OS Project 2

Scheduling Simulator

With paging

모바일시스템공학과

32214362 조윤서

32204288 조민욱

Contents

Introduction

Analyzing and planning

Virtual address 와 Physical address 를 왜 매핑해야 할까? Virtual address 라는 개념이 왜 필요할까? 컴퓨터에서 실행되는 모든 프로그램은 데이터를 읽고 쓰기 위해 메모리에 접근해야 한다. 하지만 물리 메모리(Physical Memory)는 제한된 자원이고, 모든 프로그램이 직접 물리 메모리를 사용하면 충돌이 발생하거나 효율성이 크게 떨어질 수 있다. 이를 해결하기 위해 가상 주소(Virtual address)라는 개념이 도입되었다. 예를 들어, 다음과 같은 경우 분명히 메모리 상에서는 해당 프로세스를 감당할 수 있는 용량이 충분하다. 다만 프로세스의 메모리가 직접 매핑 되는 방식이기에 직접 들어가서 사용될 수 없다. 이런 단점을 극복하기 위해 Page Table 이 생기고, 프로세스의 메모리 공간 가상화를 매핑 관계를 통해서 구현하게 되었다. 결론적으로, Virtual address 는 물리 메모리 자원을 효율적으로 사용하고, 프로세스 간의 충돌을 방지하며, 대형 프로그램의 실행을 가능하게 하기 위해 반드시 필요한 개념이다.

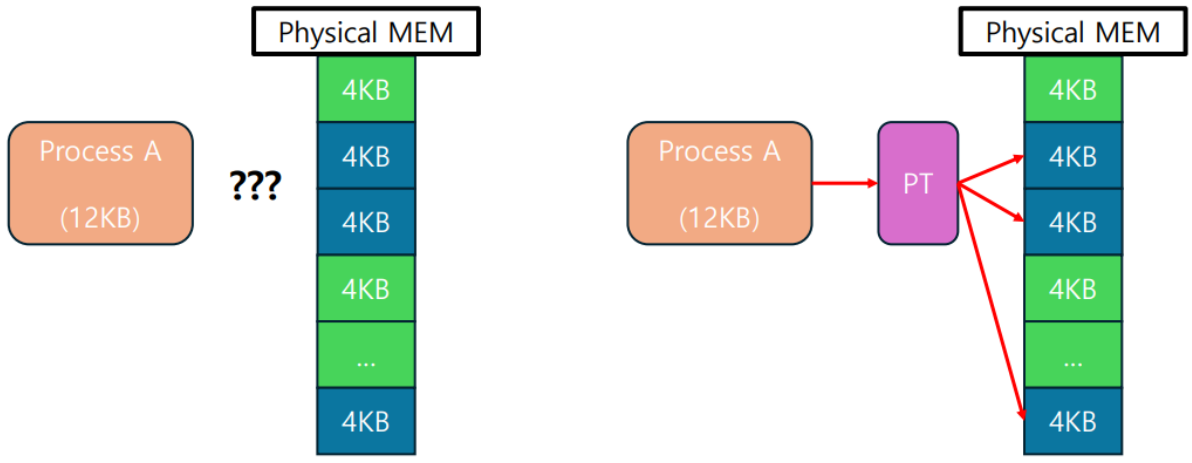

Figure 1) Process memory management

물리적 메모리 공간을 어떤 일정한 단위로 나누고, 각 단위를 프레임(페이지)라고 부른다. 즉 4GB(2^{32})의 물리 메모리를 가정했을 때, 4KB(2^{12})의 프레임은 총 2^{20} 개 존재한다.

