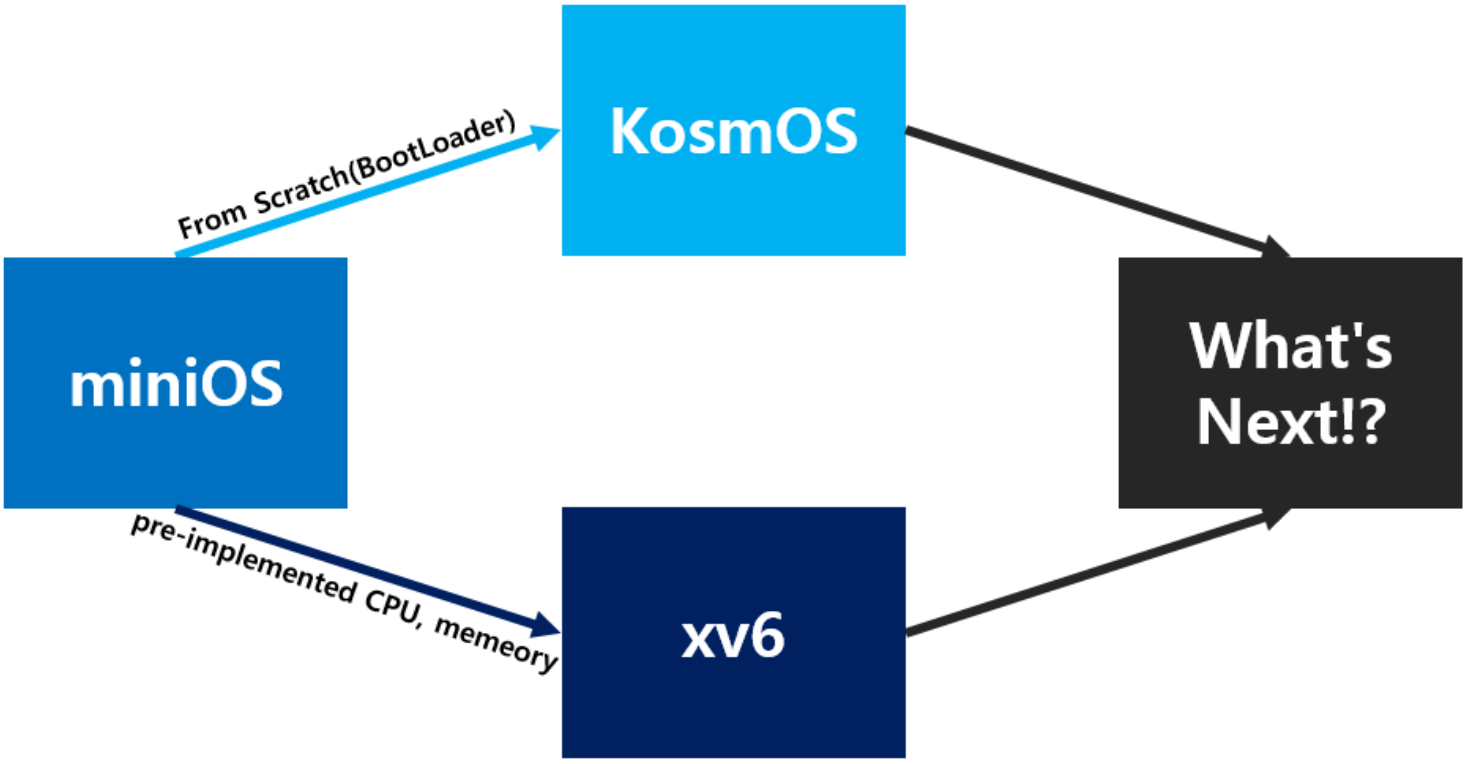
[운영체제 (Operating System)]

6팀 최종 결과 보고서



소속: 전자정보공학부(IT융합)

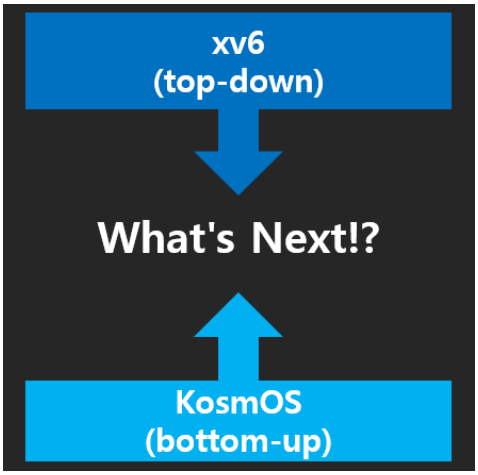
팀장: 송준규 (20201590)

팀원: 박태현 (20201588)

팀원: 안영기 (20112179)

팀원: 정하연 (20201610)

[Index]



1. 프로젝트 개요 및 최종 결과 요약

1-1. 프로젝트 개요

1-2. 프로젝트 최정 결과 요약

2. miniOS

2-1. miniOS 개요

2-2. miniOS Load & Execute Scenario Test

2-3. Conclusion of miniOS

3. KosmOS

3-1. KosmOS 개요

3-2. KosmOS "memory map & first kerner with monitor"

3-3. KosmOS "print & mouse & interrupt"

4. xv6

4-1. xv6 개요

4-2. xv6 "Round-Robin based Priority Scheduler"

4-3. xv6 "Copy on Write"

4-4. Conclusion of xv6

5. 프로젝트 결론 및 피드백

Appendix.1: 주차별 회의록

Appendix.2: 수행 일지 및 역할 분담

1. 프로젝트 개요 및 최종 결과 요약

1-1. 프로젝트 개요

저희 6팀에서는 작동하는 OS를 구현해보자는 목표를 가지고, 팀을 결성하였습니다. 이를 위해서, 결정해야할 사안이 몇 가지 있었습니다. OS를 얼만큼 구현 할 것인지, 어떻게 OS를 구현할 것인지 이렇게 두가지를 고민하였고, 강의를 통해 배운 내용을 최대한 구현해 봄에 의의를 두고, 프로젝트를 진행했습니다. 그렇게 설정된 1차 프로젝트의 목표는 아래와 같습니다.

"강의 시간에 배운 내용이 구현된, 작동하는 OS를 구현해보자"

하지만, 위의 목표를 이루기에는 많은 시행착오와 시간이 필요했기에 후술될 이유로 방향성과 목표가 재설정 되었습니다. 결론적으로 저희 6팀은 아래의 목표를 현재까지 최종 목표로 선정하여 프로젝트를 완성했습니다.

"강의 시간에 배운 내용을 구현체 위에 top-down 방식으로 구현하고,
from scratch 또한 bottom-up 방식으로 구현하여, 두 프로젝트를 병합하면,
작동하는 OS를 구현해 볼 수 있다."

1-2. 프로젝트 최종 결과 요약

저희팀은 총 3가지 결과물을 완성했습니다. 1. miniOS Test Scenario, 2. xv6, 3. KosmOS 각 결과물에대한 간단한 설명과 결론은 아래와 같습니다.

1. miniOS: Linux위에서 C언어만으로 OS에서 배운 Process와 Memory관련 Mechanism 들을 구현할 수 있는지 확인해 보기 위해, 최대한 pointer와 함수 호출만으로 stack pointer 및 excutable file 실행 Scenario를 구현해 보려 했으나, 레지스터의 활용이 제한적이고, 세부적인 HW 자원을 관리할 수 없어서, dynamic linker 및 loader의 내부 동작을 디테일 하게 수정할 수 없다는 이유로 프로젝트의 진행이 어렵다는 것을 확인했습니다. 이는 Dynamic linking이 필요한 excutable 파일과 필요 없는 excutable의 실행 결과를 통해 결론 짓게 되었습니다

2. xv6: 앞선 miniOS로 프로젝트의 디테일한 구현이 어려워, xv6라는 구현체를 사용했습니다. 프로젝트의 방향은 default xv6의 비효율 적인 부분을 찾아 개선하는 방향으로 진행했습니다. Process 와 Memory를 큰 주제로 잡아 개선해 나갔고, Process 관련해서는 기존의 Round Robin 기반 scheduler를 Round Robin based Priority Scheduler로 수정했습니다. Memory 관련해서는 Fork 시 Page를 그대로 복사하는 기존 Mechanism에서 Page를 효율적으로 사용하기 위해 "Copy-on-Write" Mechanism을 구현하여 개선했습니다.

3. KosmOS: 구현체로만 학습할 수 있는데에는 한계가 있으며, 아무런 구현체의 도움없이 Scratch 부터 구현해보자는 목표로 시작한 KosmOS는 현재 부트로더를 통해 초기 커널을 불러 오고, 커널에서 문자열을 출력할 수 있으며, Polling 기법이 아닌 인터럽트 핸들러와 디바이스 드라이버의 구현을 통해 마우스의 동작 까지 구현이 완료된 상태입니다.

2. miniOS (https://github.com/AltairKosmoTale/miniOS/tree/main/test_02)

2-1. miniOS 개요

miniOS는 Linux위에 C언어로 기능을 구현하는 방향으로 시작된 프로젝트입니다.
우선 저희가 처음으로 집중한 부분은 8주차의 선택 과제였던 "일부 메모리를 설정하여 프로세스를 로딩하는 miniOS를 설계"하는 부분입니다.

2. (선택) 일부 메모리 설정하여 프로세스 로딩하는 miniOS 설계하고 코드 제출. task_struct 사용할 것.

해당 명세를 구현하기 위해서는 결정해야 하는 사안이 많았습니다. 어떻게 메모리에 프로세스를 올릴지에 대한 방법과, 해당 프로세스를 어떻게 실행해야 하는지에 대한 방법 등을 결정해야 했습니다. 우선 저희는 실제 Linux kernel에서 해당 작업을 어떻게 처리 하는지를 최대한 따라 구현해보기로 결정했습니다.

Linux에서 실행파일은 "ELF" file format으로 구성되어 있습니다. 해당 값에는 순전히 machine에서 바로 작동할 수 있는 값만 작성되어 있는 것이 아니라, ELF header, Section Headers, Program Headers, 또한 헤더에 해당하는 다양한 section에 대한 값들이 작성 되어있습니다.

저희 팀의 **첫번째 시행착오는 "어떻게 ELF 파일 포멧을 메모리에 올려서 구조체를 잘 쪼개고, 해당 프로그램을 실행 시킬 수 있을까?" 에서 시작 되었습니다.** 앞서 언급 했듯이, 이를 구현하기 위해서는 실제로 Linux가 어떻게 동작하는지를 자세하게 조사해 보았습니다. 예를 들어, kernel(bash shell)에서 "$./ELF"를 구동 시켰을 때, kernel 에서는 process를 **fork**를 진행하고, **execve("./ELF" ~)**의 함수를 사용하여 process를 실행하게 됩니다. 이때 커널은 새로운 **process discriptor(task_struct)**를 생성하고, 부모 프로세스의 정보를 복사합니다. 또한 새로운 페이지 테이블을 생성하고, 부모 프로세스의 페이지 테이블을 복사합니다. 이 때, 페이지는 **Copy-on-Write(COW)** 방식으로 복사합니다. 새롭게 생성된 자식 프로세서의 **"CR3"** 레지스터는 새로운 페이지 테이블의 물리 주소를 가리키게 됩니다. **TSS(Task State Segment)** 또한 업데이트 됩니다. 커널에서 새로운 TSS를 생성하고, 새로운 프로세서의 커널 스택과 관련된 정보가 설정됩니다. 자식 프로세스가 execve를 한 시점에 현재 프로세스의 주소 공간을 새로운 실행 파일로 덮어 쓰게 됩니다. "./ELF"를 명령 했으니, 해당 파일을 읽고, ELF 헤더를 파싱하여 프로그램의 각 섹션을 해당하는 메모리 주소에 linker와 loader가 메모리에 로드해 줍니다. 이때 새로운 주소 공간이 할당 되는데, 기존 프로세스의 주소 공간을 해제하고, 새로운 주소 공간을 할당합니다. 해당 주소들은 ELF 파일의 코드, 데이터, 스택 등 적절한 메모리 영역에 매핑이 되며, CR3 레지스터는 새롭게 할당된 페이지 테이블의 물리 주소로 업데이트 됩니다. 이 시점에서 process discriptor와 TSS는 새로운 프로그램의 시작주소와 스택 포인터로 업데이트 됩니다. 실행 준비가 마쳐졌다면, 사용자 모드로 전환하기 위한 과정이 작동하게 됩니다. 커널은 사용자 모드로 전환하고 새롭게 로드된 프로그램의 시작 주소로 JMP 하게 됩니다. 이때 **"PC"** 레지스터는 ELF 파일의 엔트리 포인트 주소로 설정이 되며, ARM 아키텍처라고 가정 했을 때, **"CPSR"** 레지스터는 사용자 모드로 설정되고 인터럽트 플래그가 설정됩니다.

