과제#1

리눅스CPU 스케줄러분석

201610674 이영훈

유저 프로그램 소스 코드 설명

1-1

```c
int main(int argc, char *argv[]) {
    char* temp_cpu;
    char* temp_time;
    int i;
    int cpu, time; // 프로세스 갯수와 소요시간
    pid_t pids[100]; // 프로세스 갯수를 100으로 가정
```

```c
for (i=1; i<=argc; i++) { // Command line arguments 변환
    if(i==1) temp_cpu = argv[i];
    if(i==2) temp_time = argv[i];
}
cpu = atoi(temp_cpu);
time = atoi(temp_time);
```

```c
pid_t pid = 1;
for (i = 0; i<cpu-1; i++) {
    printf("Creating Process: #%d\n", i);
    pids[i] = fork();
    pid = pids[i];
    if (pids[i] < -1 ) return -1;
    else if (pids[i] == 0) {
        calc(time,i);
        break;
    }

}
if(p_num == 0) p_num = cpu-1;
if(pid > 0) calc(time,cpu-1);
wait(NULL);
return 0;
```

메인 함수에서 fork() 부분

프로세스 5개를 돌린다고 가정하면 부모 프로세스 1개와 자식 프로세스 4개를 만들기 위하여 pid 변수와 pids 배열을 사용하였다. pids로 for문안에서 자식을 구별하고 calc함수를 호출했으며 부모 프로세스는 fork가 끝난 뒤 calc함수를 호출하기 위하여 for문 밖에서 pid 변수로 부모 프로세스를 확인하여 calc함수를 호출하였다. 그 이후 wait으로 부모 프로세스가 먼저 끝난 경우 자식 프로세스를 기다리게 하였다.

```c
clock_gettime(CLOCK_MONOTONIC, &begin);
long long save = 0;
while(1) {
    for (i=0; i<ROW; i++) {
        for (j=0; k<COL; j++) {
            for(k=0; k<COL; k++) {
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
            }
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &mid);
    time_diff(&begin,&mid,&end);

    count++;

    save = end.tv_sec * NANOS + end.tv_nsec;

    if(entire_time >= time * NANOS) break;
    if(save >= 100000000) { // 100ms
        begin = mid;
        entire_time += save;
        printf("PROCESS #%02d count = %04ld %04lld ms\n", cpuid, count, save/MICROS);
        save = 0;
    }
```

calc()에서 시간과 연산 수 잼

While문이 계속 돌면서 실제 수행 시간을 더해가며 만약 100ms가 넘게 되면 그 시간을 기준으로 다시 시간을 처음부터 잰다. 그렇게 쌓인 시간은 전체 시간에 더해주어서 정해진 time이 넘어가면 while문 밖으로 나간다.

```
void time_diff(struct timespec *begin, struct timespec *mid, struct timespec *end) {
    if((mid->tv_nsec - begin->tv_nsec) < 0) {
        end -> tv_sec = mid -> tv_sec - begin -> tv_sec - 1;
        end -> tv_nsec = mid -> tv_nsec - begin -> tv_nsec + NANOS;
    }
    else {
        end -> tv_sec = mid -> tv_sec - begin -> tv_sec;
        end -> tv_nsec = mid -> tv_nsec - begin -> tv_nsec;
    }
}
```

monotonic 시간 구하는 함수

추가 수행내용 : signal handler

```
signal(SIGINT, (void *)my_sig_handler); // signal_handler 추가
```

메인 함수에서 선언

```
void my_sig_handler(int signo) {
    printf("HALT !! Process Number : %d Execution time : %lld ms \n",p_num, entire_time/MICROS);
    wait(NULL);
    exit(0);
}
```

시그널 핸들러 함수를 통해 ctrl+c 한 경우 프로세스 번호와 지금까지 수행시간을 출력 후 wait 하고 종료

1-2

```
struct sched_attr attr; // 스케줄링을 위한 구조체
memset(&attr, 0, sizeof(attr));
attr.size = sizeof(struct sched_attr);

attr.sched_policy = SCHED_RR; // 라운드로빈 설정
attr.sched_priority = 95; // 임의의 우선순위

int result = sched_setattr(getpid(), &attr, 0); // attr 구조체에 담아서 sched_setattr 함수에 보낸다.
if (result == -1) perror("Error calling sched_setattr.\n"); // 에러 검출
```

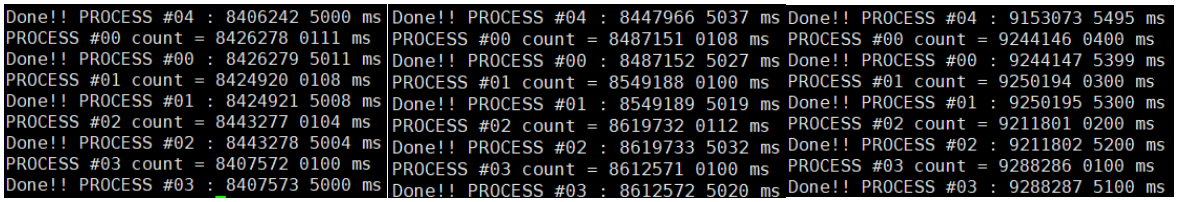
메인 함수에서 sched_attr 구조체를 할당하여 정책과 우선순위를 정하고 sched_setattr 함수 호출

```c
struct sched_attr {
    uint32_t size;
    uint32_t sched_policy;
    uint64_t sched_flags;
    int32_t sched_nice;

    uint32_t sched_priority;

    uint64_t sched_runtime;
    uint64_t sched_deadline;
    uint64_t sched_period;
};
static int sched_setattr(pid_t pid, const struct sched_attr *attr, unsigned int flags) {
    return syscall(SYS_sched_setattr, pid, attr, flags);
}
```

아래 sched_setattr함수에서 받은 매개변수와 함께 시스템콜을 호출한다.