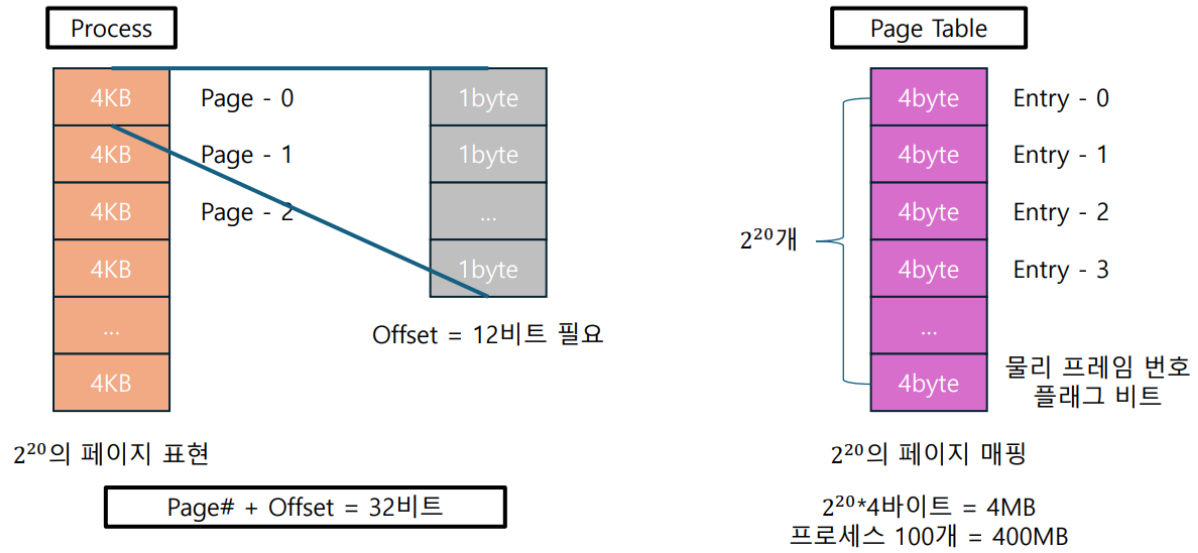

Figure 2) Process and Page Table

1-level paging 과 2-level paging 은 프레임과 페이지를 매핑해주는 관계에서 페이지 테이블의 용량을 줄이기 위해 사용되는 경우다. 두 케이스 모두 Physical address 이전에 2^{20} 개의 엔트리가 존재하고 각 값은 2^{20} 개의 서로 다른 프레임을 구별할 수 있어야 한다.

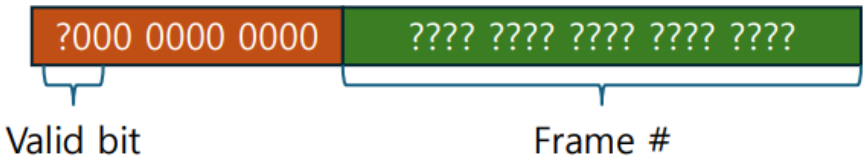

Figure 3) Entry value

1-level 의 경우 각각의 페이지가 프레임에 1 대 1 매핑 관계를 가지며, 직접적으로 페이지 테이블의 인덱스를 참조하게 된다. 즉 페이지 테이블에는 2^{20} 개의 엔트리가 존재하고, 각 엔트리는 프레임의 특정 위치 값을 가지게 된다. 이후 Offset 값을 통해 실제 바이트 물리 메모리 값을 읽고 프로세스가 사용한다.

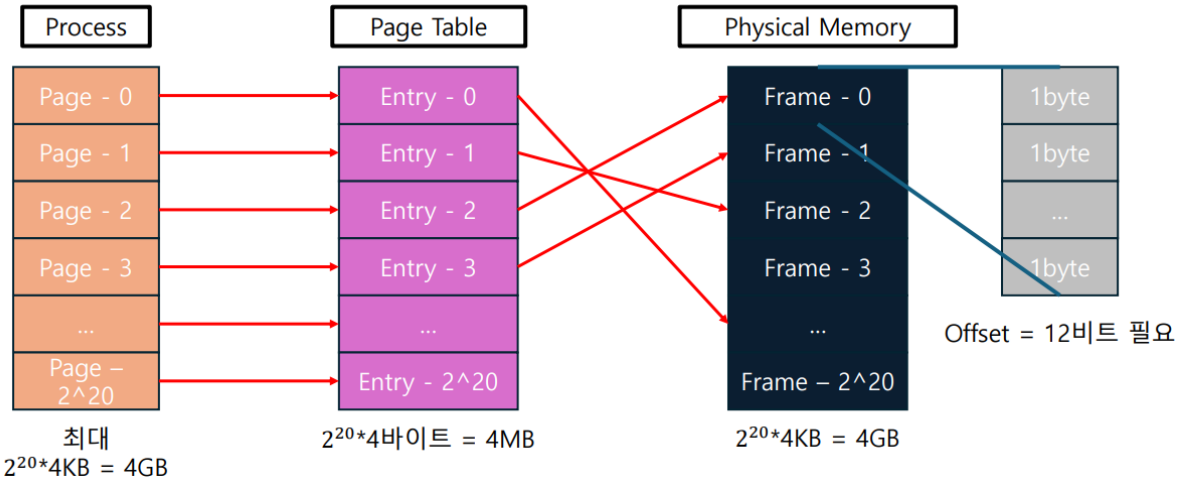

Figure 4) 1-level paging

Main explanation

Important concept

기존의 scheduling 코드를 바탕으로, 메시지 큐 2 개와 메시지 구조체 1 개를 추가로 사용하여 **'자식 프로세스의 메모리 접근 요청 / 부모 프로세스의 메모리 값 전달'**을 구현하였다. CPU 를 할당받아 실행되는 자식 프로세스는 무작위 가상 주소를 생성하고 해당 주소에 대한 접근 요청을 부모에게 보낸다.

