2024-1 운영체제 프로젝트 최종 보고서

컨텍스트 스위칭(Context Switching) 및 스케줄링(Scheduling)구현

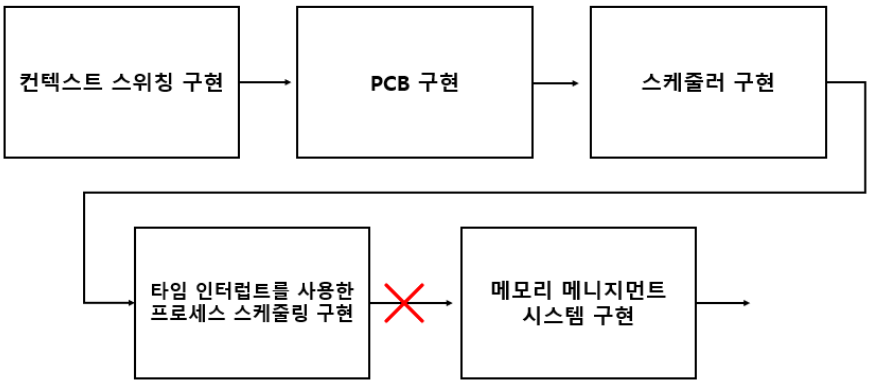
5 조

20180543 강구현 20192650 한수호

20213592 한승규 20192651 홍준기

프로젝트 요약

컨텍스트 스위치와 스케줄링은 여러 프로세스가 동시에 실행될 수 있도록 하는 운영체제의 핵심 기능이다. 이러한 기능은 멀티태스킹 환경에서 필수적이며, 운영체제의 기본이라고 할 수 있다. 본 프로젝트에서는 현대 컴퓨터의 가장 일반적인 CPU 인 x86 64 아키텍처를 기반으로 리눅스 커널 모듈에서 컨텍스트 스위치 및 스케줄러를 시뮬레이션 한다. 타임 인터럽트를 통한 컨텍스트 스위칭은 개발환경의 특성상 구현하지 못하였지만, 컨텍스트 스위칭, 스케줄러, 프로세스 구조 등 프로세스를 다루는 기본적인 요소들은 모두 구현 성공하였다.




```
[ 1163.664567] Module initialized
[ 1163.664571] Initial context for thread1: RIP=ffffffffc0966050, RSP=ffff9ab307e5c000
[ 1163.664573] Initial context for thread2: RIP=ffffffffc09660d0, RSP=ffff9ab30685a000
[ 1163.664574] Allocated stack1: ffff9ab307e5b000 (aligned: ffff9ab307e5c000)
[ 1163.664576] Allocated stack2: ffff9ab306859000 (aligned: ffff9ab30685a000)
[ 1163.765636] interrupt
[ 1163.765783] sched
[ 1163.854493] interrupt
[ 1163.854651] sched
[ 1163.854740] Thread 1 is running (1)
[ 1163.874673] Thread 1 is running (2)
[ 1163.894637] Thread 1 is running (3)
[ 1163.914576] Thread 1 is running (4)
[ 1163.934535] Thread 1 is running (5)
[ 1163.954493] Thread 1 is running (6)
[ 1163.974457] Thread 1 is running (7)
[ 1163.994413] Thread 1 is running (8)
[ 1164.014370] Thread 1 is running (9)
```

프로젝트의 최종 결과물이다. 인터럽트가 발생하여 스케줄링을 통해 Thread 2 로 전환되어야 하지만 그렇지 않고 있다. 타임 인터럽트를 통한 프로세스 스케줄링 구현에 실패한 이유에 대해서는 본 보고서의 실패 이유 분석 부분에서 자세하게 다룰 예정이다.