캡쳐화면

Time Slice 1ms 10ms 100ms

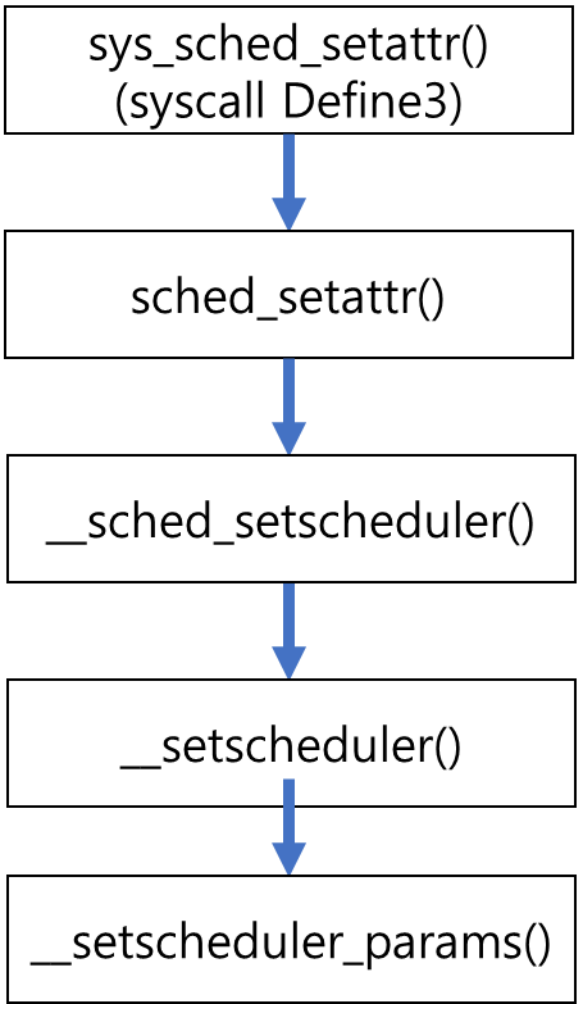


RR Time slice	**1ms**	**10ms**	**100ms**
Calculations per second (total calc / max time)	1,682.783	1,699.487	1,741.809
Baseline=1ms	100.00%	100.99%	103.50%
Baseline=10ms	99.01%	100.00%	102.49%

Time slice가 줄어들수록 연산 횟수가 줄어가는 것을 볼 수 있다. 따라서 Time slice가 적을수록 Context Switch가 자주 일어나서 그 만큼의 오버헤드가 발생하여 연산에 영향을 준다는 것을 알 수 있었다.

1-3

Sys_sched_setattr()함수 콜체인 그림



1) Sched_setattr()과 콜체인 함수들 분석 :

(코드에 주석으로 설명 기록하여 분석함)

include/linux/syscall.h asmlinkage long sys_sched_setattr()

869 lines

```
1.  asmlinkage long sys_sched_setattr(pid_t pid, struct sched_attr __user *attr, unsign
    ed int flags);
```

프로토타입

kernel/sched/core.c

SYSCALL_DEFINE3(sched_setattr, pid_t pid, struct sched_attr __user * uattr, unsigned int flags)

4578 lines

```
1.  SYSCALL_DEFINE3(sched_setattr, pid_t pid, struct sched_attr __user * uattr, unsigned int flags)     // 시스템 호출 핸들러 매크로 3
2.  {
3.      struct sched_attr attr;
4.      struct task_struct *p;
5.      int retval;
6.
7.      if (!uattr || pid < 0 || flags)
8.          return -EINVAL;  // arch/powerpc/boot/stdio.h /* Invalid argument */
9.
10.     retval = sched_copy_attr(uattr, &attr);  //attr 에 uattr 인자로 넘어온 구조체를 복사한다. 정상 수행시 0 리턴
11.     if (retval)
12.         return retval;
13.
14.     if ((int)attr.sched_policy < 0)
15.         return -EINVAL;
16.                     // RCU(Read Copy Update)는 읽는 작업이 주로 이루어지는 자료구조를 보호하기 위한 또 하나의 동기화 기법이다. 다른 동기화 방법과는 다르게 락이나 카운터를 사용하지 않는다.
17.     rcu_read_lock();    //RCU 에서 데이터를 참조하기 위한 Critical Section 을 시작 했다는 것을 통지한다.
18.     retval = -ESRCH;   /* No such process */
19.     p = find_process_by_pid(pid); // pid 를 받아옴
20.     if (p != NULL)
21.         retval = sched_setattr(p, &attr);
22.     rcu_read_unlock();  // RCU 에서 데이터의 참조가 끝난것을 통지한다.
23.
24.     return retval;
25. }
```

kernel/sched/core.c sched_copy_attr() : 유저 영역의 sched_attr를 커널 영역으로 안전하게 복사해오는 함수

4477 lines

```
1.  static int sched_copy_attr(struct sched_attr __user *uattr, struct sched_attr *attr)
2.  {
3.      u32 size;
4.      int ret;
5.
6.      if (!access_ok(VERIFY_WRITE, uattr, SCHED_ATTR_SIZE_VER0)) // 유저 프로세스 공간으로의 안전한 접근이 가능한지 확인하는 함수
7.          return -EFAULT; // /* Bad address */
8.
9.      /* Zero the full structure, so that a short copy will be nice: */
10.     memset(attr, 0, sizeof(*attr));
11.
12.     ret = get_user(size, &uattr->size);  용도 : 유저 영역의 데이터를 커널 영역으로 복사, 인자의 넘어온 변수의 길이만큼 복사한다.
13.
14.     if (ret)
15.         return ret;
16.
```

```
17.     /* Bail out on silly large: */
18.     if (size > PAGE_SIZE) // arch/alpha/include/asm/page.h PAGE_SIZE 13^2 로 고정
19.                 PAGE_SIZE 보다 큰 경우 복사 해온 데이터를 다시 유저 영역으로 넘긴다
20.         goto err_size;  // err_size:
21.                 put_user(sizeof(*attr), &uattr-
    >size); get_user 와 반대로 커널 영역의 데이터를 유저 영역으로 복사 인자의 넘어온 변수의 길
    이만큼 복사
22.                 return -E2BIG; /* Argument list too long */
23.
24.     /* ABI compatibility quirk: */ 호환성문제
25.     if (!size) size 가 0 이면 SCHED 구조체 크기로 정해줌
26.         size = SCHED_ATTR_SIZE_VER0;  48 /* sizeof first published struct */
27.
28.     if (size < SCHED_ATTR_SIZE_VER0)  SCHED 구조체 크기보다 작으면 오류가 나므로 goto

29.         goto err_size;
30.
31.     /*
32.      * If we're handed a bigger struct than we know of,
33.      * ensure all the unknown bits are 0 - i.e. new
34.      * user-space does not rely on any kernel feature
35.      * extensions we dont know about yet.
36.      */
37.     if (size > sizeof(*attr)) { 만약 size 가 우리가 아는 구조체 크기보다 큰 경우 모르는
    모든 비트가 0 인지 확인
38.         unsigned char __user *addr;
39.         unsigned char __user *end;
40.         unsigned char val;
41.
42.         addr = (void __user *)uattr + sizeof(*attr);
43.         end  = (void __user *)uattr + size;
44.
45.         for (; addr < end; addr++) { size 와 구조체 크기만큼 get_user 을 통해 비트 확인

46.             ret = get_user(val, addr);
47.             if (ret)
48.                 return ret;
49.             if (val)
50.                 goto err_size;
51.         }
52.         size = sizeof(*attr);
53.     }
54.
55.     ret = copy_from_user(attr, uattr, size); // get_user 와 다르게 size 를 지정하여 유
    저 영역의 데이터를 커널 영역으로 복사, 정상적 수행됐다면 0 을 리턴한다 return 값에는 복사
    되지 않은 바이트 수가 들어간다.
56.     if (ret)
57.         return -EFAULT;  /* Bad address */
58.
59.     /*
60.      * XXX: Do we want to be lenient like existing syscalls; or do we want
61.      * to be strict and return an error on out-of-bounds values?
62.      */
63.     attr->sched_nice = clamp(attr->sched_nice, MIN_NICE, MAX_NICE);
64.             //nice 를 MIN 과 MAX 사이의 값으로 변경되게 해준다.
65.     return 0;
66.
67. err_size:
68.     put_user(sizeof(*attr), &uattr->size);
69.     return -E2BIG;
70. }
```