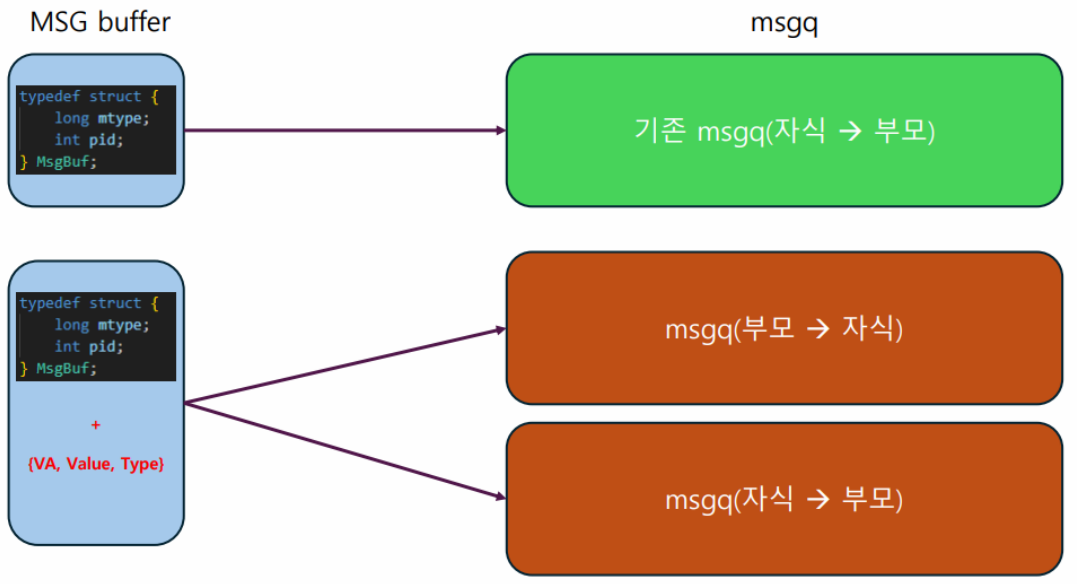

Figure 5) Whole MSG structure

- 부모 프로세스
  1. Wait Queue 처리 후 자식 프로세스에게 종료 신호(KILL)를 보냄
  2. 자식 프로세스의 MSG-1 받아 처리 → CPU 실행 종료
  3. **(추가) 자식 프로세스의 MSG-2 받아 처리 → 10 개의 가상 주소 접근**
  4. Run Queue 처리 후 자식 프로세스에게 실행 신호(KILL)를 보냄
- 자식 프로세스
  1. 부모에게 보내는 첫번째 MSG → CPU 사용 완료 MSG 전송
  2. **(추가) 부모에게 보내는 두번째 MSG → RAM 접근 요청 MSG 전송**

```
typedef struct {
    long mtype;
    int pid;
    int virtual_address[NUM_MEMORY_ACCESS]; // 0x123455
    unsigned char value[NUM_MEMORY_ACCESS]; // 'a'
    unsigned char type[NUM_MEMORY_ACCESS]; // 'w', 'r'
} MsgBuf2;
```

Figure 6) Second MSG structure

TTBR 을 페이지 디렉토리를 가리키는 초기 포인터로 선언하였다. 페이지 테이블 자체를 저장하면 페이지 테이블 크기가 커질 때 감당하지 못하기 때문에, 페이지 테이블의 시작 주소만 저장하여 필요한 경우 해당 주소를 참조하도록 설계하였다. 이는 메모리 사용을 효율적으로 관리하고, 페이지 테이블 크기가 큰 경우에도 효과적으로 접근할 수 있도록 한다. 또한 하나의 프레임을 CHAR 형 4096 개 구조체로 선언하였다. 즉 전체 메모리는 해당 프레임 구조체의 2^{20}개 배열로 표현된다.

```c
// Virtual Memory(Paging) 관련 구조체 선언
typedef struct {
    int pid;
    int** pg_dir; //TTBR
} pcb;

typedef struct {
    char block[BLOCK_TO_CHAR];
} frame;
```