1. 서 론

프로세스 스케줄링은 운영체제가 여러 프로세스에 CPU 의 자원을 정해진 알고리즘에 맞추어 효율적으로 할당하는 작업이다. 이를 통해 운영체는 여러 작업을 동시에 수행하는 것처럼 보이게 한다. 현대의 PC 는 실제로 여러 개의 코어를 통해 멀티 코어 기반의 프로세스 처리를 하지만 본 프로젝트에서는 싱글 코어를 기준으로 구현하였다. 본 프로젝트의 초기 목표는 프로세스 멀티태스킹 과정의 흐름을 별도의 라이브러리 없이 처음부터 끝까지 모두 구현하는 거였지만 기술적, 능력적 한계에 부딪힐 때마다 팀원 들과의 소통을 통해 계획을 수정 보완하였다. 초기 계획은 간단한 형태의 PCB, 컨텍스트 스위치, 라운드 로빈 스케줄러, 스케줄러를 실행할 타임 인터럽트 핸들러, interrupt service routine, memory management system 을 팀원 들과의 협력으로 구현하려고 하였지만 여러 한계에 부딪혀 계획을 수정하였다. PCB, 컨텍스트 스위치, 스케줄러를 구현하였고 인터럽트와 메모리는 리눅스 라이브러리를 사용하였다.

2.1 컨텍스트 스위칭

컨텍스트 스위칭은 OS 에서 여러 프로세스 간에 CPU 를 공유하는 방법이다. CPU 가 하나의 작업에서 다른 작업으로 전환할 때 발생한다. 현재 프로세스의 상태를 저장하고 새로운 프로세스의 상태를 복구한다.

```
typedef struct {
    uint64_t r15, r14, r13, r12, r11, r10, r9, r8;
    uint64_t rdi, rsi, rbp, rbx, rdx, rcx, rax;
    uint64_t rip, rsp;
    uint64_t rflags;
} __attribute__((packed)) context_t;
```

위의 코드는 x86 64 아키텍처의 레지스터 값들을 저장하는 구조체이다. 해당 값들을 CPU 레지스터에 직접 load, store 함으로써 컨텍스트 스위칭을 실행한다. __attribute__((packed))를 사용한 이유는 구조체의 멤버변수들이 연속된 메모리에 위치하게 하기 위함이다.

```
__asm__ volatile (
    "movq %%rsp, 0x80(%0)\n\t"
    "pushfq\n\t"
    "popq %%rax\n\t"
    "movq %%rax, 0x88(%0)\n\t"
    "movq %%rbx, 0x58(%0)\n\t"
    "movq %%rcx, 0x68(%0)\n\t"
    "movq %%rdx, 0x60(%0)\n\t"
    "movq %%rsi, 0x48(%0)\n\t"
    "movq %%rdi, 0x40(%0)\n\t"
    "movq %%rbp, 0x50(%0)\n\t"
    "movq %%r8, 0x38(%0)\n\t"
    "movq %%r9, 0x30(%0)\n\t"
    "movq %%r10, 0x28(%0)\n\t"
    "movq %%r11, 0x20(%0)\n\t"
    "movq %%r12, 0x18(%0)\n\t"
    "movq %%r13, 0x10(%0)\n\t"
    "movq %%r14, 0x08(%0)\n\t"
    "movq %%r15, 0x00(%0)\n\t"
    "leaq 1f(%%rip), %%rax\n\t"
    "movq %%rax, 0x78(%0)\n\t"
"movq %1, %%rax\n\t"
"movq 0x88(%%rax), %%rbx\n\t"
"pushq %%rbx\n\t"
"popfq\n\t"
"movq 0x80(%%rax), %%rsp\n\t"
"movq 0x58(%%rax), %%rbx\n\t"
"movq 0x68(%%rax), %%rcx\n\t"
"movq 0x60(%%rax), %%rdx\n\t"
"movq 0x48(%%rax), %%rsi\n\t"
"movq 0x40(%%rax), %%rdi\n\t"
"movq 0x50(%%rax), %%rbp\n\t"
"movq 0x38(%%rax), %%r8\n\t"
"movq 0x30(%%rax), %%r9\n\t"
"movq 0x28(%%rax), %%r10\n\t"
"movq 0x20(%%rax), %%r11\n\t"
"movq 0x18(%%rax), %%r12\n\t"
"movq 0x10(%%rax), %%r13\n\t"
"movq 0x08(%%rax), %%r14\n\t"
"movq 0x00(%%rax), %%r15\n\t"
"sti\n\t"
"jmp *0x78(%%rax)\n\t"
"1:\n\t"
: : "r" (old_ctx), "r" (new_ctx) : "memory", "rax"
```