arch/alpha/include/asm/uaccess.h get_user()

lines 59

```
#define get_user(x, ptr) \  용도 : 유저 영역의 데이터를 커널 영역으로 복사
             인자의 넘어온 변수의 길이만큼 복사한다.
             성공하면 0 오류는 -EFAULT, 오류 발생시 변수 x가 0으로 설정*/
   __get_user_check((x), (ptr), sizeof(*(ptr)))
```

lines 102 __get_user_check

```
#define __get_user_check(x, ptr, size)              \
({                              \
    long __gu_err = -EFAULT;                \ /* Bad address */
    unsigned long __gu_val = 0;             \
    const __typeof__(*(ptr)) __user *__gu_addr = (ptr); \
    if (__access_ok((unsigned long)__gu_addr, size)) {  \
        __gu_err = 0;                   \
        switch (size) {                 \
          case 1: __get_user_8(__gu_addr); break;   \
          case 2: __get_user_16(__gu_addr); break;  \
          case 4: __get_user_32(__gu_addr); break;  \
          case 8: __get_user_64(__gu_addr); break;  \
          default: __get_user_unknown(); break;     \
        }                           \
    }                           \
    (x) = (__force __typeof__(*(ptr))) __gu_val;        \
    __gu_err;                           \
})
```

lines 57 put_user

```
#define put_user(x, ptr) \  용도 : 커널 영역의 데이터를 유저 영역으로 복사
             인자의 넘어온 변수의 길이만큼 복사한다.
             성공하면 0 오류는 -EFAULT, 오류 발생시 변수 x가 0으로 설정
   __put_user_check((__typeof__(*(ptr)))(x), (ptr), sizeof(*(ptr)))
```

lines 198 __put_user_check

```
#define __put_user_check(x, ptr, size)              \
({                              \
    long __pu_err = -EFAULT;                \/* Bad address */
    __typeof__(*(ptr)) __user *__pu_addr = (ptr);       \
    if (__access_ok((unsigned long)__pu_addr, size)) {  \
        __pu_err = 0;                   \
        switch (size) {                 \
          case 1: __put_user_8(x, __pu_addr); break;    \
          case 2: __put_user_16(x, __pu_addr); break;   \
          case 4: __put_user_32(x, __pu_addr); break;   \
          case 8: __put_user_64(x, __pu_addr); break;   \
```

```
          default: __put_user_unknown(); break;     \
        }                           \
    }                               \
    __pu_err;                           \
})
```

include/linux/uaccess.h copy_from_user()

lines 144

```
copy_from_user(void *to, const void __user *from, unsigned long n)
{
    if (likely(check_copy_size(to, n, false))) // likely = if 분기문에서 True인 경우가 더 많을 것이라는 정보를 주어 성능 향상 시키는 함수

        n = _copy_from_user(to, from, n);
    return n;
}
```

include/linux/thread_info.h check_copy_size()

lines 138

```
check_copy_size(const void *addr, size_t bytes, bool is_source)
{
    int sz = __compiletime_object_size(addr);
    if (unlikely(sz >= 0 && sz < bytes)) { //likely 와 반대
        if (!__builtin_constant_p(bytes))
            copy_overflow(sz, bytes);
        else if (is_source)
            __bad_copy_from();
        else
            __bad_copy_to();
        return false;
    }
    check_object_size(addr, bytes, is_source);
    return true;
}
```

include/linux/uaccess.h _copry_from_user()

lines 110

```
#ifdef INLINE_COPY_FROM_USER // 이 옵션을 통해서 유저 어플리케이션에서 사용 될 때 인라인 함수로 제공, 커널에서 호출시에는 라이브러리를 통해 제공
static inline unsigned long
_copy_from_user(void *to, const void __user *from, unsigned long n)
{ //user address(from)에서 kernel address(to)로 nbytes(n)만큼 복사한다.
    unsigned long res = n;
    might_fault();
    if (likely(access_ok(VERIFY_READ, from, n))) {
        kasan_check_write(to, n);
        res = raw_copy_from_user(to, from, n);
    }
```

```
11.     if (unlikely(res))
12.         memset(to + (n - res), 0, res);
13.     return res;
14. } 정상적 수행됐다면 0을 리턴한다 return 값에는 복사되지 않은 바이트 수가 들어간다.
15. #else
16. extern unsigned long
17. _copy_from_user(void *, const void __user *, unsigned long);
18. #endif
```

include/linux/rcupdate.h rcu_read_lock()

lines 627

```
1.  static inline void rcu_read_lock(void)  // RCU에서 데이터 참조를 위한 Critical Sectio
    n을 시작 했다는 것을 통지
2.  {
3.      __rcu_read_lock();
4.      __acquire(RCU); // 읽기를 위한 잠금 권한을 획득한다. 읽는 것은 다수의 스레드에서 접근
    이 가능하므로 다른 CPU에서 rcu_read_lock() 함수를 이용하여 중첩해서 사용이 가능하다.
5.      rcu_lock_acquire(&rcu_lock_map);
6.      RCU_LOCKDEP_WARN(!rcu_is_watching(),
7.               "rcu_read_lock() used illegally while idle");
8.  }
```