2-2. miniOS Load & Execute Scenario Test

단순하게 ./ELF에 대한 작동 방식이 복잡하지만, 구현이 가능할 것이라고 생각하여 저희는 해당 부분을 C코드로 구현해 나가기 시작했습니다. 우선 가설을 아래와 같이 설정했습니다.

"ELF 또는 machine native한 명령어들의 sequence를 파일에 저장한 후,
해당 파일을 C언어로 읽어서, 읽은 파일 중에 entry 포인트를 정확하게 함수 포인터로 설정해 PC 레지스터의 역할을 맡게 설정합니다. 그리고 해당 함수를 호출하면 원하는 프로그램이 실행될 것이다."

위의 가설을 증명하기 위해서 저희는 dynamic linking이 필요한 외부 라이브러리를 필요로 하는 C언어 코드와 외부 라이브러리를 사용하지 않는 C언어 코드를 간단하게 작성해 보았습니다.

withoutLibrary.c
```c
int main(){
    int i,j = 0;
    while (i < 1000){
        if (i%100==0)
            j++;
        i++;
    }
    return j;
}
```

hello.c
```c
#include <stdio.h>
int main() {
    printf("hello");
    return 0;
}
```

```
~/A/Senior_1st/OS/test_02 gcc -o withoutLibrary withoutLibrary.c
~/A/Senior_1st/OS/test_02 gcc -o hello hello.c
```

해당 파일들을 gcc를 활용하여 실행 가능한 파일로 컴파일 했습니다. (이하 실행 가능 파일을 [ELF]로 표기 합니다.) [ELF]들에 대해서 바로 실행하고 분석 할 수 있도록 HEX 파일로 변환했습니다. 변환 할 때는 아래와 같이 "xxd -p" 명령어를 사용했습니다.

```
~/A/Senior_1st/OS/test_02 xxd -p withoutLibrary | tr -d '\n' > input_hex.txt
```

HEX로 추출한 파일을 바탕으로 entry pointer를 확인 할 수 있었습니다. 확인할 때는 아래와 같이 "objdump -d" 명령어를 사용했습니다.

```
~/A/Senior_1st/OS/test_02 objdump -d hello | grep -A20 "<main>:"
0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 8d 3d ac 0e 00 00    lea    0xeac(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    1158:       b8 00 00 00 00          mov    $0x0,%eax
    115d:       e8 ee fe ff ff          callq  1050 <printf@plt>
    1162:       b8 00 00 00 00          mov    $0x0,%eax
    1167:       5d                      pop    %rbp
    1168:       c3                      retq
    1169:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

```
~/A/Senior_1st/OS/test_02 objdump -d withoutLibrary | grep -A30 "<main>:"
0000000000001129 <main>:
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
    1131:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    1138:       eb 32                   jmp    116c <main+0x43>
    113a:       8b 55 f8                mov    -0x8(%rbp),%edx
    113d:       48 63 c2                movslq %edx,%rax
    1140:       48 69 c0 1f 85 eb 51    imul   $0x51eb851f,%rax,%rax
    1147:       48 c1 e8 20             shr    $0x20,%rax
```

각각 dynamic linking이 필요한 프로그램은 0x1149에, 필요없는 프로그램은 0x1129에 entry point가 있음을 확인할 수 있었습니다. 이제 xxd로 변환한 HEX값을 C언어에서 읽기 편하게 만드는 작업이 필요합니다. 저희는 이 부분에서 "elf_to_hex".py를 활용하여 진행 했습니다.

elf_to_hex.py

```python
# -*- coding: utf-8 -*-
def hex_string_to_c_array(hex_string):
    # 16진수 문자열을 두 글자씩 분리하여 리스트 생성
    hex_bytes = [hex_string[i:i+2] for i in range(0, len(hex_string), 2)]
    # C언어 스타일의 바이트 배열로 변환
    formatted_bytes = ', '.join("0x{}".format(byte) for byte in hex_bytes)
    # 완성된 문자열을 반환
    return "{{ {} }};".format(formatted_bytes)
def read_from_file(file_path):
    # 파일 읽기
    with open(file_path, 'r') as file:
        return file.read().strip()
def write_to_file(file_path, data):
    # 파일 쓰기
    with open(file_path, 'w') as file:
        file.write(data)
# 입력 파일과 출력 파일 경로 설정
input_file_path = 'input_hex.txt'
output_file_path = 'output_array.txt'
# 입력 파일에서 16진수 문자열 읽기
input_hex_string = read_from_file(input_file_path)
# 16진수 문자열을 C언어 스타일의 바이트 배열로 변환
result_array = hex_string_to_c_array(input_hex_string)
# 결과를 출력 파일에 쓰기
write_to_file(output_file_path, result_array)
print(f"Converted hex array has been written to {output_file_path}")
```

각각 dynamic linking을 사용하지 않은 코드를 input.hex, output_array.txt로,
사용한 코드를 input2.hex, output_array2.txt로 추출한 결과 입니다.

```
~/A/Senior_1st/OS/test_02 python elf_to_hex.py
Converted hex array has been written to output_array.txt
Converted hex array has been written to output_array2.txt
```

output_array.txt

```
{ 0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

output_array2.txt

```
{ 0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
~/A/Senior_1st/OS/test_02 ls -l | grep output
-rw-r--r-- 1 altair altair 100179 May 22 16:34 output_array2.txt
-rw-r--r-- 1 altair altair  98835 May 22 16:34 output_array.txt
```

추출한 array를 ELF가 아닌 단순 기계어 파일(.bin)으로 변환하는 작업이 필요합니다.
아래의 코드를 사용하여, 변환 작업을 거쳤습니다. 이때, data[] 부분에 이전에 추출 했던 각각의 output_array의 값을 입력하여 출력하였습니다.

c_file.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
        int file_num=0;
        unsigned char data[] = { 0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00,
        char file_name[1024]; // 충분한 크기의 배열을 선언
        printf("%ld\n", sizeof(data));
        sprintf(file_name, "execute_%d.bin", file_num);

        int fd = open(file_name, O_CREAT | O_WRONLY, 0644);
        if (fd < 0) {
            perror("Failed to create file");
            return 1;
        }
        write(fd, data, sizeof(data));
        close(fd);

    return 0;
}
```

각각 output_hex1, 2에 대한 txt를 array로 가진 bin 파일을 생성하는 c_file.c를 컴파일 하여, bin 파일 2개를 추출했습니다.

```
 ~/A/Senior_1st/OS/test_02  gcc -o c_file c_file.c
 ~/A/Senior_1st/OS/test_02  ./c_file
16472
16696
 ~/A/Senior_1st/OS/test_02  ls | grep .bin
execute_1.bin
execute_2.bin
```

다음에 필요한 일은 해당 파일을 메모리에 읽어서 로드하고, 함수를 스택 포인터 대신 활용해 줄 프로그램을 작성해서 실행하는 일 입니다.

저희는 실행 해야하는 파일을 "to_do_list"라는 디렉토리를 생성해 옮겼고, "to_do_list"에 해당하는 bin 파일을 메모리가 읽어와서 task_struct 개념을 활용하여 scheduling 하고 실행되는 환경을 구축하고 싶어 해당하는 코드를 아래와 같이 "scenario.c"라는 프로그램으로 작성 했습니다.

코드는 아래와 같습니다.

```c
typedef struct task_struct {
    void *stack;
    unsigned long state;  // 0: runnable, 1: running, 2: terminated
    struct task_struct *next;
} task_struct;

task_struct *head = NULL;

// 프로세스를 실행하는 함수
int execute_process(task_struct *task) {
    if (task->state != 0 && task->state != 1) {
        fprintf(stderr, "Task cannot be run\n");
        return -1;
    }
    task->state = 1; // running

    //int (*func)() = (int (*)())task->stack;  // 함수 포인터로 변환
    // int (*func)() = (int (*)())(task->stack+0x1149);
    int (*func)() = (int (*)())(task->stack+0x1129);
    int result = func();  // 함수 실행
    task->state = 2; // terminated

    return result;
}
```

구현의 편의상 **"task_struct"**에는 실행 되어야 하는 기계어를 담은 stack, state, 스케쥴링을 위한 포인터만 간단하게 작성했습니다. 프로세스를 실행하는 **"execute_process"** 함수에서는 스택 포인터 구현을 위해 stack의 위치를 main entry 만큼 더해주는 코드를 작성했습니다. 이때 작성한 offset 주소 값은 앞서 "objdump"를 통해 알게 되었던 값을 활용하였습니다.

이후로는 프로세스를 생성하는 **"create_process"** 함수를 작성했습니다.

```c
// 새로운 프로세스 생성
task_struct *create_process(char *binary_code, size_t code_size) {
    long pagesize = sysconf(_SC_PAGESIZE)*250;
    void *stack = mmap(NULL, pagesize, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANON | MAP_PRIVATE, -1, 0);
    if (stack == MAP_FAILED) {
        perror("mmap");
        return NULL;
    }
    memcpy(stack, binary_code, code_size);
    task_struct *new_task = malloc(sizeof(task_struct));
    if (new_task == NULL) {
        munmap(stack, pagesize);
        return NULL;
    }
    new_task->stack = stack;
    new_task->state = 0;  // runnable
    new_task->next = NULL;
    if (head == NULL) {
        head = new_task;
    } else {
        task_struct *temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_task;
    }
    return new_task;
}
```

위의 두가지 함수에서 값을 잘 받아서 사용할 수 있도록 아래와 같이 파일로 부터 바이너리 코드를 읽고 프로세스를 생성을 준비 하는 함수 **"load_and_create_process"**를 구현 했습니다.

```c
// 파일로부터 바이너리 코드 읽기 및 프로세스 생성
void load_and_create_process(const char *filename) {

    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("Failed to open file");
        return;
    }
    char buffer[1000000];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
    if (bytes_read <= 0) {
        perror("Failed to read from file");
        close(fd);
        return;
    }
    //create_process(buffer, bytes_read);  // bytes_read에 널 문자를 포함하여 전달
    create_process(buffer, bytes_read);
    close(fd);
}
```

실행 Test를 위해 간단한 **Round-Robin** 기반 **"run_scheduler"** 함수를 구현 했습니다.

```c
// Round-robin 스케줄러 실행
void run_scheduler() {
    task_struct *current = head;
    while (current != NULL) {
        if (current->state == 0 || current->state == 1) {
            int result = execute_process(current);
            printf("Process returned: %d\n", result);
        }
        current = current->next;
    }
}
```

main에서는 *.bin을 읽어, load -> create -> schedule -> run 될 수 있도록 구현했습니다.