현재 CPU 의 레지스터를 old_ctx 구조체에 저장하는 어셈블리 코드이다. 앞서 설명했던 것처럼 __attribute__((packed))를 사용하였고 uint64_t 변수를 사용하였기에 구조체의 시작 주소부터 8bit 씩 커질 때 마다 각 멤버변수들이 순서 대로 할당되어 있다. 예) 0x8 번지 r14, 0x78 번지

new_ctx 구조체에 저장되어 있는 값들을 현재 CPU 에 load 하는 부분이다. 두 부분을 합쳐서 switch context 함수로 정의하였다.

```
void thread1(void) {
    int count = 0;
    while (count < 20) {
        printk(KERN_INFO "Thread 1 is running (%d)\n", count + 1);
        msleep(100);
        switch_context(thread1_ctx, thread2_ctx);
        count++;
    }
}

void thread2(void) {
    int count = 0;
    while (count < 20) {
        printk(KERN_INFO "Thread 2 is running (%d)\n", count + 1);
        msleep(100);
        switch_context(thread2_ctx, thread1_ctx);
        count++;
    }
}
```

```
[   50.912996] Module initialized
[   50.912998] Initial context for thread1: RIP=ffffffffc0833170, RSP=ffff920ec9
dd7000
[   50.913000] Initial context for thread2: RIP=ffffffffc08331d0, RSP=ffff920ec9
dd8000
[   50.913001] Allocated stack1: ffff920ec9dd6000 (aligned: ffff920ec9dd7000)
[   50.913003] Allocated stack2: ffff920ec9dd7000 (aligned: ffff920ec9dd8000)
[   50.913004] Loading context:
[   50.913005] Thread 1 is running (1)
[   51.226683] Switching context from ffff920ecd6ed840 to ffff920ecd6ed300
[   51.226689] Thread 2 is running (1)
[   51.478950] Switching context from ffff920ecd6ed300 to ffff920ecd6ed840
[   51.479925] Thread 1 is running (2)
[   51.883412] Switching context from ffff920ecd6ed840 to ffff920ecd6ed300
[   51.883417] Thread 2 is running (2)
[   51.992482] Switching context from ffff920ecd6ed300 to ffff920ecd6ed840
[   51.992486] Thread 1 is running (3)
[   56.164073] Thread 2 is running (18)
[   56.272511] Switching context from ffff920ecd6ed300 to ffff920ecd6ed840
[   56.272516] Thread 1 is running (19)
[   56.384026] Switching context from ffff920ecd6ed840 to ffff920ecd6ed300
[   56.384031] Thread 2 is running (19)
[   56.492690] Switching context from ffff920ecd6ed300 to ffff920ecd6ed840
[   56.492695] Thread 1 is running (20)
[   56.600376] Switching context from ffff920ecd6ed840 to ffff920ecd6ed300
```

main 함수는 생략 하였지만 thread1_ctx, thread2_ctx 구조체를 생성하고 구조체의 멤버변수 rip 과 rsp 값을 초기화 하여 thread1 과 thread2 의 함수를 실행시킨 결과이다. thread1 과 thread2 함수 내부에서 switch_context 함수를 실행하여 thread1 에서 thread2 로 thread2 에서 thread1 로 번갈아 가며 컨텍스트 스위칭을 진행한다. 일반적인 함수 call 과 다르게 앞서 설명한 switch_context 함수를 사용하여 두 함수를 실행시켰다.

리눅스 운영체제의 유저 모드에서는 일반적으로 유저 모드에서는 cpu 의 레지스터 값을 변경할 수 없다. 그렇기 때문에 커널 모드에서 코딩을 진행하여 모듈로 실행한 모습이다. dmesg 명령어를 사용하여 printk 로 출력된 값들을 확인 하였다.