Figure 7) Page Table-Related Structures

자식 프로세스는 무작위로 10 개의 가상 메모리 접근을 생성하고, 부모에게 메시지를 보낸다.

```c
for (int i = 0; i < NUM_MEMORY_ACCESS; i++) {
    int va = rand(); // 32비트 가상 주소 생성
    char type = (rand() % 2) == 0 ? 'r' : 'w'; // 랜덤으로 읽기 또는 쓰기 결정

    // 작업 설정
    msg_to_parent.virtual_address[i] = va;
    msg_to_parent.type[i] = type;

    if (type == 'w') {
        msg_to_parent.value[i] = (char)((rand() % 26) + 'a'); // 'w'인 경우만 value를 할당
    } else {
        msg_to_parent.value[i] = 0; // 'r'인 경우 value를 0으로 초기화 (옵션)
    }

    fprintf(fp, "Child %d: VA[%d] = %08x, Value = %c, Type = %c\n",
       pid, i, va, type == 'w' ? msg_to_parent.value[i] : 0, type);
    fflush(fp);
}
```

Figure 8) Child Process – 10 VA Entries

이후 부모는 Page Fault를 확인하는 함수, Page Fault 발생을 처리하는 함수, VA를 PA 로 번역하는 함수 3 가지를 사용한다.

1. is_page_fault → VA 를 바탕으로 각 인덱스를 구한 다음, 프레임 직전 최종 테이블의 Valid bit 를 확인한다. 즉 가리키는 프레임의 존재 여부를 확인한다.
2. handle_page_fault → VA 를 바탕으로 각 인덱스를 구한 다음, 해당 페이지 테이블이 최종적으로 가리키는 새로운 프레임을 할당한다.
3. translate_VA_to_PA → VA 를 PA 로 변환한다.

이후 부모가 PA 에 실제 존재하는 데이터를 뽑고, 실제 물리 메모리 주소값을 메시지를 통해서 보낸다.

```
// 10개의 VA처리
for (int i = 0; i < NUM_MEMORY_ACCESS; i++) {
    int va = msg_from_child.virtual_address[i];
    char type = msg_from_child.type[i];

    // printf("[Parent] Processing VA: %08x, Type: %c\n", va, type);

    if (is_page_fault(pcbs[index].pg_dir, va)) { // 페이지 폴트 처리
        handle_page_fault(pcbs[index].pg_dir, va, msg_from_child.pid);
    }

    int pa = translate_VA_to_PA(pcbs[index].pg_dir, va);
    // printf("Translated VA %08x to PA %08x\n", va, pa);
    fprintf(fp, "Process %d: VA %08x -> PA %08x\n", msg_from_child.pid, va, pa);

    if (type == 'w') {
        char value = msg_from_child.value[i]; // 'write' 작업의 경우 자식으로부터 받은 value를 사용
        RAM[(pa >> 12) & FRAME_NUMBER_MASK].block[pa & PAGE_OFFSET_MASK] = (char)value;
        fprintf(fp, "[Parent] Wrote Value: %c to PA: %08x\n", value, pa);
    } else if (type == 'r') {
        char read_value = RAM[(pa >> 12) & FRAME_NUMBER_MASK].block[pa & PAGE_OFFSET_MASK];
        msg_to_child.value[i] = read_value; // 읽은 값을 메시지에 저장
        fprintf(fp, "[Parent] Read Value: %c from PA: %08x\n", read_value, pa);
    }

    // 메시지에 처리 결과를 저장
    msg_to_child.virtual_address[i] = va;
    msg_to_child.type[i] = type;
}
```

Figure 9) Parent Process – Processes After Receiving VA and Before Sending PA

자식은 부모가 보낸 메시지를 받아서 실제 물리 메모리 값을 사용한다.

```c
// 부모로부터 결과 수신
MsgBuf2 response_from_parent;
if (msgrcv(msgq_id_parent_to_child, &response_from_parent, sizeof(MsgBuf2) - sizeof(long), pid, 0) != -1) {
    // 부모로부터 받은 결과 출력
    for (int i = 0; i < NUM_MEMORY_ACCESS; i++) {
        int va = response_from_parent.virtual_address[i];
        char type = response_from_parent.type[i];
        char value = response_from_parent.value[i];

        if (type == 'r') { // 읽기 작업일 경우만 value 출력
            fprintf(fp, "Child %d received: VA[%08x], Read Value from Memory = %c, Type = %c\n",
                    pid, va, value, type);
        } else if (type == 'w') { // 쓰기 작업일 경우 value를 출력하지 않음
            fprintf(fp, "Child %d received: VA[%08x], Type = %c\n",
                    pid, va, type);
        }
    }
}
```

Figure 10) Child Process – receive PA value from parent

Unique features

2-level paging 은 기존 1-level paging 의 한계를 극복하기 위해 페이지 테이블을 관리하는 추가적인 페이지 디렉토리를 도입한 구조이다. Virtual address 의 상위 12 비트를 사용하여 페이지 디렉토리의 위치를 계산하고, 다음 8 비트를 이용해 해당 디렉토리가 가리키는 L2 페이지 테이블에 접근한다. 이러한 구조는 L2 테이블을 필요할 때만 동적으로 생성하여 메모리 낭비를 줄이고, 더 유연하고 효율적인 메모리 관리를 가능하게 한다. 특히, 1-level paging 에서는 모든 페이지 테이블을 한 번에 메모리에 적재해야 하므로 대규모 주소 공간에서는 비효율적일 수 있지만, 2-level paging 은 실제로 필요한 테이블만 메모리에 로드하여 더 높은 메모리 활용도를 보여준다.