```c
int main() {
    const char *dir_path = "to_do_list";
    DIR *dir = opendir(dir_path);
    if (!dir) {
        perror("Failed to open directory");
        return 1;
    }
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type == DT_REG) {  // 일반 파일인 경우
            // 파일 확장자가 .bin인지 확인
            const char *ext = strrchr(entry->d_name, '.');
            if (ext && !strcmp(ext, ".bin")) {
                char full_path[1024];
                snprintf(full_path, sizeof(full_path), "%s/%s", dir_path, entry->d_name);

                load_and_create_process(full_path);
            }
        }
    }
    closedir(dir);
    run_scheduler(); // 스케줄러 실행
    return 0;
}
```

우선 최종적으로 Test 해보려고 했던 dynamic linking을 사용/미사용 코드가 아니라, 단순하게 return 1~10을 진행하는 .bin 파일 10개를 생성하여 실행이 잘 되는지 test 해보았습니다.

data[] 배열에 "return i"에 해당하는 값을 linux machine로 구성하는 값을 담아주었습니다.

return_10_c_file.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    for (int i = 1; i < 10; i++) {
        char data[] = {0x48, 0xC7, 0xC0, i, 0x00, 0x00, 0x00, 0xC3};
        char file_name[20]; // 충분한 크기의 배열을 선언
        sprintf(file_name, "execute_%d.bin", i); // 파일 이름 형식화

        int fd = open(file_name, O_CREAT | O_WRONLY, 0644);
        if (fd < 0) {
            perror("Failed to create file");
            return 1;
        }
        write(fd, data, sizeof(data));
        close(fd);
    }
    return 0;
}
```

```
 ~/A/Se/OS/test_02/to_do_list ./return_10_c_file
 ~/A/Se/OS/test_02/to_do_list ls
execute_1.bin  execute_3.bin  execute_5.bin  execute_7.bin  execute_9.bin
execute_2.bin  execute_4.bin  execute_6.bin  execute_8.bin  return_10_c_file
```

단, scenario.c의 stack pointer를 offset을 0으로 해야 실행이 가능하기에, 재컴파일 하였습니다.

```c
int (*func)() = (int (*)())task->stack;  // 함수 포인터로 변환
// int (*func)() = (int (*)())(task->stack+0x1149);
//int (*func)() = (int (*)())(task->stack+0x1129);
int result = func();  // 함수 실행
```

컴파일 후, 실행 했을 때, bin 파일을 읽고, 메모리에 로드해 스케쥴링이 잘 된 것을 확인 할 수 있었습니다.

```
 ~/A/Senior_1st/OS/test_02 gcc -o scenario scenario.c
 ~/A/Senior_1st/OS/test_02 ./scenario
Process returned: 4
Process returned: 5
Process returned: 6
Process returned: 9
Process returned: 3
Process returned: 8
Process returned: 1
Process returned: 2
Process returned: 7
```

남은 일은 c언어로 작성한 code로 부터 뽑은 bin 파일이 되는지 확인하는 일입니다. 다시 scenario.c의 스택 포인터를 수정하여 재컴파일을 진행했습니다. 또한 "to_do_list" 디렉토리에 이전에 생성했던 bin파일을 제거하고, 따로 따로 외부 라이브러리 사용.bin 미사용.bin을 넣어 가며 테스트를 진행 했습니다.

```
 ~/A/Senior_1st/OS/test_02 ./scenario
Process returned: 0
 ~/A/Senior_1st/OS/test_02 ./scenario
[1]    1214 segmentation fault  ./scenario
```

결론적으로 외부 라이브러리를 사용하는 실행 파일은 segmentation fault를 출력했습니다.

혹시나 하는 마음에 gcc option 중에서 -static을 추가하여 dynamic linking을 static linkin으로

변경 해보았으나 그럼에도 결과는 동일 했습니다. 디버깅을 위해 GDB를 사용하여 어디에서 segmetation fault가 일어 났는지 살펴 보았습니다.

```
Reading symbols from ./scenario...
(gdb) break main
Breakpoint 1 at 0x168a: file scenario.c, line 97.
(gdb) run
Starting program: /home/altair/Altair/Senior_1st/OS/test_02/scenario

Breakpoint 1, main () at scenario.c:97
97      int main() {
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) step
98          const char *dir_path = "to_do_list";
(gdb) step
99          DIR *dir = opendir(dir_path);
(gdb) step
__opendir (name=0x555555556060 "to_do_list") at ../sysdeps/posix/opendir.c:88
88      ../sysdeps/posix/opendir.c: No such file or directory.
(gdb) step
89      in ../sysdeps/posix/opendir.c
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000003000 in ?? ()
```

GDB를 통해 분석한 결과 line 30, 스택 포인터 호출 부분에서 문제가 있었다는 것을 backtrace 명령어를 통해 확인할 수 있었습니다.

```
(gdb) backtrace
#0  0x0000000000003000 in ?? ()
#1  0x00007ffff7cc8162 in ?? ()
#2  0x00007fffffffdd20 in ?? ()
#3  0x00005555555553e3 in execute_process (task=0x1664da8c9) at scenario.c:30
Backtrace stopped: frame did not save the PC
```

```
28      int (*func)() = (int (*)())(task->stack+0x1149);
29      //int (*func)() = (int (*)())(task->stack+0x1129);
30      int result = func();  // 함수 실행
```

debugging log인 "frame did not save the PC"는 대게, 스택 손상 여부 및 비표준 호출 규약에 의한 에러입니다. 저희 팀은 에러를 잡고, 구현하기 위한 노력을 진행했으나, register를 직접 지정하지 않는 이상 불가능 함을 깨달았고, 해당 기능에 대한 구현은 중단하게 되었습니다.

2-3. Conclusion of miniOS

하지만 **이 과정에서 앞서 언급했던 bash shell이 어떻게 "./ELF"를 실행하는지에 대해 매우 자세하게 알 수 있었고, 이를 위해서는 regsiter를 직접 수정하고 조작 할 수 있는 환경이 갖춰져야 한다는 결론을 도출할 수 있었습니다.** OS를 함수 호출 기반의 동작 구현이 아닌 제대로 완성을 하기 위해서는 다른 대안을 찾아야 했습니다. **"register와 특정 메모리가 구현된 구현체가 따로 없을까?"** 고민하던 저희 팀은 **"emulator"**를 채택함으로써 다음 진행 방향을 정할 수 있었습니다. 저희가 채택한 에뮬레이터는 **"qemu"**입니다. 해당 에뮬레이터는 OS에서 배운 내용을 모두 구현해 볼 수 있는 정도로 CPU 및 메모리가 가상화 되어 구현이 완성된 상태였습니다. 그러명 어떤 것을 해볼 수 있을까 고민하다 일단 팀원 별로 **"bottom-up과 top-down을 구성해 나가자는 의견이 나왔고 해당 부분을 merge 한다면 완성도 높은 OS를 구현할 수 있을 것 같다"**는 결론이 나왔습니다. 이에 아래와 같이 두가지 프로젝트를 진행하게 되었습니다.

3. KosmOS https://github.com/AltairKosmoTale/KosmOS/

3-1. KosmOS 개요

Bottom-up으로 OS를 구현하기 위해 from scratch OS를 작성하기 시작했습니다. 우선 Bottom-up에서 시작을 한다면 어떤 것을 구현 할 수 있을지 고민하고, Boot Loader와 초기 Kernel을 구현하는 것을 첫번째 목표로 하여 프로젝트를 진행했습니다. Boot Loader를 구성하기 위해선 별개의 tool과 API를 필요로 했습니다. 이에 저희 팀은 **"EDKII"**와 **"UEFI API"**를 활용하여 구현을 하였습니다. 구현을 하기 위해서는 UEFI Official 문서를 학습해야 했고,

<https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf>

위의 자료를 활용하여 UEFI API의 사용법을 조사 및 학습하고, 용례를 활용하여 Boot Loader를 구성해 나갔습니다. 필요한 자료와 스펙이 많아 사용한 구조체 및 자료에 한해서 간략하게 아래에 첨부했습니다. 주로 사용한 부분은 매모리 맵 관련된 구조체 입니다.

```
[1]
#include <Uefi.h>
/*14 #ifndef __PI_UEFI_H__
15 #define __PI_UEFI_H__
17 #include <Uefi/UefiBaseType.h> [1-1]
18 #include <Uefi/UefiSpec.h> [1-2]
20 #endif*/
```

```
[2]
#include <Library/UefiLib.h>
/* Provides library functions for common UEFI operations.
Only available to DXE and UEFI module types.
EFI_UNICODE_STRING_TABLE; 구조체 정의
EFI_LOCK_STATE; 구조체 정의
EFI_LOCK;
/* EFI_STATUS EFIAPI 구조체 ( // 여러가지 구조체 구현되어 있음
( IN / OUT/ IN OUT) TYPE Variable
);*/ */
```

```
[1-1]
#include <Uefi/UefiBaseType.h> // EFI_STATUS형식 여기서 지정 (typedef RETURN_STATUS EFI_STATUS, Base.h)
/*15 #include <Base.h> [1-1-1]
24 typedef GUID                  EFI_GUID;  // 128-bit buffer containing a unique identifier value.
28 typedef RETURN_STATUS         EFI_STATUS; // Function return status for EFI API.
32 typedef VOID                  *EFI_HANDLE; // A collection of related interfaces.
36 typedef VOID                  *EFI_EVENT; // Handle to an event structure.
40 typedef UINTN                 EFI_TPL; // Task priority level.
44 typedef UINT64                EFI_LBA; // Logical block address.
49 typedef UINT64                EFI_PHYSICAL_ADDRESS; // 64-bit physical memory address.
54 typedef UINT64                EFI_VIRTUAL_ADDRESS; // 64-bit virtual memory address.
(EFI_Time, IP, IP-Union etc...)*/

[1-1-1]
$HOME/edk2/MdePkg/Include/Base.h // Base Type 정리
typedef UINTN RETURN_STATUS // 부호 없는 정수형
```

```
[1-2]
#include <Uefi/UefiSpec.h>
/*18 #include <Uefi/UefiMultiPhase.h>
20 #include <Protocol/DevicePath.h>
21 #include <Protocol/SimpleTextIn.h>
22 #include <Protocol/SimpleTextInEx.h>
23 #include <Protocol/SimpleTextOut.h>
EFI_ALLOCATE_TYPE; 구조체 정의
여러가지 define for memory
EFI_MEMORY_DESCRIPTOR; 구조체 정의
여러가지 definitions for EFI_STATUS
EFI_RUNTIME_SERVICES; 구조체 정의
EFI_BOOT_SERVICES; 구조체 정의
EFI_CONFIGURATION_TABLE; 구조체 정의
EFI_SYSTEM_TABLE; 구조체 정의
EFI_LOAD_OPTION; 구조체 정의
EFI_BOOT_KEY_DATA; 구조체 정의
EFI_KEY_OPTION; 구조체 정의
2234 #include <Uefi/UefiPxe.h>
2235 #include <Uefi/UefiGpt.h>
2236 #include <Uefi/UefiInternalFormRepresentation.h> */
```

```
[gBS->GetMemoryMap()의 구조]
typedef
EFI_STATUS
(EFIAPI *EFI_GET_MEMORY_MAP) (
 IN OUT UINTN *MemoryMapSize,
 OUT EFI_MEMORY_DESCRIPTOR *MemoryMap
 OUT UINTN *MapKey,
 OUT UINTN *DescriptorSize,
 OUT UINT32 *DescriptorVersion
 );

[메모리 맵 구조체 from Main.c]
struct MemoryMap {
    UINTN buffer_size;
    VOID* buffer;
    UINTN map_size;
    UINTN map_key;
    UINTN descriptor_size;
    UINT32 descriptor_version;
};

[EFI_MEMORY_DESCRIPTOR 구조체]
typedef struct {
 UINT32 Type;
 EFI_PHYSICAL_ADDRESS PhysicalStart;
 EFI_VIRTUAL_ADDRESS VirtualStart;
 UINT64 NumberOfPages;
 UINT64 Attribute;
} EFI_MEMORY_DESCRIPTOR;
```

구현을 진행할 때에는 qemu의 기능인 "info register"를 활용해 monitoring을 할 수 있었습니다.

```
(qemu) info registers
RAX=000000000043282e RBX=00000000000da7a6 RCX=000000000000adba RDX=000000000000b008
RSI=000000000000b008 RDI=000000003f2547a8 RBP=000000000043d5e8 RSP=000000003feac410
R8 =000000003feac437 R9 =0000000000000001 R10=0000000000000000 R11=000000000000ffff
R12=0000000000000002 R13=000000003feac4a6 R14=000000000003d090 R15=0000000000000000
RIP=000000003ea3bd2b RFL=00000246 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0030 0000000000000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0038 0000000000000000 ffffffff 00af9a00 DPL=0 CS64 [-R-]
SS =0030 0000000000000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0030 0000000000000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0030 0000000000000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0030 0000000000000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 0000000000000000 0000ffff 00008200 DPL=0 LDT
TR =0000 0000000000000000 0000ffff 00008b00 DPL=0 TSS64-busy
GDT=     000000003fbee698 00000047
IDT=     000000003f306018 00000fff
CR0=80010033 CR2=0000000000000000 CR3=000000003fc01000 CR4=00000668
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=0000000000000000
DR6=00000000ffff0ff0 DR7=0000000000000400
EFER=0000000000000500
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
XMM08=00000000000000000000000000000000 XMM09=00000000000000000000000000000000
XMM10=00000000000000000000000000000000 XMM11=00000000000000000000000000000000
XMM12=00000000000000000000000000000000 XMM13=00000000000000000000000000000000
XMM14=00000000000000000000000000000000 XMM15=00000000000000000000000000000000
(qemu)
```