2.2 컨텍스트 스위칭 구현 시 문제점과 해결 방안

기본적으로 리눅스 위에서 구현을 하다보니 하드웨어 자원을 직접 사용하는데 크나큰 제약이 많았다. 그 첫번째가 user 모드에서는 CPU 의 레지스터를 직접 변경 할 수 없다는 것이다.

```
guhyeon@guhyeon-VirtualBox:~/test$ ./main
세그멘테이션 오류 (코어 덤프됨)
```

기본적으로 cpu 레지스터를 직접 변경하려는 시도를 하면 위의 세그멘테이션 오류가 출력되며 코드가 실행 되지 않는다. 이 문제를 해결 방법은 유저모드가 아닌 커널 모드에서 코드를 실행 시키는 것이다. 그러기 위하여 커널 모듈로 코딩을 진행 하였다. 커널 모듈이란 요청시 리눅스 커널에 동적으로 로드될 수 있는 프로그램 코드 조각이다. 기본적으로 커널 모드에서 실행되기 때문에 유저모드에서 할 수 없었던 CPU 레지스터의 직접 수정을 할 수 있다. 번거로운 점은 디버깅, 혹은 결과 출력을 위한 printf 대신 printk 를 사용하여야 한다는 점이다. printk 의 출력은 dmesg 로 확인 가능하다.

```
Box:~/test1$ sudo insmod test.ko
```

make 파일을 통해 생성한 모듈을 로딩하는 리눅스 명령어이다.

```
lBox:~/test1$ sudo dmesg
[  632.386026] Module Loaded!
```

로드된 모듈의 printk 출력값을 sudo dmesg 명령어로 확인한 모습이다.

두번째 문제점은 switch context 함수의 구현이었다. 생소한 x86 아키텍처의 어셈블리 언어를 다루다 보니 chat gpt 로 생성한 코드가 동작하지 않을 때 디버깅에 있어 어려움을 겪었다. 이를 해결하기 위하여 팀 회의를 진행하며 x86 아키텍처의 어셈블리를 공부하는 간단한 스터디를 진행하였다. 이를 통해 어셈블리로 구성된 switch context 함수를 구현하였고 몇번의 디버깅 후 정상적으로 동작함을 확인하였다.

3.1 Task Queue

PCB 를 단순화하여 각 태스크의 상태와 레지스터 상태를 저장하고 관리하는 운영 체제의 핵심 자료구조를 구현했다.

```c
typedef struct task {
    uint8_t PID;
    context_t context;
    struct task* next;
} task;
```

위 코드는 PCB 를 프로세스의 상태 정보를 포함하는 task 구조체로 구현한 코드이다. PID, 레지스터 값들을 저장한 context 구조체가 있고 next 는 다음 태스크를 가리키는 포인터로, 작업 큐에서 링크드 리스트(linked list)를 형성한다.

```c
typedef struct {
    task* head;
    task* tail;
    uint8_t count;
    task* current_t;
} taskQueue;
```

위 Task 구조체와 더불어 Circular Queue 를 구성하는 taskQueue 구조체이다. taskQueue 의 head, tail, queue 에 들어있는 task 의 개수를 저장하는 count, 현재 실행되고 있는 task 를 가리키는 current_t 를 멤버변수로 가지고 있다.

```c
void init_taskQueue(taskQueue* q);
uint8_t isEmpty(taskQueue* q);
void enqueue(taskQueue* q, task* new_task);
void dequeue(taskQueue* q, uint8_t PID);
task* get_next_task(taskQueue* q);
```

task 의 추가, 제거, 및 다음 실행할 task 의 선택을 하여 task queue 를 제어하는 함수들이다. enqueue 함수를 통해 새로운 task 를 큐에 추가하고, dequeue 함수를 통해 특정 PID 를 가진 task 를 큐에서 제거한다. 또한 get_next_task 함수는 다음에 실행할 task 를 계산하는데 사용된다. 스케줄러에서 사용하는 함수이다. 처음

3.2 Task Queue 구현 시 문제점과 해결방안

처음 코딩을 시작할 때, 실행할 작업을 담아두는 자료구조를 어떻게 구성해야 할지 몰라 막막했습니다. 스케줄러와 상호작용하면서 각 작업의 상태와 정보를 저장하고 관리하는 방법을 제로 베이스에서 생각해내기에는 어려움이 많았습니다. 몇 차례 시행착오를 겪은 후, 어려움을 극복하기 위해 리눅스 코드를 분석하기로 결정했습니다. 또한 xv6 의 코드와 GitHub 에 올라와

있는 minios 프로젝트들의 코드를 참고하였고, 결국 순환 큐로 구성하기로 결정한 후로는 순조롭게 진행할 수 있었습니다. 원하는 작업을 종료하기 위해 설계한 dequeue 함수는 기존 큐의 정의와는 다르게 설계되어 엄밀히 따지면 큐라고 할 수 없지만, 본 프로젝트의 목적에 맞게 정상적으로 동작한다.