lines 80 __rcu_read_lock()

```
1.  static inline void __rcu_read_lock(void)
2.  {
3.      if (IS_ENABLED(CONFIG_PREEMPT_COUNT))
4.          preempt_disable();  // CPU간의 선점을 방지한다.
5.  }
```

lines 679 rcu_read_unlock()

```
1.  static inline void rcu_read_unlock(void)
2.  {
3.      RCU_LOCKDEP_WARN(!rcu_is_watching(),
4.               "rcu_read_unlock() used illegally while idle");
5.      __release(RCU);   // __acquire()에서 획득한 잠금을 해제한다.
6.      __rcu_read_unlock();
7.      rcu_lock_release(&rcu_lock_map); /* Keep acq info for rls diags. */
8.  }
```

lines 86 __rcu_read_unlock()

```
1.  static inline void __rcu_read_unlock(void)
2.  {
3.      if (IS_ENABLED(CONFIG_PREEMPT_COUNT))
4.          preempt_enable(); // __rcu_read_lock에서 CPU간의 선점을 방지를 해제한다.
5.  }
```

kernel/sched/core.c

lines 4421 sched_setattr()

```
1.  int sched_setattr(struct task_struct *p, const struct sched_attr *attr)
2.  {
3.      return __sched_setscheduler(p, attr, true, true);
4.  }
```