3-2. KosmOS "memory map & first kernel with monitor"

BootLoader에서 Kernel을 불러온 후, Kernel이 모니터에 픽셀을 출력하는 부분은 성공적으로 구현이 완료되었습니다. 아래는 BootLoader의 Main.c에 대한 코드 리뷰입니다. 우선, Memory Map을 받을 구조체 와 받아올 함수입니다.

```c
struct MemoryMap {
  UINTN buffer_size;
  VOID* buffer;
  UINTN map_size;
  UINTN map_key;
  UINTN descriptor_size;
  UINT32 descriptor_version;
};
```

```c
EFI_STATUS GetMemoryMap(struct MemoryMap* map) {
  if (map->buffer == NULL) {
    return EFI_BUFFER_TOO_SMALL;
  }

  map->map_size = map->buffer_size;
  return gBS->GetMemoryMap(
      &map->map_size,
      (EFI_MEMORY_DESCRIPTOR*)map->buffer,
      &map->map_key,
      &map->descriptor_size,
      &map->descriptor_version);
}
```

이후로, UEFI API 문서를 참고하여 구현한 읽어온 Memory Map을 따로 저장하는 함수 입니다.

```c
EFI_STATUS SaveMemoryMap(struct MemoryMap* map, EFI_FILE_PROTOCOL* file) {
  CHAR8 buf[256];
  UINTN len;

  CHAR8* header =
    "Index, Type, Type(name), PhysicalStart, NumberOfPages, Attribute\n";
  len = AsciiStrLen(header);
  file->Write(file, &len, header);

  Print(L"map->buffer = %08lx, map->map_size = %08lx\n",
      map->buffer, map->map_size);

  EFI_PHYSICAL_ADDRESS iter;
  int i;
  for (iter = (EFI_PHYSICAL_ADDRESS)map->buffer, i = 0;
       iter < (EFI_PHYSICAL_ADDRESS)map->buffer + map->map_size;
       iter += map->descriptor_size, i++) {
    EFI_MEMORY_DESCRIPTOR* desc = (EFI_MEMORY_DESCRIPTOR*)iter;
    len = AsciiSPrint(
        buf, sizeof(buf),
        "%u, %x, %-ls, %08lx, %lx, %lx\n",
        i, desc->Type, GetMemoryTypeUnicode(desc->Type),
        desc->PhysicalStart, desc->NumberOfPages,
        desc->Attribute & 0xfffffflu);
    file->Write(file, &len, buf);
  }
  return EFI_SUCCESS;
}
```

이외에도 이미지가 포함된 볼륨의 루트 디렉토리를 여는 OpenRootDir 함수,
그래픽 출력 프로토콜을 구성하는 OpenGOP 함수, 픽셀 포맷의 문자열을 반환하는 GetPixelFromUnicode 함수 또한 구현을 마쳤습니다. 아래에 순서대로 첨부 했습니다.

```
92  EFI_STATUS OpenRootDir(EFI_HANDLE image_handle, EFI_FILE_PROTOCOL** root) {
93    EFI_LOADED_IMAGE_PROTOCOL* loaded_image;
94    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL* fs;
95
96    gBS->OpenProtocol(
97        image_handle,
98        &gEfiLoadedImageProtocolGuid,
99        (VOID**)&loaded_image,
100       image_handle,
101       NULL,
102       EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL);
103
104   gBS->OpenProtocol(
105       loaded_image->DeviceHandle,
106       &gEfiSimpleFileSystemProtocolGuid,
107       (VOID**)&fs,
108       image_handle,
109       NULL,
110       EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL);
111
112   fs->OpenVolume(fs, root);
113
114   return EFI_SUCCESS;
115 }
```

```
119 EFI_STATUS OpenGOP(EFI_HANDLE image_handle,
120                    EFI_GRAPHICS_OUTPUT_PROTOCOL** gop) {
121   UINTN num_gop_handles = 0;
122   EFI_HANDLE* gop_handles = NULL;
123   gBS->LocateHandleBuffer(
124       ByProtocol,
125       &gEfiGraphicsOutputProtocolGuid,
126       NULL,
127       &num_gop_handles,
128       &gop_handles);
129
130   gBS->OpenProtocol(
131       gop_handles[0],
132       &gEfiGraphicsOutputProtocolGuid,
133       (VOID**)gop,
134       image_handle,
135       NULL,
136       EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL);
137
138   FreePool(gop_handles);
139
140   return EFI_SUCCESS;
141 }
```

```
143 const CHAR16* GetPixelFormatUnicode(EFI_GRAPHICS_PIXEL_FORMAT fmt) {
144   switch (fmt) {
145     case PixelRedGreenBlueReserved8BitPerColor:
146       return L"PixelRedGreenBlueReserved8BitPerColor";
147     case PixelBlueGreenRedReserved8BitPerColor:
148       return L"PixelBlueGreenRedReserved8BitPerColor";
149     case PixelBitMask:
150       return L"PixelBitMask";
151     case PixelBltOnly:
152       return L"PixelBltOnly";
153     case PixelFormatMax:
154       return L"PixelFormatMax";
155     default:
156       return L"InvalidPixelFormat";
157   }
158 }
```

BootLoader의 main 함수는 UefiMain이라는 이름으로 image_handle과 system_table을 parameter로 하여 구성되어 있습니다. 처음에 Page 단위로 메모리 맵을 가져올 버퍼를 준비합니다. 해당 버퍼를 활용해 앞서 정의한 GetMemoryMap 함수를 호출해 메모리 맵을 가져옴, OpenRootDir를 호출해 루트 디렉토리를 엽니다. 해당 루트 디렉토리에 SaveMemoryMap 함수를 활용해 메모리 맵을 저장한 후에 닫고, OpenGOP함수를 호출해 출력 프로토콜을 준비합니다. 미리 입력한 디스플레이 정보인 해상도, 픽셀 형식, 프레임 버퍼를 출력하고, 프레임 버퍼를 흰색으로 초기화 합니다. 이후에는 루트 디렉토리에 있는 Kernel.elf를 읽기 모드로 열어 해당 파일의 크기만큼 메모리를 할당합니다. 할당된 메모리에 Kernel 파일을 읽어 들이고, 최신 메모리 맵을 저장한 후에 Boot Loader를 종료합니다. 이때 Kernel의 entry point를 앞서 miniOS에서 사용해 본대로 계산하여 호출합니다. Boot Loader가 작동을 했다면 Kernel이 실행되는 환경이 되어야 할 것 입니다. 아래는 위의 전 과정과 주석을 담은 코드입니다.

```
161 EFI_STATUS EFIAPI UefiMain( EFI_HANDLE image_handle, EFI_SYSTEM_TABLE *system_table) {
162     Print(L"Hello, KosmOS!\n");
163 
164     CHAR8 memmap_buf[4096 * 4];
165     struct MemoryMap memmap = {sizeof(memmap_buf), memmap_buf, 0, 0, 0, 0};
166     GetMemoryMap(&memmap);
167 
168     EFI_FILE_PROTOCOL* root_dir;
169     OpenRootDir(image_handle, &root_dir);
170 
171     EFI_FILE_PROTOCOL* memmap_file;
172     root_dir->Open(
173       root_dir, &memmap_file, L"\\memmap",
174       EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE | EFI_FILE_MODE_CREATE, 0);
175 
176     SaveMemoryMap(&memmap, memmap_file);
177     memmap_file->Close(memmap_file);
```

```
181     EFI_GRAPHICS_OUTPUT_PROTOCOL* gop;
182 
183     OpenGOP(image_handle, &gop);
184     Print(L"Resoultion: %ux%u, Pixel Format: %s, %u pixels/line\n",
185         gop->Mode->Info->HorizontalResolution,
186         gop->Mode->Info->VerticalResolution,
187         GetPixelFormatUnicode(gop->Mode->Info->PixelFormat),
188         gop->Mode->Info->PixelsPerScanLine);
189     Print(L"Frame Buffer: 0x%0lx - 0x%0lx, Size: %lu bytes\n",
190         gop->Mode->FrameBufferBase,
191         gop->Mode->FrameBufferBase + gop->Mode->FrameBufferSize,
192         gop->Mode->FrameBufferSize);
193 
194     UINT8* frame_buffer = (UINT8*)gop->Mode->FrameBufferBase;
195     for (UINTN i=0; i<gop->Mode->FrameBufferSize; i++){
196         frame_buffer[i]=255; // 8비트 전체가 1 0xff -> 흰색
197     }
```

```
202     // kernel.elf를 읽기 전형으로 열기
203     EFI_FILE_PROTOCOL* kernel_file;
204     root_dir->Open(
205         root_dir, &kernel_file, L"\\kernel.elf",
206         EFI_FILE_MODE_READ, 0);
207 
208     // 미리 메모리 잡아 놓기
209     /* 왜 더 크게 잡는가 (?) -> 구조체에 FileName 부분 배열 요소 개수 정의 X,
210     so, FileName 부분 크기 미리 잡을 필요 있음*/
211     UINTN file_info_size = sizeof(EFI_FILE_INFO) + sizeof(CHAR16) * 12;
212     UINT8 file_info_buffer[file_info_size];
213     // 파일 크기 계산 후, 생성한 메모리에 담기
214     kernel_file->GetInfo(
215         kernel_file, &gEfiFileInfoGuid,
216         &file_info_size, file_info_buffer);
217     // 타입 캐스팅
218     EFI_FILE_INFO* file_info = (EFI_FILE_INFO*)file_info_buffer;
219     UINTN kernel_file_size = file_info->FileSize;
```

```
    /* 이제 커널의 크기를 알았기 때문에, 메모리 확보
    1. 메모리 확보 방법
        1-1. 어디라도 좋으니, 비어있는 공간에서 확보: AllocateAnyPages
        1-2. 지정한 어드레스 이후에 비어있는 공간에서 확보: AllocateMaxAddress
        1-3. 지정한 주소에서 확보: AllocateAddress
        커널은 0x100000번지에 배치를 전제로 만들었다. -> ld.lld 옵션
        so, 다른 위치에 배치시키면 정상 작동 X

    2. 확보할 메모리 영역 유형
    3. 크기
    4. 확보한 메모리 공간의 어드레스를 저장하기 위한 변수 지정
        포인터 넘기는 이유: AllocateAnyPages, AllocateMaxAddress의 경우 값 저장 따로 필요
        but now I don't need it -> 변숫값이 변뀌는 경우가 없기 때문
        return 값이 성공 실패인 상황에서 다른 파라미터 전달: 포인터로 구현 */

    /* 페이지 수 = (kernel_file_size + 0xfff) / 0x1000
    0xfff 더해주는 이유: 페이지수를 올림 하기 위해 -> 잘려서 누락되는 것 방지 */
    EFI_PHYSICAL_ADDRESS kernel_base_addr = 0x100000;
    gBS->AllocatePages(
        AllocateAddress, EfiLoaderData,
        (kernel_file_size + 0xfff) / 0x1000, &kernel_base_addr);

    // 공간 확보 완료 했으니, 파일 읽어 들이기
    kernel_file -> Read(kernel_file, &kernel_file_size, (VOID*)kernel_base_addr);
    Print(L"Kernel: 0x%0lx (%lu bytes)\n", kernel_base_addr, kernel_file_size);
```

```
    // 커널 구동 시키는 일만 남았다 (?)
    // 아닙니다. UEFI BIOS의 부트 서비스 중지 필요 (OS에 방해됨)

    /* ExitBootServices()는 최신 메모리 맵의 맵 키 요구
    지정된 맵 키가 최신의 메모리 맵과 연결된 맵 키가 아닌 경우 실행 실패

    실패시 다시 메모리 맵을 얻고 해당 맵 키를 사용해 재실행
    초기 메모리 맵 취득 -> 함수 호출 사이 여러 기능 사용 -> 첫 시도 반드시 실패
    두번째 실행 -> 실패 시 중대한 오류 이므로 무한루프 처리로 정지 (?)
    */
    EFI_STATUS status;
    status = gBS->ExitBootServices(image_handle, memmap.map_key);
    if (EFI_ERROR(status)){
        status = GetMemoryMap(&memmap);
        if (EFI_ERROR(status)){
            Print(L"failed to get memmory map: %r\n", status);
            while(1);
        }
        status = gBS-> ExitBootServices(image_handle, memmap.map_key);
        if (EFI_ERROR(status)) {
            Print(L"Could not exit boot service: %r\n", status);
            while(1);
        }
    }
```

```
    /* 커널 가동하는 부분
    엔트리 포인트가 놓인 위치를 계산해서 엔트리 포인트 호출
    엔트리 포인트: C언어 에서는 main()
    현재 엔트리 포인트: KernelMain()
    So, KernelMain()이 있는지 특정, 호출하는 것이 부트로더의 큰 목적

    EFL 형식의 사양에 따르면, 64비트용 ELF 엔트리 포인트 어드레스 위치:
    오프셋 24 바이트 위치에서 8파이트 정수로 작성 */
    UINT64 entry_addr = *(UINT64*)(kernel_base_addr+24);

    // 포인터를 함수로 정의 하기 위한 준비
    // Parameter, return 모두 void인 함수

    // typedef void EntryPointType(void);
    // typedef void EntryPointType(UINT64, UINT64);
    typedef void __attribute__((sysv_abi)) EntryPointType(UINT64, UINT64);
    EntryPointType* entry_point = (EntryPointType*)entry_addr;

    // So, 함수로 호출 가능
    entry_point(gop->Mode->FrameBufferBase, gop->Mode->FrameBufferSize);
    Print(L"All done\n");

    while (1);
    return EFI_SUCCESS;
}
```