또한, 2-level paging 은 스왑 작업에서도 큰 장점을 제공한다. 필요한 페이지가 물리 메모리에 없을 경우, 디스크에서 데이터를 가져와 L2 테이블을 동적으로

업데이트한다. 이를 통해 프로세스 간 메모리 격리를 확실히 보장하고, 큰 주소 공간을 지원하며 메모리 관리의 복잡성을 줄인다.

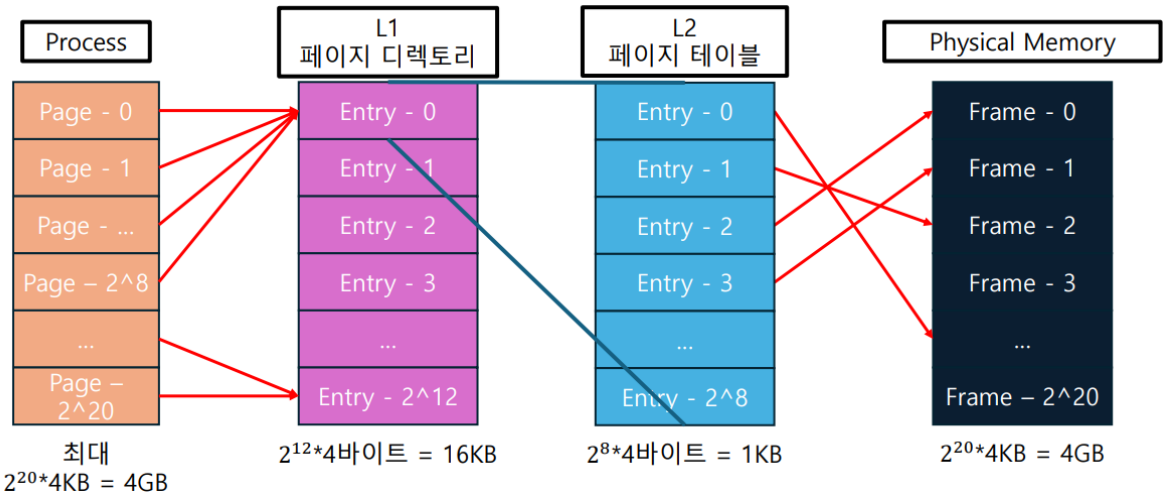

Figure 11) 2-level paging

스와핑의 구현은 다음과 같은 방식으로 이루어졌다. Swapping 기능을 구현하기 위해 먼저 RAM 용량을 기존의 4GB 에서 1GB 로 줄이고, 대신 4GB 용량의 디스크를 추가하여 부족한 메모리를 보완할 수 있도록 설정하였다. RAM 이 부족한 상황에서 페이지 폴트가 발생하면, 메모리 내에서 덜 자주 사용된 페이지를 디스크로 이동시키는 swap-out 작업이 수행된다. 이 과정에서 단순히 임의의 프레임(Frame)을 선택하는 것이 아니라, 가장 오랫동안 사용되지 않은 프레임을 선택하여 제거하도록 LRU(Least Recently Used) 알고리즘을 적용하였다.

LRU 알고리즘은 메모리 접근 기록을 기반으로 페이지 교체 대상을 선택하며, 최근 사용된 페이지는 유지하고 오래 사용되지 않은 페이지를 우선적으로 교체하는 방식으로 메모리 사용의 효율성을 높인다. 디스크로 이동된 페이지는 필요할 때 다시 RAM 으로 불러오는 swap-in 작업을 통해 재활용되며, 이 과정에서 페이지 테이블의 상태도 함께 업데이트된다. 이를 통해 제한된 RAM 환경에서도 효율적인 메모리 관리를 구현할 수 있었으며, 스와핑은 디스크와 RAM 간의 데이터 교환을 원활히 수행한다.

결과적으로, 이러한 구현은 메모리가 부족한 환경에서도 프로세스의 실행을 안정적으로 지원하고, 메모리 자원의 활용도를 극대화하였다.