lines 4143 __sched_setscheduler()

```
1.  static int __sched_setscheduler(struct task_struct *p, const struct sched_attr *att
    r, bool user, bool pi)
2.  {
3.      int newprio = dl_policy(attr-
    >sched_policy) ? MAX_DL_PRIO - 1 : // 정책이 deadline 인지 확인 MAX_DL_PRIO = 0
4.                  MAX_RT_PRIO - 1 - attr->sched_priority;
5.      int retval, oldprio, oldpolicy = -1, queued, running;
6.      int new_effective_prio, policy = attr->sched_policy;
7.      const struct sched_class *prev_class;
8.      struct rq_flags rf;
9.      int reset_on_fork;
10.     int queue_flags = DEQUEUE_SAVE | DEQUEUE_MOVE | DEQUEUE_NOCLOCK;
11.             // 0x02 /* Matches ENQUEUE_RESTORE */
12.             //  0x04 /* Matches ENQUEUE_MOVE */
13.             //      0x08 /* Matches ENQUEUE_NOCLOCK */
14.     struct rq *rq;
15.
16.     /* The pi code expects interrupts enabled */
17.     BUG_ON(pi && in_interrupt()); // in_interrupt()는 현재 실행 중인 코드가 interrupt
     context 인지 true 로 반환해준다. false 는 process context
18. recheck:
19.     /* Double check policy once rq lock held: */
20.     if (policy < 0) {
21.         reset_on_fork = p->sched_reset_on_fork;
22.         policy = oldpolicy = p->policy;
23.     } else {
24.         reset_on_fork = !!(attr->sched_flags & SCHED_FLAG_RESET_ON_FORK);
25.
26.         if (!valid_policy(policy)) // idle || fair(normal, batch) || rt(fifo, rr) |
    | dl  인지 확인
27.             return -EINVAL; // /* Invalid argument */
28.     }
29.
30.     if (attr->sched_flags & ~(SCHED_FLAG_ALL | SCHED_FLAG_SUGOV)) // attr-
    >sched_flags & ~(0x01 | 0x02 | 0x04 | 0x10000000) 즉 XOOOOXXX bit 에서 O 에 1bit 가
    들어가게 된 경우 에러 검출
31.                         //
32.         return -EINVAL;  /* Invalid argument */
33.
34.     /*
35.      * Valid priorities for SCHED_FIFO and SCHED_RR are
36.      * 1..MAX_USER_RT_PRIO-1, valid priority for SCHED_NORMAL,
37.      * SCHED_BATCH and SCHED_IDLE is 0.
38.      */
39.     // mm_struct 인 memory map mm 변수  MAX_RT_PRIO 와 MAX_USER_RT_PRIO 는 100 으로 매크
    로 되어있다.
40.     if ((p->mm && attr->sched_priority > MAX_USER_RT_PRIO-1) ||
```

```
41.         (!p->mm && attr->sched_priority > MAX_RT_PRIO-
    1))      // sched_RT 인 경우 우선순위가 100 이 넘는 경우 오류를 검출한다.
42.         return -EINVAL;  /* Invalid argument */ sched
43.     if ((dl_policy(policy) && !__checkparam_dl(attr)) ||    // policy 가 deadline 이
    면서 deaeline 에서 매개변수나 deadline 이 0 인 경우, SCALE bit, MSB 설정 에러 등 에러 검
    출
44.         (rt_policy(policy) != (attr-
    >sched_priority != 0))) // policy 가 rt 인 경우 우선순위가 0 인 경우, rt 가 아닌데 우선순
    위가 있는 경우 에러 검출
45.         return -EINVAL;  /* Invalid argument */
46.
47.     /*
48.      * Allow unprivileged RT tasks to decrease priority:
49.      */   우선 순위를 높이고 다른 우선 순위를 설정하도록 허용, (다른 UID) 프로세스 자체적
    으로 FIFO 및 라운드 로빈 (실시간) 스케줄링 사용 허용, 다른 사람이 사용하는 스케줄링 알고리
    즘 처리 및 설정 방법.
50. 다른 프로세스에서 CPU 선호도 설정 허용
51.     if (user && !capable(CAP_SYS_NICE)) {  // nice 값을 사용하는 지 확인 (fair, rt)
52.
53.         if (fair_policy(policy)) { // fair 인 경우 nice 값이 -20~19 사이인지 체크
54.             if (attr->sched_nice < task_nice(p) &&
55.                 !can_nice(p, attr->sched_nice))
56.                 return -EPERM; /* Operation not permitted */
57.         }
58.
59.         if (rt_policy(policy)) { // rt 인 경우 우선순위 에러 체크
60.             unsigned long rlim_rtprio =
61.                     task_rlimit(p, RLIMIT_RTPRIO);
62.
63.             /* Can't set/change the rt policy: */
64.             if (policy != p->policy && !rlim_rtprio)
65.                 return -EPERM; /* Operation not permitted */
66.
67.             /* Can't increase priority: */
68.             if (attr->sched_priority > p->rt_priority &&
69.                 attr->sched_priority > rlim_rtprio)
70.                 return -EPERM; /* Operation not permitted */
71.         }
72.
73.          /*
74.           * Can't set/change SCHED_DEADLINE policy at all for now
75.           * (safest behavior); in the future we would like to allow
76.           * unprivileged DL tasks to increase their relative deadline
77.           * or reduce their runtime (both ways reducing utilization)
78.           */
79.         if (dl_policy(policy)) // 여기서는 deadline 이용 x
80.             return -EPERM; /* Operation not permitted */
81.
82.         /*
83.          * Treat SCHED_IDLE as nice 20. Only allow a switch to
84.          * SCHED_NORMAL if the RLIMIT_NICE would normally permit it.
85.          */
86.         if (idle_policy(p-
    >policy) && !idle_policy(policy)) { // idle 은 nice 20 으로 취급, nice 값 에러
87.             if (!can_nice(p, task_nice(p)))
88.                 return -EPERM; /* Operation not permitted */
89.         }
90.
91.         /* Can't change other user's priorities: */
92.         if (!check_same_owner(p)) // 다른 유저 우선순위 건들기 x
93.             return -EPERM; /* Operation not permitted */
94.
95.         /* Normal users shall not reset the sched_reset_on_fork flag: */
```

```
96.         if (p->sched_reset_on_fork && !reset_on_fork) //   sched_reset_on_fork 리셋 x
97.             return -EPERM; /* Operation not permitted */
98.     }
99.
100.     if (user) {
101.         if (attr->sched_flags & SCHED_FLAG_SUGOV)
102.             return -EINVAL; /* Invalid argument */
103.
104.         retval = security_task_setscheduler(p); // cpuset을 사용할 수 있는지 확인
105.         if (retval)
106.             return retval;
107.     }
108.
109.     /*
110.      * Make sure no PI-waiters arrive (or leave) while we are
111.      * changing the priority of the task:
112.      *
113.      * To be able to change p->policy safely, the appropriate
114.      * runqueue lock must be held.
115.      */
116.     rq = task_rq_lock(p, &rf); // policy을 바꿀 때 PI waiter (pi 란 프로세스설정이 상속이 되냐는 Boolean 변수다) 가 사라지지 않도록 런큐를 잠근다.
117.     update_rq_clock(rq);
118.
119.     /*
120.      * Changing the policy of the stop threads its a very bad idea:
121.      */
122.     if (p == rq->stop) { 스레드 정지정책 바뀌면 에러
123.         task_rq_unlock(rq, p, &rf);
124.         return -EINVAL; /* Invalid argument */
125.     }
126.
127.     /*
128.      * If not changing anything there's no need to proceed further,
129.      * but store a possible modification of reset_on_fork.
130.      */
131.     if (unlikely(policy == p->policy)) {
132.         if (fair_policy(policy) && attr->sched_nice != task_nice(p)) fair 이면서 nice가 변경되었다면 goto
133.             goto change;
134.         if (rt_policy(policy) && attr->sched_priority != p->rt_priority) rt 이면서 우선순위가 변경되었다면 goto
135.             goto change;
136.         if (dl_policy(policy) && dl_param_changed(p, attr)) // deadline 이면서 매개변수 변하면 goto
137.             goto change;
138.
139.         p->sched_reset_on_fork = reset_on_fork; // 아무것도 바꾸지 않았을 때도 따로 저장해서 관리

140.         task_rq_unlock(rq, p, &rf);
141.         return 0;
142.     }
143. change:
144.
145.     if (user) {
146. #ifdef CONFIG_RT_GROUP_SCHED // cgroup의 rt 그룹 스케줄링을 지원하는 커널 옵션
147.         /*
148.          * Do not allow realtime tasks into groups that have no runtime
149.          * assigned.
150.          */
```

```
151.         if (rt_bandwidth_enabled() && rt_policy(policy) && // 런타임 작업 그룹이 할
    당되어 있지 않으면서 rt 일 경우 에러
152.                 task_group(p)->rt_bandwidth.rt_runtime == 0 &&
153.                 !task_group_is_autogroup(task_group(p))) {
154.             task_rq_unlock(rq, p, &rf);
155.             return -EPERM; /* Operation not permitted */
156.         }
157. #endif
158. #ifdef CONFIG_SMP // Symmetric multiprocessing 개별적으로 연산이 돌아감
159.         if (dl_bandwidth_enabled() && dl_policy(policy) && //dl 이면서 대역폭 존재
160.                 !(attr->sched_flags & SCHED_FLAG_SUGOV)) { // dl 에러 검출 x 시
161.             cpumask_t *span = rq->rd->span;
162.
163.             /*
164.              * Don't allow tasks with an affinity mask smaller than
165.              * the entire root_domain to become SCHED_DEADLINE. We
166.              * will also fail if there's no bandwidth available.
167.              */
168.             if (!cpumask_subset(span, &p-
    >cpus_allowed) || // cpumask < root_domain 이거나
169.                 rq->rd->dl_bw.bw == 0) { // 런큐에 대역폭이 없으면 에러
170.                 task_rq_unlock(rq, p, &rf);
171.                 return -EPERM; /* Operation not permitted */
172.             }
173.         }
174. #endif
175.     }
176.
177.     /* Re-check policy now with rq lock held: */
178.     if (unlikely(oldpolicy != -1 && oldpolicy != p-
    >policy)) { // 다시 한번 체크해보기 위한 goto
179.         policy = oldpolicy = -1;
180.         task_rq_unlock(rq, p, &rf);
181.         goto recheck;
182.     }
183.
184.     /*
185.      * If setscheduling to SCHED_DEADLINE (or changing the parameters
186.      * of a SCHED_DEADLINE task) we need to check if enough bandwidth
187.      * is available.
188.      */
189.     if ((dl_policy(policy) || dl_task(p)) && sched_dl_overflow(p, policy, attr)) {
    //deadline 으로 바꾸거나 매개변수를 바꾸는 경우에 충분한 대역폭이 있나 확인하기
190.         task_rq_unlock(rq, p, &rf);
191.         return -EBUSY;
192.     }
193.
194.     p-
    >sched_reset_on_fork = reset_on_fork; // 현재 스케쥴링과 우선순위를 이전 변수에 넣는다.

195.     oldprio = p->prio;
196.
197.     if (pi) {
198.         /*
199.          * Take priority boosted tasks into account. If the new
200.          * effective priority is unchanged, we just store the new
201.          * normal parameters and do not touch the scheduler class and
202.          * the runqueue. This will be done when the task deboost
203.          * itself.
204.          */
205.         new_effective_prio = rt_effective_prio(p, newprio); // rt 의 새 우선순위 할당

206.         if (new_effective_prio == oldprio) // 전 우선순위와 같다면 그대로 유지
```

```
207.             queue_flags &= ~DEQUEUE_MOVE;
208.     }
209.
210.     queued = task_on_rq_queued(p); // p가 런큐에 큐 되어 있나
211.     running = task_current(rq, p); // 현재 런큐에 올라가있나
212.     if (queued)
213.         dequeue_task(rq, p, queue_flags); 프로세스가 더이상 실행 가능할 상태가 아닐 때
214.     if (running)
215.         put_prev_task(rq, p); 실행중인 테스크를 다시 queue에 넣는다
216.
217.     prev_class = p->sched_class; 현재 클래스를 prev에 저장
218.     __setscheduler(rq, p, attr, pi); 스케쥴러를 설정한다.
219.
220.     if (queued) {
221.         /*
222.          * We enqueue to tail when the priority of a task is
223.          * increased (user space view).
224.          */
225.         if (oldprio < p->prio) // 바뀐 우선순위가 더 큰 경우 큐에서 자리를 바꿔준다.
226.             queue_flags |= ENQUEUE_HEAD;
227.
228.         enqueue_task(rq, p, queue_flags); // 프로세스가 실행가능한 상태로 들어간다.
229.     }
230.     if (running)
231.         set_curr_task(rq, p); 테스크의 스케쥴링 클래스나 태스크 그룹을 바꿀때
232.
233.     check_class_changed(rq, p, prev_class, oldprio); // 전 클래스와 전 우선순위로 현
    재의 클래스와 우선순위를 비교하여 제대로 바뀌었는지 확인
234.
235.     /* Avoid rq from going away on us: */
236.     preempt_disable(); //선점 지연 비활성화
237.     task_rq_unlock(rq, p, &rf); // policy을 바꿨으니 잠금을 해제한다.
238.
239.     if (pi)
240.         rt_mutex_adjust_pi(p); // 우선순위 설정을 한 경우에 pi 체인을 다시 확인함
241.
242.     /* Run balance callbacks after we've adjusted the PI chain: */
243.     balance_callback(rq); // rq lock을 acquire 실패시 rq lock과 선점을 다른
    프로세스가 가지고 있을시에 rq에서 탈취함. 이 때 선점 지연 비활성화 상태다.
244.     preempt_enable(); //선점 지연 활성화
245.
246.     return 0;
247. }
```