아래는 Boot Loader에서 로드하는 Kernel의 main.cpp 파일입니다.

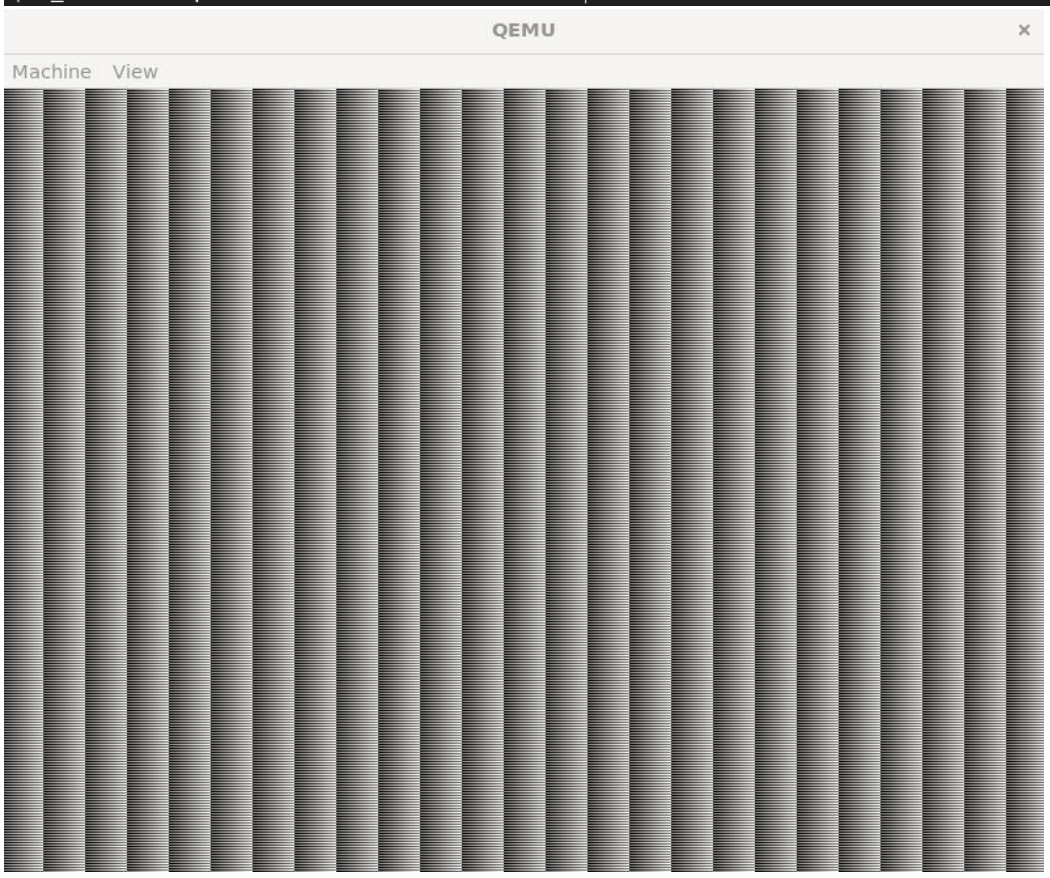
```cpp
#include <cstdint>
// 부트로더에서 픽셀을 그리는 것은 이미 성공
// so, 커널에서 똑같이 그리기 -> 부트로더에서 커널로 렌더링 정보 넘기기

// 파라미터로 2개의 64 비트 정수 추가
extern "C" void KernelMain(uint64_t frame_buffer_base,
                           uint64_t frame_buffer_size){
    // C++ 스러운 타입 캐스팅 // C 언어 (uint8_t*) frame_buffer_base와 유사
    // "정수와 포인터의 전환이기에, 신중해야 한다"를 쉽게 알 수 있다.
    uint8_t* frame_buffer = reinterpret_cast<uint8_t*>(frame_buffer_base);
    for(uint64_t i=0; i<frame_buffer_size; ++i){
        frame_buffer[i] = i%256;
    }
    while (1) __asm__("hlt");
}
```

Boot Loader에서 모니터와 관련된 정보를 입력 받아 출력하고 cpu의 구동을 멈추도록 현재 구현이 되어 있는 상태입니다. 앞서 작성한 BootLoader의 main.c, Kernel의 main.cpp를 EDKII 디렉토리에 넣고, BUILD를 하여 ELF 파일을 두개 얻을 수 있었습니다.

```
~/Altair/kosmos/edk2  ls                                                    06:01:11 PM
ArmPkg              CryptoPkg           FatPkg              Maintainers.txt     pip-requirements.txt  StandaloneMmPkg
ArmPlatformPkg      disk.img            FmpDevicePkg        MdeModulePkg        ReadMe.rst            UefiCpuPkg
ArmVirtPkg          DynamicTablesPkg    IntelFsp2Pkg        MdePkg              RedfishPkg            UefiPayloadPkg
BaseTools           edksetup.bat        IntelFsp2WrapperPkg mnt                 SecurityPkg           UnitTestFrameworkPkg
Build               edksetup.sh         KosmosLoaderPkg     NetworkPkg          ShellPkg
Conf                EmbeddedPkg         License-History.txt OvmfPkg             SignedCapsulePkg
CONTRIBUTING.md     EmulatorPkg         License.txt         PcAtChipsetPkg      SourceLevelDebugPkg
```

현재 까지 구현한 Loader와 초기 Kernel을 qemu에서 실행한 결과입니다. 아래와 같이 의도된 대로 출력이 모니터에 잘 되고 있는 것을 확인할 수 있습니다.

```
~/A/OSLog  run-qemu $OS_HOME/edk2/Build/KosmosLoaderX64/DEBUG_CLANG38/X64/Loader.efi
$OS_HOME/workspace/mikanos/kernel/kernel.elf
```



3-3. KosmOS "print & mouse & interrupt"

최초로 monitor에 0~255의 값을 찍어본 이후에, 집중한 부분은 동작 과정에서 최적화 할 수 있는 부분을 최적화하고, 1개의 device를 사용해 보는 것에 집중해보았습니다. 선정한 device는 mouse 이며, 최종 제출 전까지 interrupt를 활용한 mouse의 동작 및 문자열 출력을 목표로 프로젝트를 진행했습니다. **이하 부분은 설명과 code가 거의 1:1로 대응하여 text로 구현한 내용을 작성했습니다.** 첫번째 단계는 monitor에 의도된 대로 렌더링 및 문자열을 쓰는 과정입니다. 이를 위해서 frame_buffer를 위한 구조체를 사용했습니다. 초기에 부트로더가 OS에 필요한 렌더링 정보를 전달하며, UEFI의 GOP에서 취득한 정보를 Kernel Main의 파라미터에 전달해 주었습니다. 또한 Write Pixel 함수를 사용하여 화면을 그리는 것을 kernel의 main.cpp에 구현했습니다. 현재 까지 과정을 진행한 결과 원하는 위치에 직사각형 출력을 구현할 수 있었습니다.

두번째 단계에서는 C++의 기능을 활용하여 첫번째 단계의 동작 과정을 효율적으로 실행할 수 있도록 수정했습니다. 앞선 단계에서 픽셀 데이터 형식에 대한 판단과 분기문을 실행하는 것이 해당 함수를 호출할 때 마다 호출하는 문제가 있어서 해당 문제를 해결하기 위해 "픽셀 렌더링 인터페이스"와 "픽셀 데이터 형식에 따라 실제 렌더링" 되는 부분을 override를 사용하여 분리하였습니다. 또한 Class를 사용하기 위해서 생성자를 사용했어야 했는데, 현재 구현 단계 에서는 **memory layout에 대한 구현이 존재하지 않아서 일반적인 new를 사용할 수 없었습니다. 이에, 메모리 영역을 확보하지 않는 생성자인 "displacement new"라는 생성자를 따로 operator 제작을 통해 정의 했고, 해당 함수를 사용해 memory layout이 없는 환경에서 class의 형태를 활용한 solution을 코드에 적용시킬 수 있었습니다**. 해당 과정에서 loader가 보다 기존보다 큰 영역을 바라봐야 했습니다. 그 이유는 기존의 loader는 loader section에서 rodata, text section 만을 바라보면 되었지만, class 및 모니터 정보 활용을 위해서 .data, .bss section을 불러올 필요가 있어서 elf 정보를 다룰 수 있는 구조체를 header로 작성하여 loader를 개량했습니다.

세번째 단계에서는 문자열 출력에 집중했습니다. 이미지 렌더링과 다르게 별도의 폰트 렌더링 함수를 정의하여 사용했습니다. 이때, 렌더링 기법이 달라져도 사용 가능하도록 인터페이스와 구현을 분리하여 구현했습니다. 또한 .cpp 와 header들을 분할하여 컴파일을 진행하도록 main.cpp에 있던 함수들을 분리하여 각 파일에 존재하도록 분리했습니다. 폰트 사용을 위해서 python script를 이용해 각각 크기에 맞게 pixel로 찍혀있는 font.txt를 가공하여 사용하였습니다. 또한 sprintf 외부 라이브러리를 통해 monitor에 출력을 하는 것을 구현해 보았고, kernel의 내부 변수를 출력할 수 있는 printk 함수를 구현하여 디버깅 및 커널 내부 변화를 편하게 파악 할 수 있도록 구현해 보았습니다. 해당 내용 까지 구현한 화면은 아래와 같습니다.

문자열에 대한 출력을 완성한 후, OS와 상호작용하기 위해 Device를 한대 연결하는 것을 구현해 보았습니다. 키보드와 마우스라는 선택지가 있었는데, OS강의를 수강하면서 보다 폭넓은 구현을 하기 위해서는 마우스가 더 매력적인 선택지여서 마우스 사용을 목표로 구현을 진행했습니다. font와 비슷하게 마우스 또한 pixel과 대응하도록 ascii값을 알맞게 찍어 사용했습니다. 마우스 커서를 렌더링 하는 것을 사실 text를 작성하는 것과 크게 다르지 않았습니다. 이에 저희가 집중한 부분은 USB 호스트 드라이버를 원활한 사용입니다. 호스트 드라이버를 만드는 방법은 여러가지가 존재하지만, 저희는 자료 조사를 통해 1. 컨트롤러를 제어하는 컨트롤러 드라이버, 2. USB 규격으로 결정된 API를 제공하는 USB 버스 드라이버 3. USB 타킷의 종류별로 제공하는 클래스 드라이버 이렇게 총 3가지 계층으로 구성된 드라이버를 구성했습니다. 사실 디테일한 부분을 포함해서 드라이버를 구성하기에는 드라이버 하나 자체로도 내용이 방대 하기 때문에, 이미 구현된 usb부분을 참고하여 사용했습니다.

