내부 단편화	증가	감소
입출력 효율	나쁨	좋음

- 위의 결과를 표로 정리하면 다음과 같다. Page_size가 커짐에 따라 Table_size는 감소하며, 내부단편화는 증가, 입출력 효율은 Page_Fault가 발생할 시 Disk에서 더욱 큰 Data를 Memory에 로드해야 하기 때문에 더 많은 시간이 걸리므로 나빠진다

Replacement Policy에 따른 Pagefult횟수의 변화			
	Default	Random	LRU
13	565	574	539
14	526	537	498
15	472	463	441
18	132	128	109
20	46	46	46

- 전체적으로 모든 알고리즘이 Page_size가 증가함에 따라 Fault 발생 횟수가 감소하고 있음을 확인할 수 있다.
- 차이점으로는 출력 결과에서 victimPage를 지정하는 부분에서 차이가 발생했다. Default 방식은 단순히 순차탐색을 통해 처음 발견되는 page를 victimpage로 지정하고 있음을 확인하였고, Random의 경우 랜덤하게 처음 발견된 page를 victimpage로 지정하였다. LRU의 경우 가장 오래전에 사용된 메모리를 victimpage로 지정하는것을 확인할 수 있었다.
- 또한 Random방식의 특성상 Fault횟수가 일정한 다른 Policy들과 달리 계속 변화하는 특성을 보였다.

마지막으로 LRU Cache가 제대로 동작하는지 확인하기 위해 입력되는 Page번호에 따른 **LRU Cache 변화**를 확인해 보았다.

1. 꽉 찬 Frame으로 인한 PageFault가 발생

페이지 폴트 발생: 페이지 **3952를 프레임 3**로 로드

LRU Cache: **3679** 3375 12653 7429 3599 3407 2540 3607 12697 2568 3257 4171 4147 4179 4012 5882 4026 3259 3994 3621 3533 12683 3377 2909 11353 12472 3258 7447 3690 3148 136 3172 3268 3156 3173 3583 2954 2637 4029 3248 4077 3381 4111 2634 4135 4068 3523 4083 4015 3942 3812 14653 7469 3605 3194 2120 3580 15100 12487 3386 3850 0 **16211** **3952**

페이지 폴트 발생: 페이지 **4087를 프레임 3**로 로드

LRU Cache: **4087** **3679** 3375 12653 7429 3599 3407 2540 3607 12697 2568 3257 4171 4147 4179 4012 5882 4026 3259 3994 3621 3533 12683 3377 2909 11353 12472 3258 7447 3690 3148 136 3172 3268 3156 3173 3583 2954 2637 4029 3248 4077 3381 4111 2634 4135 4068 3523 4083 4015 3942 3812 14653 7469 3605 3194 2120 3580 15100 12487 3386 3850 0 **16211**

위의 결과를 살펴볼 때 가장 오래된 페이지인 3952를 victim Page로 지정하고 3952가 위치해있던 Frame인 3번 Frame을 새로 입력된 4087에 할당 하는 것을 확인할 수 있다.

2. 이미 Memory에 Load되어 있는 Page가 다시 입력될 때

Page input	**LRU Cache**
3583	LRU Cache: **3583**
12697	LRU Cache: 12697 **3583**
2634	LRU Cache: 2634 12697 **3583**
3952	LRU Cache: 3952 2634 12697 **3583**
12697	LRU Cache: 12697 3952 2634 **3583**
2634	LRU Cache: 2634 12697 3952 **3583**
12697	LRU Cache: 12697 2634 3952 **3583**
3583	LRU Cache: **3583** 12697 2634 3952

위의 결과와 같이 이미 Load되어 있는 Page가 들어오면 해당 Page에 대한 정보를 최신의 노드로 갱신하고 있음을 알 수 있다(제일 왼쪽 즉, 헤드노드가 최신의 정보, Tail쪽으로 갈수록 오래된 정보).

3.3 과제를 수행하며 느낀점

이번 학기 시스템 프로그래밍 강좌를 수강하면서 개인적으로 가장 어려웠던 파트인 Virtual Memory 관련 과제가 나와서 솔직히 처음에는 당황했지만 과제를 수행하는 과정에서 온라인강의, 과거 운영체제 수업에서 봤던 강의자료 등등을 살펴보며 개념을 잡을 수 있는 기회가 있었습니다.

또한 지난학기 운영체제 수업에서는 이론 위주의 강의라 두루뭉술한 느낌이였다면 이번 과제를 통해 기본적인 Page_size 계산, Page의 개수 계산 등을 실제로 해보고 도출된 값들을 통해 Page_Table을 직접 구성하고, LRU를 통한 Replacement Policy도 구현해보면서 Virtual Memory와 좀더 친숙해질 수 있었습니다.

마지막으로 LRU 구현에 연결리스트 자료구조를 활용하였는데 처음에 정보를 업데이트 하는 과정이 쉽지 않았지만 과거 수강했던 자료구조 강의자료를 다시 살펴보며 해결할 수 있었습니다.