4.1 스케줄러

Round Robin (RR) 스케줄링은 CPU 스케줄링 알고리즘 중 하나로, 각 프로세스가 일정한 시간 동안 CPU 를 사용할 수 있도록 하는 선점형 스케줄링 기법이다. RR 스케줄링의 목표는 모든 프로세스에 공정하게 CPU 시간을 할당하여, 시스템의 응답 시간을 개선하는 것이다. 각 프로세스는 고정된 시간 단위(Time Quantum) 동안 실행되며, 시간이 만료되면 다음 프로세스로 전환된다.

RR 스케줄링은 모든 프로세스가 동일한 시간의 CPU 사용량을 보장하여 공정성을 유지한다. 또한 짧은 시간 단위로 프로세스 스위칭이 이루어지기 때문에, 사용자와 시스템 간의 상호작용이 중요한 환경에서 유리하다. x86_64 아키텍처는 타이머 인터럽트와 효율적인 멀티태스킹 지원으로, RR 스케줄링의 빈번한 컨텍스트 스위칭 요구를 충족시키기에 적합하여 본 프로젝트의 리눅스 커널 모듈 구현에 적합하였다. Round Robin 스케줄링은 sched 함수, get_next_task 함수, 그리고 SWITCH_CONTEXT 매크로 3 가지로 구성 되어있다.

```
void sched(taskQueue* q) {
    asm volatile (        "sti\n\t");
    printk(KERN_INFO "sched\n");
    sched_counter++;
    if (sched_counter >= 1) {
        sched_counter = 0;
        task* next_task = get_next_task(q);
```

함수 정의 및 초기설정부분에서 작업 큐를 받아서 스케줄링을 진행한다. asm volatile ("sti \ n \ t") 인라인 어셈블리 코드로, cpu 인터럽트를 활성화하고 prink 를 통해서 커널 로그에 스케줄러가 호출되었음을 출력한다.

다른 작업 선택 시 task*next_task=get_next_task(q)를 통해 다음 작업을 가져온다. 다음 작업 및 현재 작업이 유효한지 확인하고 컨텍스트 스위칭을 준비하고 실행한다. 현재 작업이 없다면 단순 current_task 를 next_task 로 설정하여 스위칭을 완료한다.

```
if (next_task != NULL) {
    if (current_task != NULL) {
        task* old_task = current_task;
        current_task = next_task;
        SWITCH_CONTEXT(&(old_task->context), &(next_task->context));
    } else {
        current_task = next_task;
    }
}
```

4.2 스케줄러 구현 시 문제점과 해결방안

스케줄러 구현 시 가장 큰 문제는 다음에 실행할 task 를 어떻게 효율적으로 불러올 것인가였다. 기존의 task queue 구조체는 이러한 기능을 완벽하게 지원하지 못했다. 이를 해결하기 위해 task queue 구조체를 수정하여 현재 실행 중인 task 를 저장하는 포인터 변수 current_t 를 추가하고, 다음에 실행할 task 를 불러오는 함수인 get_next_task 를 구현하였다.

기존의 task queue 구조체는 단순히 task 들을 순차적으로 저장하는 방식이다. 그러나 스케줄링의 효율을 높이기 위해 다음에 실행할 task 를 빠르게 선택할 수 있는 메커니즘이 필요했다. 이에 따라 task queue 구조체에 current_t 변수를 추가하여 현재 실행중인 task 를 저장하고, get_next_task 함수를 통해 queue 에서 다음 task 를 선택하도록 했다. 해당 함수는 현재 task 가 끝나면 끝나면 다음 task 로 Pointer 를 이동시키고, 마지막 task 에 도달하면 처음 task 로 돌아간다.

5. 타임 인터럽트

```
void timer_callback(struct timer_list *t) {

        printk(KERN_INFO "interrupt\n");
        mod_timer(&switch_timer[0], jiffies + msecs_to_jiffies(100));
        sched(q);

}
```

10ms 마다 timer_callback 함수가 실행되어 shced 함수를 호출한다. 요약에서 말했듯이 정상적으로 동작하지 않으며 그 이유는 후술한다. 원래는 ISR 부터 코딩하고자 하였지만 이를 구현하기 위해서는 리눅스의 인터럽트 파트를 실제 코드 부분부터 이

해하여야 했기 때문에(하드웨어를 다루는 부분은 운영체제가 담당하고 있기 때문에) 라이브러리에서 지원하는 소프트웨어 타이머를 사용하여 구현하였다.