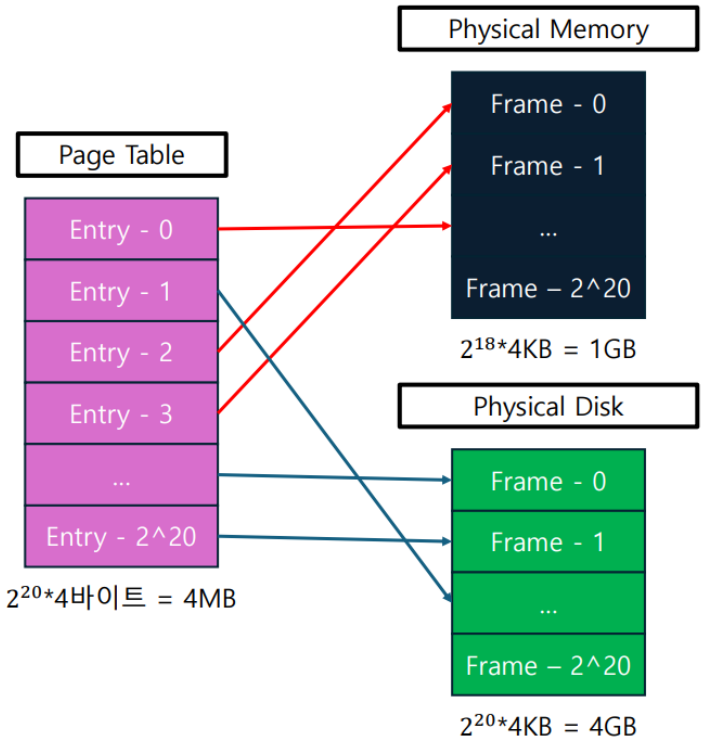

Figure 12) Swapping logic

스와핑이 적용된, 페이지 테이블에 담긴 실제 값은 다음과 같다. 상위 첫 번째 비트는 Valid bit, 상위 두 번째 비트는 RAM/DISK bit, 하위 20 개 비트는 FRAME NUMBER 를 나타낸다.

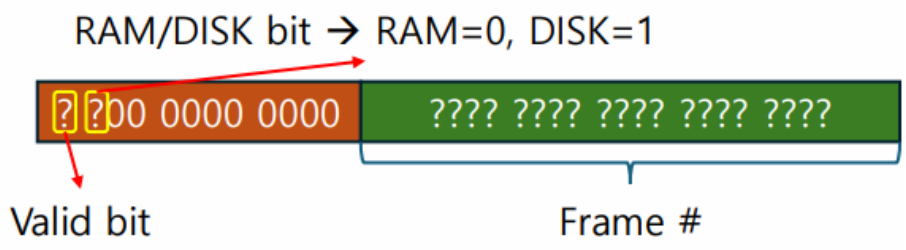

Figure 13) Swapping Frame value

swap_in()은 translate_VA_to_PA() 함수에서 가상 주소(VA)가 가리키는 페이지가 RAM 에 존재하지 않고 DISK 에 있을 때 호출된다. 이 함수는 DISK 에 저장된 페이지를 RAM 으로 불러와 해당 페이지의 데이터를 복원하는 역할을 수행한다. 또한, swap_in() 호출 후에는 페이지 테이블 엔트리를 업데이트하여 페이지가 RAM 으로 이동했음을 반영하고, 해당 프레임의 유효 비트를 설정한다.

```
// DISK에 접근하는 경우 스왑 인 필요
if ((L2[l2_index] & VALID_BIT_MASK) && (L2[l2_index] & RAM_DISK_BIT_MASK)) {
    frame = swap_in(frame);
    L2[l2_index] = (frame & FRAME_NUMBER_MASK) | VALID_BIT_MASK;
}
```

Figure 14) Swap in – translate_VA_to_PA

반면, swap_out()은 RAM 이 가득 찬 상황에서 handle_page_fault() 함수에 의해 호출되며, 가장 오랫동안 사용되지 않은 프레임을 선택하여 DISK 로 이동시키는 역할을 한다. 이 과정에서 선택된 프레임의 데이터를 DISK 에 저장하고, 페이지 테이블 엔트리를 수정하여 해당 페이지가 DISK 에 있음을 표시한다.

```
// 새로운 프레임 할당
int frame = allocate_frame();
if (frame == -1) { // RAM이 부족하면 스왑 아웃 수행
    printf("RAM is full. Performing swap out.\n");
    swap_out();                 // 가장 오래된 프레임을 DISK로 이동
    frame = allocate_frame();   // 다시 프레임 할당 시도
    if (frame == -1) {
        printf("Error: No available frames even after swap out\n");
        exit(1);
    }
}
```

Figure 15) Swap out – handel_page_fault

Outro

Environment

- Linux environment. Basically, Local Ubuntu with window and MAC.
- To Run code, Place the swapping.c and swapping.h files inside the same folder.
- Compiler it and execute out file.
  - Time tick, completed processes, Run-Queue status, Wait-Queue status.
  - Process movement (Run/Wait)
  - Process CPU/IO burst shift
  - Running time, waiting time, CPU use time
  - (추가) The log, showing the child process randomly generating virtual addresses, values to be stored in RAM, and operation types indicating either a 'write' to memory or a 'read' from RAM.
  - (추가) The log, showing the parent process accessing RAM to either write or read values.