kernel/sched/sched.h inline으로 정의된 policy

```
1.  static inline int idle_policy(int policy)
2.  {
3.      return policy == SCHED_IDLE;
4.  }
5.  static inline int fair_policy(int policy)
6.  {
7.      return policy == SCHED_NORMAL || policy == SCHED_BATCH;
8.  }
9.
10. static inline int rt_policy(int policy)
11. {
12.     return policy == SCHED_FIFO || policy == SCHED_RR;
```

```
}

static inline int dl_policy(int policy)
{
    return policy == SCHED_DEADLINE;
}
static inline bool valid_policy(int policy)
{
    return idle_policy(policy) || fair_policy(policy) ||
        rt_policy(policy) || dl_policy(policy);
}
```

kernel/sched/core.c __setscheduler()

lines 4106

```
static void __setscheduler(struct rq *rq, struct task_struct *p,
                const struct sched_attr *attr, bool keep_boost)
{
    __setscheduler_params(p, attr);

    /*
     * Keep a potential priority boosting if called from
     * sched_setscheduler().
     */
    p->prio = normal_prio(p); // task의 우선순위를 각 정책에 맞게 맞춰주고
    if (keep_boost) // pi가 true인 경우 즉 향상된 우선순위를 제공한 경우
        p->prio = rt_effective_prio(p, p->prio);

    if (dl_prio(p->prio)) // 각각 설정한 스케쥴링 정책에 맞게 클래스도 변경한다.
        p->sched_class = &dl_sched_class;
    else if (rt_prio(p->prio))
        p->sched_class = &rt_sched_class;
    else
        p->sched_class = &fair_sched_class;
}
```

lines 4080 __setscheduler_params()

```
static void __setscheduler_params(struct task_struct *p,
        const struct sched_attr *attr)
{
    int policy = attr->sched_policy; // policy에 받아온 policy를 넣는다

    if (policy == SETPARAM_POLICY) // setparam_policy은 -1로 -
1 인 경우에는 함수가 변경하지 않도록 해줌
        policy = p->policy;

    p->policy = policy; // task의 policy를 현재 policy로 바꿔주고

    if (dl_policy(policy))  // dl인경우 dl에 맞는 매개변수(runtime, deadline, period
, flags, bw, density)를 맞춰준다
        __setparam_dl(p, attr);
```

```
13.     else if (fair_policy(policy)) // fair인 경우 현재 nice에서 default prio를 더해주는데 이 때 default prio는 120값을 가진다 (nice 범위 -20~19 = 40, MAX_RT_PRIO = 100, 100 + 20) 이 것을 static_prio에 넣어준다.
14.         p->static_prio = NICE_TO_PRIO(attr->sched_nice);
15.
16.     /*
17.      * __sched_setscheduler() ensures attr->sched_priority == 0 when
18.      * !rt_policy. Always setting this ensures that things like
19.      * getparam()/getattr() don't report silly values for !rt tasks.
20.      */
21.     p->rt_priority = attr->sched_priority;
22.     p->normal_prio = normal_prio(p); // policy가 무엇이냐에 따라서 각각 최대 우선순위에서 1을 뺀 값을 넣어준다
23.     set_load_weight(p, true); // load_weight란 공정한 cpu시간을 할당하기 위하여 우선순위의 비율을 정한 것이다.  위에서 하지 않은 정책들인 Idle과 other인 경우에 필요한 load_weight을 설정해준다.
24. }
```