6. main 함수

엄밀히 따지면 module init 함수라 해야 하지만 편의를 위해 main 함수라고 칭한다.

```
q = kmalloc(sizeof(taskQueue), GFP_KERNEL);
init_taskQueue(q);

task* t1 = (task*)kmalloc(sizeof(task), GFP_KERNEL);
t1->PID = 1; 0)
task* t2 = (task*)kmalloc(sizeof(task), GFP_KERNEL);
t2->PID = 2; 0)

stack1 = kmalloc(4096, GFP_KERNEL);
stack2 = kmalloc(4096, GFP_KERNEL);
thread1_ctx = kmalloc(sizeof(context_t), GFP_KERNEL);
thread2_ctx = kmalloc(sizeof(context_t), GFP_KERNEL);

thread1_ctx->rsp = (uint64_t)(stack1 + 4096); 1)
thread1_ctx->rip = (uint64_t)thread1; 2)
thread1_ctx->rflags = 0x00000200; 3)

thread2_ctx->rsp = (uint64_t)(stack2 + 4096); 1)
thread2_ctx->rip = (uint64_t)thread2; 2)
thread2_ctx->rflags = 0x00000200; 3)

t1->context=*thread1_ctx;
t2->context=*thread2_ctx;

enqueue(q, t1);
enqueue(q, t2); 4)
```

taskQueue 인 q 와 thread_ctx1, thread_ctx2 는 전역 변수로 선언되어 있다. 해당 코드는 프로세스의 초기화 과정이라고 볼 수 있다. PID 를 설정하고 0), 각 thread 에 stack 을 할당하고 1), 시작 명령어 주소를 할당하고 2), 초기 flag 값을 할당한다 3). queue 에 집어넣어 실행, 스케줄링 될 준비를 한다 4)

```
void thread1(void) {
    int count = 0;
    while (count < 40) {
        printk(KERN_INFO "Thread 1 is running (%d)\n", count + 1);
    udelay(10000);
    udelay(10000);
        count++;
    }
    dequeue(q, 1);
    while(1){
    udelay(10000);
    udelay(10000);

    printk(KERN_INFO "Thread 1 is sleep\n" );
    }
}
```

프로세스로 처리될 함수이다 정해진 명령어를 실행 후 dequeue(q,1)을 실행함으로써 스케줄러의 스케줄링 대상에서 제외된다. 프로세스가 종료되는 것이다.

7. 실패 이유 분석

결론부터 말하자면 thread1 을 실행 중일 때 타이머 인터럽트가 발생하지 않는다.

```
[ 1163.664567] Module initialized
[ 1163.664571] Initial context for thread1: RIP=ffffffffc0966050, RSP=ffff9ab307e5c000
[ 1163.664573] Initial context for thread2: RIP=ffffffffc09660d0, RSP=ffff9ab30685a000
[ 1163.664574] Allocated stack1: ffff9ab307e5b000 (aligned: ffff9ab307e5c000)
[ 1163.664576] Allocated stack2: ffff9ab306859000 (aligned: ffff9ab30685a000)
[ 1163.765636] interrupt
[ 1163.765783] sched
[ 1163.854493] interrupt
[ 1163.854651] sched
[ 1163.854740] Thread 1 is running (1)
[ 1163.874673] Thread 1 is running (2)
[ 1163.894637] Thread 1 is running (3)
[ 1163.914576] Thread 1 is running (4)
[ 1163.934535] Thread 1 is running (5)
[ 1163.954493] Thread 1 is running (6)
[ 1163.974457] Thread 1 is running (7)
[ 1163.994413] Thread 1 is running (8)
[ 1164.014370] Thread 1 is running (9)
```