Screenshots

동일한 VA 접근에 대해 제대로 PA 변환이 이루어지고, 값의 Write/Read 이 동작하는지 확인해보자. 초기 VA 접근에 대해 Page Fault 가 발생하게 되고, 새로운 빈 프레임이 할당된다. Write mode 를 통해 특정 값이 실제 물리 메모리 위치에 저장된다. 이후 동일한 VA Read mode 접근을 통해 이전에 넣어둔 값을 제대로 불러오는 것을 확인할 수 있다.

또한 출력 주석처리를 통해 특정 'Swap In/Swap Out'을 확인할 수 있다. 정상적으로 프레임 값 복사와 스왑이 일어난다.

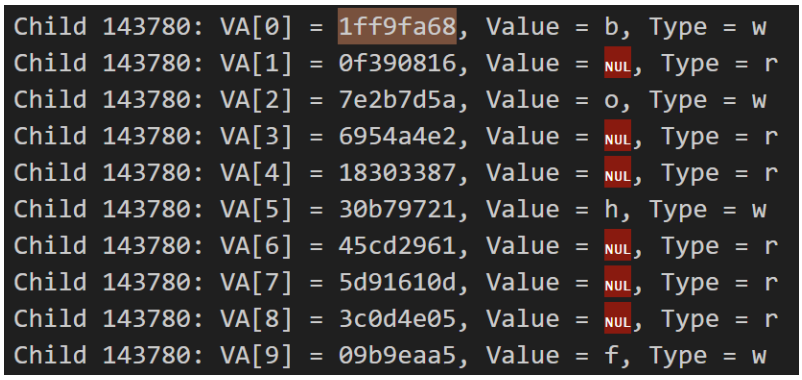