1-3 2) 스케쥴링 클래스 분석 (rt클래스)

kernel/sched/sched.h

lines 1558

```
1.  extern const struct sched_class stop_sched_class;
2.  extern const struct sched_class dl_sched_class;
3.  extern const struct sched_class rt_sched_class;
4.  extern const struct sched_class fair_sched_class;
5.  extern const struct sched_class idle_sched_class;
```

5개의 클래스 정의

Lines 1476 struct sched_class

```
1.  struct sched_class {
2.      const struct sched_class *next;  // 우선순위 순서 stop -> dl -> rt -> fair -> idle (for_each_class()에서 돔)
3.      void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags); // 프로세스가 실행 가능한 상태로 진입
4.      void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags); // 프로세스가 더 이상 실행 가능한 상태가 아닐때
5.      void (*yield_task)   (struct rq *rq); // 프로세스가 스스로 yield() 시스템콜을 실행했을 때
6.      bool (*yield_to_task)(struct rq *rq, struct task_struct *p, bool preempt); 현재 프로세서를 다른 스레드에 양보하거나 스레드를 프로세서쪽으로 가속시킴
7.
8.      void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags); // 현재 실행 중인 프로세스를 선점할 수 있는지 검사
9.
10.     /*
11.      * It is the responsibility of the pick_next_task() method that will
12.      * return the next task to call put_prev_task() on the @prev task or
13.      * something equivalent.
14.      *
15.      * May return RETRY_TASK when it finds a higher prio class has runnable
16.      * tasks.
17.      */
```

```
18.     struct task_struct * (*pick_next_task)(struct rq *rq, struct task_struct *prev,
      struct rq_flags *rf); //실행할 다음 프로세스를 선택
19.     void (*put_prev_task)(struct rq *rq, struct task_struct *p); // 실행중인 태스크를
     다시 내부 자료구조에 큐잉
20.
21. #ifdef CONFIG_SMP // SMP 설정일 때 (symmetric multiprocessing)
22.     int  (*select_task_rq)(struct task_struct *p, int task_cpu, int sd_flag, int fl
    ags); //  wake 또는 fork 밸런싱인 경우에만 가장 낮은 우선 순위부터 요청한 태스크의 우선순
    위 범위 이내에서 동작할 수 있는 cpu 를 찾아 선택한다.
23.     void (*migrate_task_rq)(struct task_struct *p);
24.
25.     void (*task_woken)(struct rq *this_rq, struct task_struct *task); // task 가 실행
    중이지 않고 조정하지 않는다면 밀어낸다
26.
27.     void (*set_cpus_allowed)(struct task_struct *p, const struct cpumask *newmask);
     // 요청 태스크가 운영될 수 있는 cpu 들을 지정한다
28.
29.     void (*rq_online)(struct rq *rq); // rq 온라인 설정
30.     void (*rq_offline)(struct rq *rq); // rq 오프라인 설정
31. #endif
32.
33.     void (*set_curr_task)(struct rq *rq);  //태스크의 스케줄링 클래스나 태스크 그룹을
    바꿀때
34.     void (*task_tick)(struct rq *rq, struct task_struct *p, int queued); // 타이머
    틱 함수가 호출
35.     void (*task_fork)(struct task_struct *p); // cfs 인 경우 fork 된 새 child task 의
    vruntime 을 결정해준다.
36.     void (*task_dead)(struct task_struct *p); // dl 인 경우 total_bw 속 dl_bw 감소후
    dl 타이머를 중지시켜준다.
37.
38.     /*
39.      * The switched_from() call is allowed to drop rq->lock, therefore we
40.      * cannot assume the switched_from/switched_to pair is serliazed by
41.      * rq->lock. They are however serialized by p->pi_lock.
42.      */
43.     void (*switched_from)(struct rq *this_rq, struct task_struct *task); // from 으
    로부터 policy 교체, 큐에 있는 마지막 작업이면 pull 을 통해 다른 작업을 불러온다.
44.     void (*switched_to)  (struct rq *this_rq, struct task_struct *task); // to 로 po
    licy 교체
45.     void (*prio_changed) (struct rq *this_rq, struct task_struct *task, int oldprio
    ); //  task 의 priority 가 같은 type 내에서 바뀔 때 호출
46.
47.     unsigned int (*get_rr_interval)(struct rq *rq, struct task_struct *task); // rt
    클래스에서 사용 rr 인 경우 인터벌을 리턴하고 fifo 인 경우 0 리턴
48.
49.     void (*update_curr)(struct rq *rq);// 현재 런타임을 업데이트한다.
50.
51. #define TASK_SET_GROUP      0
52. #define TASK_MOVE_GROUP     1
53.
54. #ifdef CONFIG_FAIR_GROUP_SCHED // cgroup 의 cfs 그룹 스케줄링을 지원하는 커널 옵션
55.
56.     void (*task_change_group)(struct task_struct *p, int type);
57. #endif
58. };
```

그 중 rt클래스의 구조체를 분석해보았다.

kernel/sched/rt.c rt_sched_class

lines 2366

```
1.  const struct sched_class rt_sched_class = {
2.      .next           = &fair_sched_class, // 우선순위 순서 stop -> dl -> rt -> fair -
    > idle (for_each_class)
3.      .enqueue_task       = enqueue_task_rt, // 프로세스가 실행 가능한 상태로 진입
4.      .dequeue_task       = dequeue_task_rt, //프로세스가 더 이상 실행 가능한 상태가 아닐
    때
5.      .yield_task     = yield_task_rt, //프로세스가 스스로 yield() 시스템콜을 실행했을 때

6.
7.      .check_preempt_curr = check_preempt_curr_rt,//현재 실행 중인 프로세스를 선점(preem
    pt)할 수 있는지 검사
8.
9.      .pick_next_task     = pick_next_task_rt,//실행할 다음 프로세스를 선택
10.     .put_prev_task      = put_prev_task_rt,//실행중인 태스크를 다시 내부 자료구조에 큐
    잉
11.
12. #ifdef CONFIG_SMP
13.     .select_task_rq     = select_task_rq_rt, // wake 또는 fork 밸런싱인 경우에만 가장
    낮은 우선 순위부터 요청한 태스크의 우선순위 범위 이내에서 동작할 수 있는 cpu를 찾아 선택한
    다.
14.
15.     .set_cpus_allowed       = set_cpus_allowed_common, //요청 태스크가 운영될 수 있는
    cpu 들을 지정한다
16.     .rq_online              = rq_online_rt, // rq 온라인 설정
17.     .rq_offline             = rq_offline_rt, // rq 오프라인 설정
18.     .task_woken     = task_woken_rt, // task가 실행중이지 않고 조정하지 않는다면 밀어
    낸다
19.     .switched_from      = switched_from_rt, // rt에서 클래스 바꿈
20. #endif
21.
22.     .set_curr_task          = set_curr_task_rt, //태스크의 스케줄링 클래스나 태스크 그
    룹을 바꿀때
23.     .task_tick      = task_tick_rt, // 타이머 틱 함수가 호출
24.
25.     .get_rr_interval    = get_rr_interval_rt, // rr인 경우 인터벌 리턴, fifo인 경우 0
     리턴
26.
27.     .prio_changed       = prio_changed_rt, // 우선순위 교체
28.     .switched_to        = switched_to_rt, // rt 클래스로 switch 함
29.
30.     .update_curr        = update_curr_rt, // 현재 런타임을 업데이트한다.
31. };
```