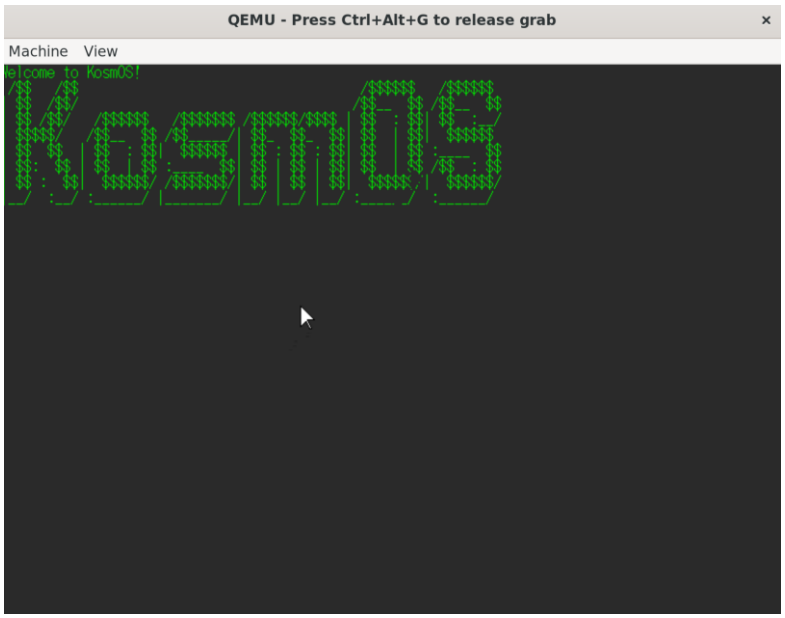
위의 정보로 저희가 구성한 시나리오는 다음과 같습니다. 1. PCI 버스에 연결 된 Device 중에, 2. xHC를 찾고, 초기화 하며, 3. USB 버스상에서 마우스를 찾아서 초기화 한 후, 4. 마우스로 부터 데이터를 수신한다. 이를 위해 **"PCI Configuration Space"**를 분석하여 필요한 정보를 얻어오는 것을 조사하였습니다. 우선 PCI Configuration Space를 읽어 오기 위해 **"CONFIG_ADDRESS" 레지스터, "CONFIG_DATA" 레지스터**를 사용해야 했습니다. 해당 정보들은 각각 IO의 **"0x0cf8", "0x0cfc"**에 32비트 레지스터로 구성되어 있어 이를 **"pci.cpp"** 파일에 정의하여 사용했습니다. IO 포트를 읽고 쓰는 **"asmfunc.asm"** 파일을 작성하여 활용했습니다. PCI 디바이스를 등록하는 배열을 글로별 변수로 정의하는 **"pci.hpp"**를 작성하였고, pci.cpp에 버스에 연결된 디바이스를 재귀적으로 탐색하는 코드를 추가해 주었습니다. 뿐만 아니라 지정된 버스상의 디바이스를 탐색하고, 해당 함수로 지정된 디바이스의 function을 탐색하는 함수를 추가로 작성했습니다. 지정된 해당 function을 조사하고, 발견한 PCI 디바이스를 devices 배열에 추가하는 함수 또한 구현했습니다. 이러한 과정을 통해 QEMU에서 마우스의 사용이 가능하도록 세팅하는 과정을 마무리 되었습니다.

xHCI Spec에서 xHC를 제어하는 레지스터 MMIO의 address는 PCI configuration space에 있는 "BAR0" 레지스터에 작성되어 있어 해당 값을 읽어와서 사용했습니다. 해당 레지스터는 32bit 였기에, 2개의 BAR를 사용하도록 하여 64bit 주소를 설정할 수 있었습니다. 이 외에 **"SwitchEhci2Xhci()", "XUSB2PR"**과 같은 특정 함수들을 직접 구현하기에 난이도가 있어 잘 정리 되어있는 자료를 구글링하여 사용했습니다. 마우스의 이벤트를 능동적으로 조사하는 "Polling" 기법을 활용하는 것은 효율적인 OS에는 적합하지 않습니다. 마우스를 선택한 이유가 사실 바로 이부분 이었습니다. 저희 조에서는 **Polling이 아닌 Interrupt기반의 처리를 구현해 보기 위해** 마우스라는 주제를 선택하게 되었습니다.

일정 시간마다 특정 처리를 수행하는 것은 인터럽트를 사용하지 않으면 매우 어렵습니다. 이에, xv6에서 page fault handler를 활용해 COW를 구현했듯이, interrupt handler를 구현 및 IDT에 등록하여 이벤트가 발생하면 하드웨어가 CPU에 신호를 보내고, CPU가 handler를 호출하고, handler가 작업을 마치면 중단한 작업을 처리하게끔 구현을 진행했습니다.

xHCI Spec에 맞는 인터럽트 핸들러 또한, official document의 용례를 보고 구현했으며, 해당 과정은 USB 호스트 드라이버에 쌓인 이벤트가 처리하게 끔 지시하고, 처리해야 하는 이벤트 도착시 인터럽트 발생과 핸들러에서 해당 처리를 드라이버에 지시하는 흐름으로 구성되어 있습니다. 인터럽트의 구현을 위해서는 interrupt descriptor에 대해 자세히 조사 및 학습하였고, **SetIDTEntry**라는 함수를 통해 parameter로 받은 값을 지정한 인터럽트 디스크립터에 설정을 기록하도록 구현했습니다. IDT의 설정을 한 차례 마무리 한 후에는 CPU에게 해당 위치를 알려주었습니다. MSI 인터럽트 또한 구현하여 인터럽트 핸들러가 실행하는 거에 맞춰, 인터럽트를 발생 시키는 로직까지 구현을 진행했습니다. 현재 시점까지는 사실 저희가 모든 기능을 구현했다고 보기는 어렵습니다. 그러나, OS 및 하드웨어 관련 강의 내용을 바탕으로 다양한 문서를 이해하는 것에 의의를 두고 학습을 위해 진행했습니다. 하여 앞선 인터럽트 관련 구현 내용을 정리 하면 아래와 같습니다.

인터럽트를 다루기 위해, 인터럽트 핸들러, 인터럽트 디스크립터, 인터럽트 발생지에 대한 설정을 완료 했고, 핸들러를 통해 End Of Interrupt 레지스터에 값을 작성했습니다. 인터럽트 디스크립터는 IDT라는 배열 구조로 구성되며, 인터럽트 핸들러의 주소와 속성을 담고 있다는 사실에 맞춰 구현했습니다. 이후 IDT의 시작 주소 및 크기를 "lidt"명령어로 CPU에게 등록하고, xHCI Spec상 MSI 방식으로 인터럽트를 발생 시키기 위해 Message address 및 Message Data 레지스터를 설정하는 과정을 구현했습니다. 힘겹게 인터럽트를 구현했으나, 인터럽트 핸들러의 처리에 시간이 오래 걸렸습니다. 이에 인터럽트 처리 과정 중 Queue 자료구조를 적용시켜 보다 처리를 가볍게 수정하여 마우스 사용까지 최종 구현을 완성 할 수 있었습니다.



3-4 Conclusion of KosmOS:

Qemu, UEFI API와 EDKII를 사용해서 Bottom-up 방식의 OS를 구현 해볼 수 있었습니다. BootLoader가 Kernel을 불러오는 과정과 이때 모니터에 대한 값을 파라미터로 넘겨 Kernel이 모니터에 출력 하는 부분을 구현한 상태에서, interrupt를 구현하여 마우스의 사용이 가능하도록 구현을 성공적으로 구현했습니다.

4. xv6 (https://github.com/AltairKosmoTale/xv6)

4-1. xv6 개요

Top-down으로 OS를 구현하기 위해 교육용으로 구현이 잘 되어있는 xv6를 miniOS에서 얻어낸 결론에 의거하여 채택하게 되었습니다. xv6에서는 어떤 프로젝트를 진행 할 수 있을까 고민하다가, 강의 시간에 배운 내용을 최대한 구현 할 수 있는 환경이 xv6라고 판단해서, 최대한 강의 진도에 맞춰서 개발을 하는 것을 목표로 설정했습니다. 이에 xv6를 구성하고 있는 여러가지 코드를 분석하였고 해당 코드가 있는 이유와 메커니즘을 하나씩 분석해 보았습니다.

처음으로 xv6에서 처음 수행 한 것은 **User Program**이 잘 구동하는 지 확인해 보는 일 이었습니다. 해당 작업은 간단한 C 프로그램을 작성해서, build 목록에 포함만 해주면 되는 간단한 일 이었습니다.

두번째로 구현한 것은 **System Call** 입니다. xv6에 새로운 system call을 생성하기 위해서 fork와 exec이 어떻게 구현되고 호출 되는지 확인해 보았습니다. 두 system call의 정상 작동을 위해서는 "defs.h"에 선언을 해줘야 하며, "syscall.c"에 외부 선언, "syscall.h"에 system call 분류 code를 위한 define pre-processor, "users.h"에 선언 추가, "usys.S"에 SYSCALL("system_call") 작성이 필요하다는 것을 확인 하였고, 이를 바탕으로 system call이 xv6위에서 어떻게 작동하는지 알 수 있었습니다. 이러한 사실을 바탕으로 일반적인 user program function call에서 "학번"을 출력하는 program과 system call로 "학번"을 출력하는 프로그램을 작성하고 테스트 해봤습니다.

4-2. xv6 "Round-Robin based Priority Scheduler"

세번째로 구현한 것은 **Round-Robin 기반 Priority Scheduler**입니다. Priority Scheduler를 구현하기 위해서는 기존에 작동하고 있던 Scheduler를 분석할 필요가 있었습니다. xv6의 Scheduler는 각 CPU가 실행 가능한 프로세스를 선택하고 실행한 후 제어를 스케쥴러로 반환하는 무한 루프로 구현된 상태였습니다. 기본 동작 방식으로는 스케쥴러가 ptable을 순회하며 Round-Robin 스케쥴링을 수행하는 것을 확인 할 수 있었습니다. 이 과정에서 swtch를 통해 커널 스택과 레지스터가 저장 및 복원 되고, swtch는 실행 중인 프로세스가 스케쥴러로 전환 하도록 스케쥴링된 호출을 할 때 반환한다는 것을 알게 되었습니다. trap.c에서 yield를 통해 해당 프로세스 CPU 자원을 해제하고, Context Switching이 가능하다는 것을 확인했습니다. 해당 정보들을 확인한 후, 본격적으로 Scheduler를 구성하기 시작했습니다.