Figure 16) 자식 프로세스의 메모리 접근 요청

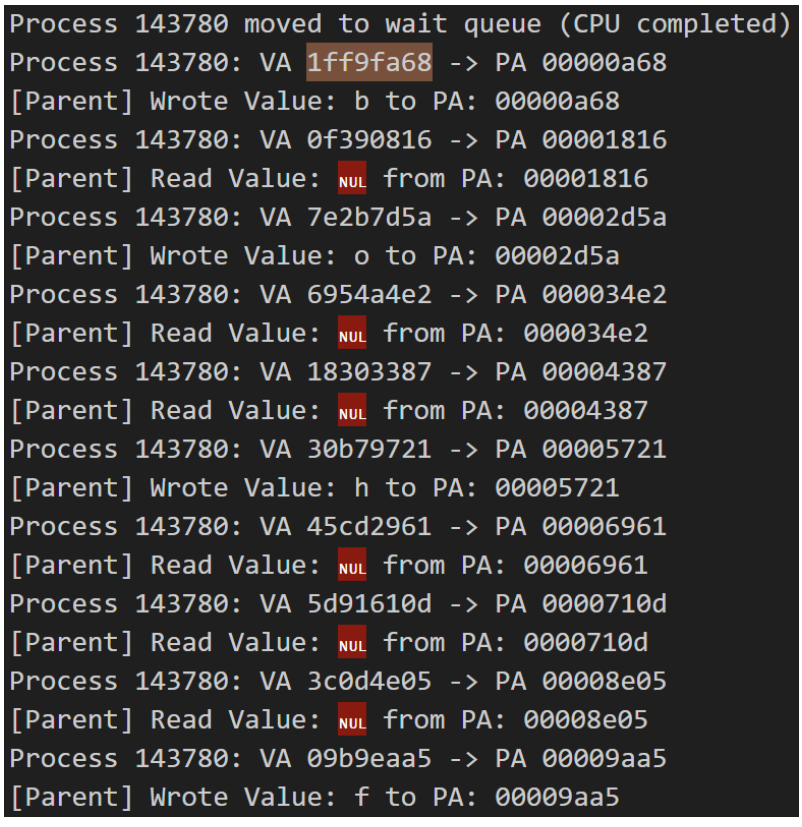

Figure 17) 부모 프로세스의 메모리 접근 처리

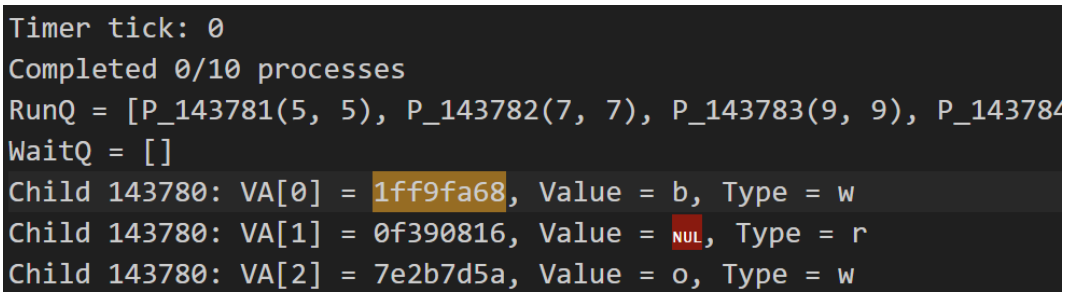

Figure 18) 결과 검증 - 초기 VA 접근

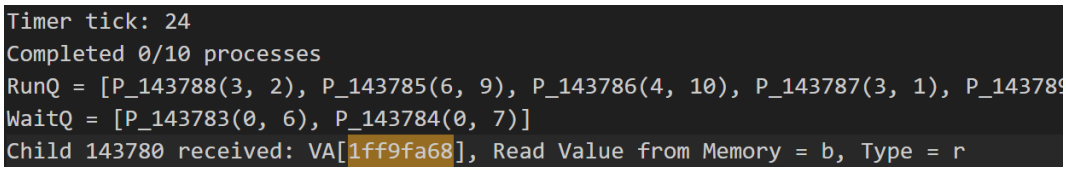

Figure 19) 결과 검증 - 이후 VA 접근

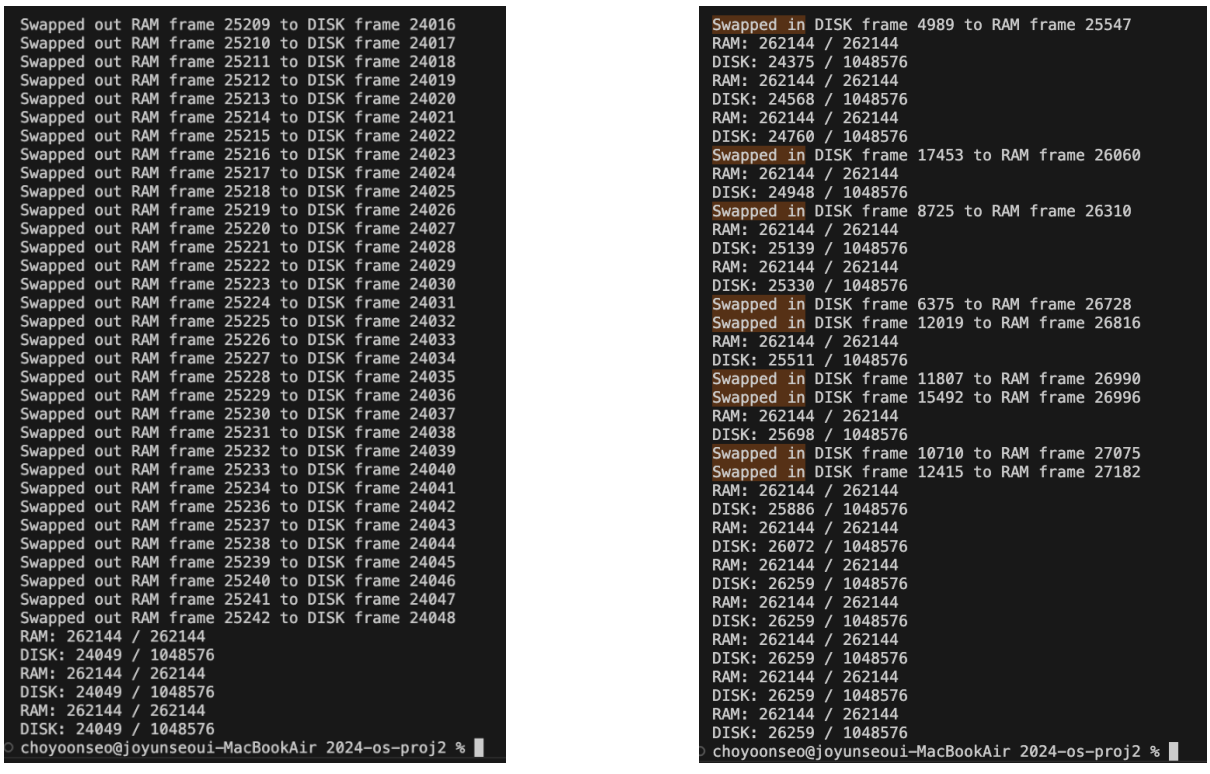

Figure 20) Swapping example