1-3 3) Printk()을 통한 콜체인 확인과 스케쥴링 클래스 (rt-rr) 분석 (캡쳐화면)

```
May 14 16:21:47 os201610674 kernel: [  695.798941] sys_sched_setattr() call!
May 14 16:21:47 os201610674 kernel: [  695.798944] sched_setattr() call!
May 14 16:21:47 os201610674 kernel: [  695.798944] __sched_setscheduler() call!
May 14 16:21:47 os201610674 kernel: [  695.798951] __setscheduler() call!
May 14 16:21:47 os201610674 kernel: [  695.798952] __setscheduler_params() call!
May 14 16:23:49 os201610674 kernel: [  695.798954] priority : 95 policy : 2
```

1-3 1) 그림과 같이 총 5개의 함수 콜체인이 일어나는 것을 알 수 있었다. 그리고 유저 어플리케이션에서 넣은 우선순위와 policy가 제대로 전달받았는지 확인하기 위해 출력해본 결과 제대로 전달 되었다.

```
May 15 00:47:50 os201610674 kernel: [  704.544483] sys_sched_setattr() call!
May 15 00:47:50 os201610674 kernel: [  704.544485] sched_setattr() call!
May 15 00:47:50 os201610674 kernel: [  704.544485] __sched_setscheduler() call!
May 15 00:47:50 os201610674 kernel: [  704.544491] __setscheduler() call!
May 15 00:47:50 os201610674 kernel: [  704.544492] __setscheduler_params() call!
May 15 00:47:50 os201610674 kernel: [  704.544493] priority : 95 policy : 2 pid : 2048
May 15 00:47:50 os201610674 kernel: [  704.643428] next process pid : 2049
May 15 00:47:50 os201610674 kernel: [  704.743419] next process pid : 2050
May 15 00:47:51 os201610674 kernel: [  704.843418] next process pid : 2051
May 15 00:47:51 os201610674 kernel: [  704.943417] next process pid : 2052
May 15 00:47:51 os201610674 kernel: [  705.043427] next process pid : 2048
May 15 00:47:51 os201610674 kernel: [  705.143416] next process pid : 2049
May 15 00:47:51 os201610674 kernel: [  705.247414] next process pid : 2050
May 15 00:47:51 os201610674 kernel: [  705.347415] next process pid : 2051
May 15 00:47:51 os201610674 kernel: [  705.447415] next process pid : 2052
```

```
2049 root     -96   0    4512     80      0 R  20.2  0.0   0:07.25 cpu
2048 root     -96   0    4512    856    796 R  19.9  0.0   0:07.29 cpu
2050 root     -96   0    4512     80      0 R  19.9  0.0   0:07.19 cpu
2051 root     -96   0    4512     80      0 R  19.9  0.0   0:07.20 cpu
2052 root     -96   0    4512     80      0 R  19.9  0.0   0:07.19 cpu
```

1-3 2)에서 분석한 스케쥴링 클래스 중 RT클래스에서 다음 프로세스를 정확히 찾아가는지 확인하기 위해 pick_next_task_rt() 함수에 다음 프로세스의 pid를 출력하게 하고 유저 어플리케이션을 프로세스 5개로 수행해보았다. 그 결과 유저 어플리케이션은 RT클래스 중 라운드 로빈으로 설정 되고나서 프로세스 5개가 돌아가면서 수행되는 것을 볼 수 있었다.

※ 특히 어려웠던 점 및 해결 방안

제일 먼저 1-1 유저 어플리케이션에서 각각 프로세스들의 시간을 구할 때 애를 먹었다. Get_time()함수를 처음 써보면서 time_spec 구조체를 공부하고 그 차이를 구할 때 앞쪽 시간의 nsec이 음수가 될 때 예외처리를 해줘야 하는 것을 생각하기 어려웠다. 인터넷에서 찾아본 함수를 인용하여 어플리케이션을 완성 시켰다.

그리고 1-3에서 sched_setattr()을 분석하는 부분에서 양이 상당히 방대했으며 꽤나 많은 함수에 들어가서 여러 동작을 취하는데 그 동작 하나하나 쫓아가면서 기능을 알아보았는데 어려웠다. 특히 pi변수는 느닷없이 등장했으나 제대로 된 설명을 찾기가 어려워서 이해가 안되었다. bootlin에 있는 주석으로는 확실히 설명이 부족하여 대부분 구글링을 하며 여러 설명들을 찾아보며 이해하였다. 이렇게 하나하나 분석해가니 더디지만 확실히 이해를 할 수 있었다. 전부 분석을 하며 공부를 같이 하니 콜 체인에 printk()을 커널에 넣어주니 한 번에 원하는 답을 얻어 낼 수 있었다.

처음 스케쥴링 클래스가 아닌 schedule() 함수를 분석하면서 스케쥴링되는 과정을 먼저 알게 되고 스케쥴링 클래스를 분석해보았다. 처음에는 생각을 잘못하여 함수를 먼저 분석하게 되었는데 이후 클래스를 보니까 구조체에서 어떤 부분이 어떤 역할을 하는지에 대해 쉽게 이해할 수 있었다.

마지막으로 커널을 수정하고 컴파일하면서 리눅스에 대해 조금 더 알게 된 거 같다. 항상 윈도우 운영체제에서 코딩만 해오다가 이번 과제를 처음에 직면했을 때는 마냥 어려워 보였지만 계속 공부하면서 추상적으로만 이해했던 개념들을 제대로 이해한 것 같다.