우선, 각 프로세스에 priority 개념을 추가했습니다. "proc.h" 파일의 "proc" 구조체에 int priority 멤버 변수를 추가했습니다. 또한 새로운 Process가 생성 되었을 때의 값 또한 설정해 줘야 하기에, userinit(void) 함수와 함수가 호출하는 allocproc()를 수정해 주었습니다.

```
38  struct proc {
39    uint sz;
40    pde_t* pgdir;
41    char *kstack;
42    enum procstate state;
43    int pid;
44    struct proc *parent;
45    struct trapframe *tf;
46    struct context *context;
47    void *chan;
48    int killed;
49    struct file *ofile[NOFILE];
50    struct inode *cwd;
51    char name[16];
52    int priority;
53    int count;
54    int age;
55  };
```

```
64  static struct proc*
65  allocproc(void)
66  {
67    struct proc *p;
68    char *sp;
69    acquire(&ptable.lock);
70    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
71      if(p->state == UNUSED)
72        goto found;
73    release(&ptable.lock);
74    return 0;
75  found:
76    p->state = EMBRYO;
77    p->pid = nextpid++;
78    p->priority = 5;
79    p->count = 0;
```

allocproc에서 ptable 중 비어있는 프로세스를 할당할 때 priority를 임의의 값 5로 지정해 주었습니다. 또한 fork를 통해서 process가 생성될 때는 임의로 부모와 같은 priority를 갖도록 설정했습니다.

```
223     np->priority = curproc->priority;
```

priority의 getter & setter의 경우 system call로 구현 하였습니다. setter에서는 process id와 priority를 arguments로 하여 해당 id를 가진 process의 priority의 값을 수정할 수 있도록 구현했고, getter의 경우, 해당 id를 가진 process의 priority 값을 출력하도록 구현 했습니다. 아래는 getter, setter에 해당하는 system call 코드 입니다.

```
163  int
164  set_proc_priority(int pid, int priority)
165  {
166    struct proc *p;
167    acquire(&ptable.lock);
168    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
169      if(p->pid == pid) {
170        p->priority = priority;
171        release(&ptable.lock);
172        return 1;
173      }
174    }
175    release(&ptable.lock);
176    return -1;
177  }
```

```
179  int
180  get_proc_priority(int pid)
181  {
182    struct proc *p;
183    int priority = -1;
184    acquire(&ptable.lock);
185    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
186      if(p->pid == pid) {
187        priority = p->priority;
188        break;
189      }
190    }
191    release(&ptable.lock);
192    return priority;
193  }
```

우선 getter를 test 해보았습니다. 각각 test code와 출력 결과 입니다.

```
13  void get_proc_priority_test(void)
14  {
15      printf(1, "Test1\n");
16      for(int i=0; i<10; i++){
17          int priority = get_proc_priority(i);
18          if(priority != -1){
19              printf(1, "[*] pid : %d\n", i);
20              printf(1, "[*] priority : %d\n", priority);
21              sleep(1);
22          }
23      }
24  }
```

```
$ proctest
Test1
[*] pid : 0
[*] priority : 0
[*] pid : 1
[*] priority : 7
[*] pid : 2
[*] priority : 9
[*] pid : 3
[*] priority : 1
```

priority가 잘 출력 되는 것을 확인해 볼 수 있었습니다.

fork된 자식 process가 부모와 같은 priority를 잘 출력하는지 test 해보았습니다.

아래와 같이 부모 process의 priority와 자식 process의 priority가 같은 것을 알 수 있습니다.

```
27  void fork_test(void)
28  {
29      printf(1, "Test2\n");
30      int pid = fork();
31      if(pid == 0){
32          int priority = get_proc_priority(getpid());
33          printf(1, "[*] c_pid : %d\n", getpid());
34          printf(1, "[*] c_priority : %d\n", priority);
35          exit();
36      }
37      else if(pid > 0){
38          wait();
39          int priority = get_proc_priority(getpid());
40          printf(1, "[*] p_pid : %d\n", getpid());
41          printf(1, "[*] p_priority : %d\n", priority);
42      }
43  }
```

```
Test2
[*] c_pid : 4
[*] c_priority : 4
[*] p_pid : 3
[*] p_priority : 4
```

fork된 상태 후, child process의 priority 값을 setter로 수정한 경우도 결과가 잘 나온 것을 확인할 수 있었습니다.

```
46  void fork_setter_test(void)
47  {
48      printf(1, "Test3\n");
49      set_proc_priority(getpid(), 3);
50      int pid = fork();
51      if(pid == 0){
52          int priority = get_proc_priority(getpid());
53          printf(1, "[*] c_pid : %d\n", getpid());
54          printf(1, "[*] c_priority : %d\n", priority);
55          exit();
56      }
57      else if(pid > 0){
58          int priority = get_proc_priority(getpid());
59          printf(1, "[*] p_pid : %d\n", getpid());
60          printf(1, "[*] p_priority : %d\n", priority);
61          wait();
62      }
63  }
```

```
Test3
[*] p_pid : 3
[*] p_priority : 3
[*] c_pid : 5
[*] c_priority : 3
```

스케쥴러는 아래와 같이 구현을 진행했습니다. starvation이 발생할 수 있는 부분에 대해서 aging 로직을 추가한 것이 핵심입니다. p->count가 100을 넘게 되면, 해당 프로세스의 priority를 증가시키고, p->priority가 10 (가장 낮음)이 되면 다시 1(가장 높음)으로 초기화 합니다. 또한 선택된 process의 count를 +10 해줌을 통해 전환 로직을 구현 하였습니다.

```
478  void scheduler(void) {
479    struct proc *p;
480    struct cpu *c = mycpu();
481    c->proc = 0;
482    for(;;) {
483      sti();
484      acquire(&ptable.lock);
485      int highest_priority = 0xff;
486      struct proc *selected_proc = 0;
487      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
488        if(p->state == RUNNABLE) {
489          p->count++;
490          if (p->count >= 100) {
491            p->priority++;
492            if(p->priority == 10)
493              p->priority = 1;
494            p->count = 0;
495          }
496        }
497      }
498      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
499        if(p->state != RUNNABLE)
500          continue;
501        if(p->priority < highest_priority) {
502          highest_priority = p->priority;
503          selected_proc = p;
504        }
505      }
506      if (selected_proc) {
507        selected_proc->count += 10;
508        c->proc = selected_proc;
509        switchuvm(selected_proc);
510        selected_proc->state = RUNNING;
511        swtch(&(c->scheduler), selected_proc->context);
512        switchkvm();
513        c->proc = 0;
514      }
515      release(&ptable.lock);
516    }
517  }
```

해당 코드의 test를 위해, proc dump를 사용하여 값을 확인해 보면서 무한 루프로 돌아가는 test process가 count와 priority를 바꿔 가며 잘 작동 하는 것을 확인해 볼 수 있었습니다.

```
pid     priority        state   name
1       8               sleep   init 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
2       2               sleep   sh 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
3       7               sleep   proctest 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
6       1               run     loop
pid     priority        state   name
7       5               run     loop
pid     priority        state   name
8       9               runble  loop
```

```
pid     priority        state   name
1       8               sleep   init 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
2       2               sleep   sh 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
3       7               sleep   proctest 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
6       2               run     loop
pid     priority        state   name
7       1               run     loop
pid     priority        state   name
8       9               runble  loop
```

```
pid     priority        state   name
1       8               sleep   init 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
2       2               sleep   sh 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
3       7               sleep   proctest 80104157 80104205 801050ad 80106201 80105f4c
pid     priority        state   name
6       4               run     loop
pid     priority        state   name
7       5               runble  loop
pid     priority        state   name
8       1               run     loop
```

Round-Robin Scheduler의 구현을 위해 사용한 모든 파일은 아래와 같습니다.

- defs.h : set_proc_priority, get_proc_priority
- syscall.c : sys_set_proc_priority, sys_get_proc_priority 추가
- syscall.h : set_proc_priority, get_proc_priority에 대한 시스템콜 번호 선언
- sysproc.c : sys_set_proc_priority, sys_get_proc_priority에 대한 wrapper 구현
- user.h : set_proc_priority, get_proc_priority 선언 추가
- usys.S : set_proc_priority, get_proc_priority SYSCALL 선언
- proc.c : procdump 함수 수정, scheduler: aging logic 추가,
  set_proc_priority, get_proc_priority 함수 구현
- proc.h : proc구조체에 priority, count 멤버변수 추가
- loop.c : priority, aging test를 위한 while loop
- Makefile : proctest.c, loop.c 추가

4-3. xv6 "Copy on Write"

네번째로 구현한 것은 **Copy on Write**입니다. Copy on Write (이하 COW)는 데이터를 Write할 때, 이전의 내용을 Copy한다는 뜻을 가집니다. 즉, Write를 하지 않으면 Copy를 진행하지 않습니다. Process에서 Fork하게 되면, 기존 구현에서는 부모 프로세스의 모든 virtual memory를 Copy 해서 자식 프로세스에게 할당하는 것을 xv6동작 분석을 통해 알아냈습니다. 부모나 자식 프로세스가 COW 메모리 영역에 접근하게 되면 그 영역을 먼저 복사하고 작성하게 됩니다.

COW 구현을 위해서는 알아야할 배경지식이 있어 조사 및 정리 해보았습니다. x86 아키텍쳐의 **CR3** 레지스터는 top-level 페이지 디렉토리를 가리키고 있습니다. 또한 page table의 entry의 일부는 **TLB**에 cache되어 있습니다. 이에, 메인 메모리에 위치한 page table에 어떠한 변화가 생기면 TLB에 cache 되어 있던 page table의 변화가 생겼다는 것을 알려줘야 합니다. 이를 위해 xv6에서 제공되는 **"lcr3"**함수를 호출해야 한다는 것을 알게 되었습니다. "lcr3" 함수는 TLB를 Flush 하는 함수입니다. "lcr3(V2P(p->pgdir));" 다음과 같이 사용되며, 해당 함수는 switchuvm 함수에서도 호출됩니다. 또한 context switching시 매번 호출 됩니다.

COW는 총 4가지 단계에 걸쳐 구현을 완성했습니다. 각각의 과정은 **1단계: 빈 페이지의 숫자를 반환하는 시스템콜 구현, 2단계: 각 메모리 페이지들을 대상으로 reference counter 적용, 3단계: vm.c의 copyuvm 수정, 4단계: page fault handler 구현** 이렇게 총 4가지 단계로 구성되어 있습니다.

우선 getNumFreePages 시스템 콜을 추가해 주었습니다. kalloc.c에 선언이 되어있고, system call 추가를 위해 syscall.c, syscall.h, usys.S에 내용을 추가해 주었으며, wrapper 함수를 sysproc.c에 추가해주었습니다.

syscall.c:

```
108  extern int sys_getNumFreePages(void);
```

syscall.h:

```
25  #define SYS_getNumFreePages 24
```

usys.S:

```
34  SYSCALL(getNumFreePages)
```

sysproc.c:

```
114  int
115  sys_getNumFreePages(void)
116  {
117    return GetNumFreePages();
118  }
```

kalloc.c:

```
41  uint GetNumFreePages() {
42    return num_free_pages;
43  }
```

해당 파일에 "uint num_free_page;"를 통해 전역 변수를 선언 했습니다.

```
12  uint num_free_pages;
```

초기화 하는 함수인 kinit1에서 선언한 전역 변수를 0으로 초기화 하는 로직을 추가했습니다.

```
50  void
51  kinit1(void *vstart, void *vend)
52  {
53    initlock(&kmem.lock, "kmem");
54    kmem.use_lock = 0;
55    num_free_pages = 0;
56    freerange(vstart, vend);
57  }
```

또한 allocation을 진행하는 부분에서는 증가시켜야 하고, free 할 부분에서는 감소 시켜야 하기에 각각 "kalloc"함수에서 "kfree" 함수에서 --, ++ 로직을 추가했습니다.

```
111  char*
112  kalloc(void)
113  {
114    struct run *r;
115
116    if(kmem.use_lock)
117      acquire(&kmem.lock);
118
119    num_free_pages--;
```

```
96    if(get_refcount(V2P(v)) == 0) {
97      memset(v, 1, PGSIZE);
98      num_free_pages++;
```

1단계에서는 free page를 반환하는 system call 추가와 free page 변수를 사용할 수 있도록 설정이 완료되었습니다.