첫번째 인터럽트를 통해 스케줄러는 다음 실행할 프로세스를 정하고 두번째 인터럽트에서 Thread1 를 cpu 에 로드하여 Thread1 을 실행한다. 그후로는 인터럽트가 발생하지 않는다.

```
[  408.211987] Thread 1 is running (1)
[  408.231830] Thread 1 is running (2)
[  408.251674] Thread 1 is running (3)
[  408.271358] interrupt
[  408.271520] Thread 1 is running (4)
[  408.291362] Thread 1 is running (5)
[  408.311207] Thread 1 is running (6)
[  408.331085] Thread 1 is running (7)
[  408.331935] interrupt
[  408.350929] Thread 1 is running (8)
[  408.370773] Thread 1 is running (9)
[  408.390616] Thread 1 is running (10)
[  408.391386] interrupt
[  408.410460] Thread 1 is running (11)
```

인터럽트가 정상적으로 발생한다면 이러한 결과를 보였을 것이다. Thread1 이 실행 중일 때 인터럽트가 발생하지 않는 이유를 밝히려면 리눅스의 인터럽트 시스템의 전반적인 지식을 알아야 하겠지만 지금 수준에서 추측하기를 인터럽트 핸들러 함수가 정상적으로 종료되지 않았기 때문이라고 생각한다. 코드를 보면 알 수 있듯이 인터럽트 핸들러가 실행되는 중에 jmp, ret 어셈블리어를 통하여 강제로 프로그램의 흐름을 변경한다. 이렇게 되면 핸들러가 정상적으로 종료되지 않은 채 다른 명령어를 실행하게 된다. 핸들러가 정상적으로 종료되면서 다음 인터럽트가 발생하도록 모종의 조치를 취한다면 인터럽트가 발생하지 않는 이유를 설명할 수 있을 것이다.

8. 해결 시도 및 결론

이를 해결하기 위하여 jmp 나 ret 으로 다음 실행할 명령어를 강제로 변경하지 않고 stack 의 값을 변경하여 인터럽트 핸들러가 자연스럽게 우리가 원하는 위치로 리턴 하게 하는 방법을 시도해보았다.

```
"movq 0x78(%%rax), %%rax\n\t" \
"movq %%rax, 0x10(%%rsp)\n\t" \
```

수정된 context switch 함수의 일부이다. 0x78 에 저장되어 있는 task 의 명령어 시작주소를 rax 레지스터에 mov 한 후 스택의 첫번째 주소의 값을 rax, 즉 원하는 명령어 위치(thread1 함수)로 변경하는 코드이다. 이렇게 되면 핸들러가 종료되면서 알아서 해당 위치로 return 할 것이고 핸들러는 정상 종료되며 다음 번 인터럽트가 발생할 것이라고 생각하였다. 하지만 결과는 예상과 달랐다. 수정하기 전과 똑같이 thread1 만 실행되고 인터럽트는 발생하지 않았다. 실행이 되지 않으면 않았지 같은 결과가 나온 것이 의문이었고 stack 값을 직접 확인해본 결과 인터럽트 핸들러는 알 수 없는 위치로 리턴 한 후 그 위치에서 어떠한 일을 실행한 뒤 다시 원래 위치로 복귀하는 것을 알아내었다. 이는 리눅스의 구조적인 문제로 이를 해결하려면 인터럽트 서비스 루틴을 처음부터 끝까지 작성하는 것이 있을 수 있겠고 리눅스 위에서가 아닌 마이크로프로세서나 다른 별도의 시뮬레이터를 통해 기존 OS 의 제약을 받지 않는 채로 프로젝트를 진행하는 것일 것이다. 한달 정도의 시간이 더 있었다면 충분히 성공했을 것이라는 것이 팀 내의 의견이다. 프로젝트를 성공적으로 마무리하지 못한 것에 있어서 아쉬움이 남지만 이를 통해 우리 팀은 많은 것을 배웠다. 여기서 끝내지 않고 방학 때나 시간의 여유가 생길 때 문제점을 보완하여 정상 동작 시킬 예정이다.

9. 역할 분담 및 기여도

강구현	컨텍스트 스위칭 구현 및 전체 시스템 합치기 프로젝트 가이드라인 및 발표 ppt 작성	30%
한수호	컨텍스트 스위칭 구현 및 발표 ppt 작성	23%
한승규	task 구조체 구현 및 보고서 작성	23%
홍준기	스케줄러 구현 및 보고서 작성	23%