2단계에서는 각 메모리 페이지들을 대상으로 reference counter를 적용 시켰습니다. 우선 kalloc.c 파일에 "uint pgrefcount[PHYSTOP >> PGSHIFT];"를 통해 전역 변수를 선언했습니다. "#define PGSHIFT 12"를 선언 했으며, "PHYSTOP >> PGSHIFT"를 통해서 총 페이지 수를 계산합니다. PHYSTOP은 memlayout.h에 정의 되어 있으며, PGSHIFT는 mmu.h에 정의되어 있는 것을 확인했습니다.

```
13  uint pgrefcount[PHYSTOP >> PGSHIFT];
```

이후에는 page의 reference counter 값을 읽어오는 함수 "get_refcount", page의 reference counter 값을 증가시키는 함수 "inc_refcount", page의 reference counter 값을 감소 시키는 "dec_refcount" 함수를 구현했습니다.

```
29  uint get_refcount(uint pa) {
30    return pgrefcount[pa >> PGSHIFT];
31  }
32
33  void inc_refcount(uint pa) {
34    pgrefcount[pa >> PGSHIFT]++;
35  }
36
37  void dec_refcount(uint pa) {
38    pgrefcount[pa >> PGSHIFT]--;
39  }
40
41  uint GetNumFreePages() {
42    return num_free_pages;
43  }
```

이후에는 차례로, freerange 함수에서 pgrefcount 배열을 0으로 초기화 하고, kfree 함수에서 dec_refcount 함수를 호출하도록, copyuvm 함수에서는 inc_refcount 함수를 호출 하도록, kalloc 함수에서는 refcount를 1로 설정하도록 로직을 각각 추가해 주었습니다.

kalloc.c_freerange():

```
66  void
67  freerange(void *vstart, void *vend)
68  {
69    char *p;
70    p = (char*)PGROUNDUP((uint)vstart);
71    for(; p + PGSIZE <= (char*)vend; p += PGSIZE){
72      pgrefcount[V2P(p) >> PGSHIFT] = 0;
73      kfree(p);
74    }
75  }
```

kalloc.c_kree();

```
92    if(get_refcount(V2P(v)) > 0) {
93      dec_refcount(V2P(v));
94    }
```

kalloc.c_kalloc.c

```
122    if(r){
123      kmem.freelist = r->next;
124      pgrefcount[V2P((char*)r) >> PGSHIFT] = 1;
125    }
```

vm.c_copyuvm():

```
356      inc_refcount(pa);
357    }
358    lcr3(V2P(pgdir)); // flush TLB
359    return d;
```

2단계에서는 reference counter를 적용하고, 관련된 함수가 적절한 시점에 호출 되도록 구현을 완료했습니다.

3단계에서는 vm.c의 copyuvm 함수의 로직을 수정했습니다. xv6의 fork 함수에서 copyuvm 함수가 호출됩니다. 기존의 copyuvm 함수는, 부모 프로세스의 페이지 테이블을 자식 프로세스에게 "복사"하는 함수입니다. 엄연히 "Copy-on-Write"와는 다른 함수입니다. copyuvm 함수를 호출해서 새로운 process가 생성될 때 메모리 페이지들을 생성하고, 실제로 복사하는 것이 아니라, 부모 process의 메모리 페이지들을 사용하도록 변경하고 Write 권한을 없애야 합니다. 또한 이전에 정의 했던 inc_refcount 함수를 호출하여 해당 메모리 페이지들의 reference counter를 증가시켜야 합니다. 언급된 순서에 따라 먼저 copyuvm 시 원래 VP에 매핑이 되도록 수정했습니다. 기존에는 kalloc()해서 memmove로 복사 후, mappages를 V2P(mem)을 해줬다면, 부모 PA를 그대로 인자로 넣어줬습니다.

```
344    // if((mem = kalloc()) == 0)
345      // goto bad;
346    // memmove(mem, (char*)P2V(pa), PGSIZE);
347    // if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
348    //   kfree(mem);
349    //   goto bad;
350    // }
351    // 부모 페이지 테이블을 공유하도록 수정
352    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
353      goto bad;
354    }
```

fork 함수에서는 copyuvm 함수를 호출하도록 수정했습니다.

proc.c_fork():

```
204  int
205  fork(void)
206  {
207    int i, pid;
208    struct proc *np;
209    struct proc *curproc = myproc();
210
211    if((np = allocproc()) == 0){
212      return -1;
213    }
214
215    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
216      kfree(np->kstack);
217      np->kstack = 0;
218      np->state = UNUSED;
219      return -1;
220    }
```

Write 권한을 없애기 위해서 mmu.h에 선언된 PTE_W 부분 비트를 0으로 만들어서 권한을 해제하였고, 이 시점에 inc_refcount 함수를 호출해 해당 메모리 페이지들의 reference counter를 증가시켰습니다.

vm.c_copyuvm():

```
339    *pte &= ~PTE_W; // page table entry의 PTE_W을 0비트로 -> read only
340
341    pa = PTE_ADDR(*pte);
342    flags = PTE_FLAGS(*pte); // page table entry flag
```

```
352      // 부모 페이지 테이블을 공유하도록 수정
353      if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
354        goto bad;
355      }
356      inc_refcount(pa);
```

3단계에서는 COW의 동작 방식에 맞게 동작하도록 copyuvm 수정을 완료했습니다.

4단계 에서는 page fault handler를 구현했습니다. trap.c 파일의 trap 함수를 수정하여 page fault 발생 시, 핸들러를 호출하도록 수정했습니다. 이미 traps.h 파일에 T_PGFLT 라는 page fault 핸들러 번호가 존재하여, trap.c에 있는 handler siwtch-case 부분에 page fault에 대한 case를 추가했습니다.

traps.h:

```
18  #define T_PGFLT         14      // page fault
```

trap.c_trap():

```
80    case T_PGFLT:
81      pagefault();
82      break;
```

vm.c 파일에 pagefault() 함수를 구현했습니다. 해당 함수는 여러 가지 기능을 하는데, x86에서 page fault address를 저장하는 register인 CR2를 읽는 "rcr2()" 함수를 호출하여 page fault가 발생한 VA를 읽어 들이는 기능, 해당 VA의 page table entry 확인 후 PA를 찾는 기능, PA를 이용하여 해당 메모리 페이지의 reference counter 값을 확인하고, 해당 값이 보다 크면 새로운 페이지를 할당 받아서 복사하도록, 1인 경우에는 현재 PTE의 write 권한만 추가했습니다.

```
416  void pagefault(void){
417    uint va_fault = rcr2();
418    if(va_fault < 0){
419      return;
420    }
421    pte_t *pte_fault = walkpgdir(myproc()->pgdir, (void *)va_fault, 0);
422    uint pa = PTE_ADDR(*pte_fault);
423    if(pa == 0){
424      return;
425    }
426
427    int ref_count = get_refcount(pa);
428
429    if(ref_count > 1){
430      char *mem = kalloc();
431      if(mem == 0){
432        panic("pagefault: kalloc failed");
433      }
434      memset(mem, 0, PGSIZE);
435      memmove(mem, (char*)P2V(pa), PGSIZE);
436      *pte_fault = V2P(mem) | PTE_FLAGS(*pte_fault);
437      dec_refcount(pa);
438    }else if(ref_count == 1){
439      *pte_fault |= PTE_W;
440    }else{
441      panic("pagefault: ref_count < 1");
442    }
443  }
```

4단계 에서는 page fault handler 구현을 완료했습니다.

여러 단계에 걸쳐 구현한 CoW를 Test 하기 위해 Test Program을 작성했습니다.

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int global = 0;
6
7  void test1(){
8      printf(1, "[Test 1]\n");
9      printf(1, "Before sbrk() getNumFreePages() = %d\n", getNumFreePages());
10     sbrk(4096);
11     printf(1, "After sbrk() getNumFreePages() = %d\n", getNumFreePages());
12 }
13
14 void test2(){
15     printf(1, "[Test 2]\n");
16     printf(1, "Before fork() getNumFreePages() = %d\n", getNumFreePages());
17     int pid = fork();
18     if(pid == 0){ // Child
19         sleep(10);
20         printf(1, "[-] Child: global = %d, getNumFreePage() = %d\n", global, getNumFreePages());
21         global = 1;
22         printf(1, "[+] Child: global = %d, getNumFreePage() = %d\n", global, getNumFreePages());
23     }else{ // Parent
24         printf(1, "[-] Parent: global = %d, getNumFreePage() = %d\n", global, getNumFreePages());
25         wait();
26         printf(1, "[+] Parent: global = %d, getNumFreePage() = %d\n", global, getNumFreePages());
27     }
28 }
```

test1의 경우, sbrk 전후로 Page 수를 비교하는 프로그램입니다.
test2는 fork 전후로 자식 및 부모 프로세스 에서의 페이지 수 변화를 관찰하는 코드입니다.
전역 변수를 child에서 수정 한 이후에 free page수가 어떻게 변하는지, 프로그램이 끝나기 전에 free page 수가 어떻게 변하는지 살펴보았습니다. 아래는 실행 결과입니다.

```
$ ./cow
[Test 1]
Before sbrk() getNumFreePages() = 56732
After sbrk() getNumFreePages() = 56731
[Test 2]
Before fork() getNumFreePages() = 56731
[-] Parent: global = 0, getNumFreePage() = 56663
[-] Child: global = 0, getNumFreePage() = 56663
[+] Child: global = 1, getNumFreePage() = 56662
[+] Parent: global = 0, getNumFreePage() = 56731
$
```

Test1의 결과로, sbrk(4096)시, free page의 개수가 1 줄어드는 것을 확인할 수 있었습니다.
Test2의 결과를 보면, fork 전에 후로 page가 프로그램에 비례하여 줄어드는 것을 확인해 볼 수 있고, 전역 변수 수정 전 까지는 page의 수가 변경 되지 않습니다. 이는 read만 하고 있는 시점에서는 원본 페이지를 유지하기 때문입니다. 그러나 child에서 전역 변수를 수정한 순간 새로운 page가 할당 되어 free page 수가 1 줄어드는 것을 확인해 볼 수 있습니다. 또한 프로그램이 종료되는 시점에 자식 프로세스가 종료되고, free page 수가 복원되었다는 것을 확인해 볼 수 있습니다. 구현한 COW가 복제된 페이지를 효율적으로 관리하고 있는 것을 알 수 있습니다.
Copy-on-Write의 구현을 위해 사용한 모든 파일은 아래와 같습니다.

- defs.h : get_refcount, inc_refcount, dec_refcount, GetNumFreePages, pagefault
- kalloc.c : num_free_pages, pgrefcount, get_refcount, inc_refcount, dec_refcount, GetNumFreePages, freerange, kfree
- mmu.h : PGSHIFT 선언
- syscall.c : sys_getNumFreePages systemcall 추가
- syscall.h : SYS_getNumFreePages systemcall 번호 선언
- trap.c : pagefault handler 추가
- user.h : getNumFreePages 추가
- usys.S : getNumFreePages SYSCALL 추가
- vm.c : copyuvm PTE_W 비트 수정, 부모 페이지 테이블 공유 하도록 수정, lcr3 추가, pagefault
- cow.c : Copy-On-Write 테스트 코드 작성

4-4. Conclusion of xv6

Qemu를 사용하여 구현된 교육용 OS 구현체 xv6 위에서 user program와 system call을 구현해 보았고, Process 관련해서는 강의 시간에 배운 Scheduler, Process 개념, Aging기법을 적절히 활용하여 Starvation이 없는 Priority Scheduler를 성공적으로 구현 했습니다. Memory 관련해서는 기존의 복사 기반의 xv6 fork 및 copyuvm 함수를 "Copy on Wirte" mechanism에 맞춰 수정하였고, page fault hanlder를 구현함으로 COW를 성공적으로 구현 했습니다.

5. 프로젝트 결론 및 피드백

프로젝트 동안 **miniOS**의 Test Scenario, Round Robin based Priority Scheduler 및 Copy on Write가 적용된 **xv6**, Boot Loader를 통해 초기 커널을 메모리에 올려 실행하고, 해당 커널이 모니터 정보를 활용하여 출력 하며, polling이 아닌 interrupt 기반 마우스의 활용이 가능한 from scractch OS인 **KosmOS**를 완성했습니다. 각각의 프로젝트 대신 한개의 더 완성도 높은 프로젝트를 진행할 수 있었지만, 3개의 프로젝트를 진행하면서 오히려 더 많은 것을 배울 수 있었습니다. xv6를 구현할 때는 강의 시간에 배운 Hardware 관련 부분을 직접 설정하지는 못하지만 lcr3와 같은 함수 처럼 "레지스터를 읽는다"를 함수적으로 호출하고 그 결과가 의도된 대로 잘 사용되는 것에서 Hardware적인 관점을 고려하며 Software적인 로직에 집중할 수 있는 느낌을 받았습니다. xv6에서 page fault handler를 구현체로 구현했다면, KosmOS에선 interrupt handler를 진짜 Hardware 단에서 구현하는 느낌을 받았습니다. 또한 KosmOS에서는 메모리 layout이 구현되어 있지 않아 "displacement new"와 주어진 환경을 기반으로 프로그래밍을 진행하는 문제를 해결하면서 각각의 단계에서 풀어야 하는 다양한 문제를 해결하며 자신감이 붙었고, 추후에는 xv6에서 만들어놓은 로직을 KosmOS위에 올려 두 프로젝트가 합쳐진 결과물 제작을 기대해 볼 수 있을 것 같습니다.