과제 #1
-리눅스 CPU 스케줄러 분석

학번 : 201711032
이름 : 고영훈

• 유저 프로그램 소스 코드 설명 (3장 이내)

sched_attr 구조체

```
22  struct sched_attr
23  {
24          uint32_t size;
25          uint32_t sched_policy;
26          uint64_t sched_flags;
27          int32_t sched_nice;
28          uint32_t sched_priority;
29          uint64_t sched_runtime;
30          uint64_t sched_deadline;
31          uint64_t sched_period;
32  };
```

- **size** : sched_attr 구조체의 크기를 정의합니다.
- **sched_policy** : 스케줄링 정책 설정(SCHED_*)
- **sched_flags** : 스케줄링 행동을 컨트롤
- **sched_nice** : nice value를 지정
- **sched_priority** : priority 설정 (낮을수록 높은 우선순위)
- **sched_runtime** : deadline 스케줄링에 사용되는 value
- **sched_deadline** : deadline 설정(nanosecond 단위)
- **sched_period** : 주기 설정(nanosecond 단위)

sched_setattr 함수

```
32  static int sched_setattr(pid_t pid, const struct sched_attr *attr, unsigned int flags)
33  {
34          return syscall(SYS_sched_setattr, pid, attr, flags);
35  }
```

- **pid_t pid** : 스케줄링 알고리즘을 변경하고자하는 프로세스의 pid
- **const struct sched_attr *attr** : 스케줄링 정보를 담은 구조체(const로 값 보존)
- **unsigned int flags** : flag는 따로 지정하지 않음.
- **return** : 인자로 상수로 정의된 sched_setattr 시스템 콜, pid, attr(구조체), flags를 넘긴다.

main에서의 사용

```
97          struct sched_attr attr;
98
99          memset(&attr, 0, sizeof(attr));
100         attr.size = sizeof(struct sched_attr);
101
102         attr.sched_priority = 95;
103         attr.sched_policy = SCHED_RR;
104
105         result = sched_setattr(getpid(), &attr, 0);
106
107         if(result == -1)
108         {
109                 perror("Error calling sched_setattr.");
110         }
111
```

- **(97행)** shced_attr 구조체 사용

- **(99행)** attr memset으로 메모리 초기화
- **(100행)** 구조체 크기만큼 attr.size에 넣음
- **(102행)** priority는 95로 설정
- **(103행)** policy를 SCHED_RR(라운드 로빈 스케줄링)으로 설정
- **(105행)** 위에서 정의한 sched_setattr함수 호출, 현재의 pid, 구조체, flag는 0으로 설정
- **(107행)** syscall함수가 -1을 return하면 오류

cpu 연산하는 calc 함수

```
47          clock_gettime(CLOCK_MONOTONIC, &begin);
48          while(1)
49          {
50                  for(int i=0; i< ROW; i++)
51                  {
52                          for(int j=0; j< COL; j++)
53                          {
54                                  for(k=0; k < COL; k++) matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
55                          }
56                  }
57                  count++;
58                  clock_gettime(CLOCK_MONOTONIC, &end);
59                  ms = (end.tv_sec - begin.tv_sec)*1000 + (end.tv_nsec - begin.tv_nsec)/1.0e6;
60                  if(ms>=(100*flag))
61                  {
62                          printf("PROCESS #%02d count = %02d %d\n", cpuid, count, 100);
63                          flag++;
64                  }
65                  if(ms >= time*1000) break;
66          }
67          printf("DONE!! PROCESS #%02d : %06d %ld\n", cpu, count, ms);
68          exit(0);
```

- **(47, 58행)** 시간 차 측정하기 위해 clock_gettime의 CLOCK_MONOTONIC 사용
- **(59행)** timespec 구조체의 tv_sec(초), tv_nsec(나노 초) 필드를 ms(밀리 초)으로 변환
- **(60행)** ms가 100이 지날때마다 출력해주기 위함. flag를 1만큼 증가시켜줌
- **(65행)** 요청한 cpu 수행시간을 다 했을 때 break
- **(68행)** exit(0)로 자식 프로세스 종료

fork() 수행

```
89          int ProcessNum = atoi(argv[1]);
90          int cpuTime = atoi(argv[2]);
91          int status;
92          int runProcess =0;
93          pid = (pid_t *)malloc(sizeof(pid_t) * ProcessNum);
94          while(runProcess < ProcessNum)
95          {
96                  printf("Creating Process: #%d\n", runProcess);
97                  pid[runProcess] = fork();
98                  if(pid[runProcess] < 0)
99                  {
100                         return -1;
101                 }
102                 else if(pid[runProcess] == 0)
103                 {
104                         calc(cpuTime, runProcess);
105                 }
106                 else
107                 {
108                         runProcess++;
109                 }
110         }
111         for(int i=0; i<ProcessNum; i++)
112                 wait(0);
113         return 0;
114 }
```

- **(89, 90행)** Command line argument를 atoi로 변환하여 대입
- **(93행)** pid를 malloc(), pid를 malloc으로 한 이유는 signal handler에서도 pid 접근이 필요하기때문에 pid를 전역변수로 선언하고, 이에 따라 동적할당이 필요함
- **(94행)** 프로세스의 개수만큼 fork()를 수행. 부모가 직접 cpu연산 하지 않고 자식이 cpu연산 하게끔 수행.
- **(97행)** fork() 수행 후 반환된 pid 값을 pid배열에 저장
- **(100행)** fork의 반환값이 0보다 작으면 fork() 오류 발생(fork 오류)
- **(102~105행)** pid값이 0이면 자식 프로세스임을 의미. cpu 연산 수행
- **(106~109행)** 그 외의 경우는 pid>0, 부모 프로세스임을 의미. runProcess를 증가
- **(111, 112행)** wait(0)으로 자식 프로세스가 종료 될 때까지 부모프로세스는 종료하지 않고 기다림.

signal handler 구현

```
16  long ms;
17  int count, cpuid, parentPid;
18  pid_t *pid;
```

- **(16~17행)** signal handler 함수에서 접근이 필요한 변수들은 전역 변수로 설정.
- **(28행)** pid도 sigint_handler함수에서 접근이 필요하여 전역 변수로 선언.(main에서 동적할당)

main()에서 signal 호출

```
89  int main(int argc, char *argv[])
90  {
91          parentPid = getpid();
92          signal(SIGINT, sigint_handler);
```

- **(91행)** 부모프로세스의 pid를 parentPid 변수에 저장
- **(92행)** <signal.h>의 signal() 호출,
  재정의할 signal은 SIGINT(2번 시그널),
  재정의 함수는 sigint_handler()

sigint_handler 함수

```
79  void sigint_handler(int signo)
80  {
81          if(getpid()==parentPid) {}
82          else
83          {
84                  printf("DONE!! PROCESS #%02d : %06d %ld\n", cpuid, count, ms);
85          }
86          exit(0);
87  }
```

- **(81행)** 부모프로세스는 cpu수행을 하지 않았으므로 출력을 하지 않음
- **(82~85행)** 현재까지의 cpu 수행 결과를 출력
- **(86행)** exit(0)으로 부모, 자식 프로세스를 종료

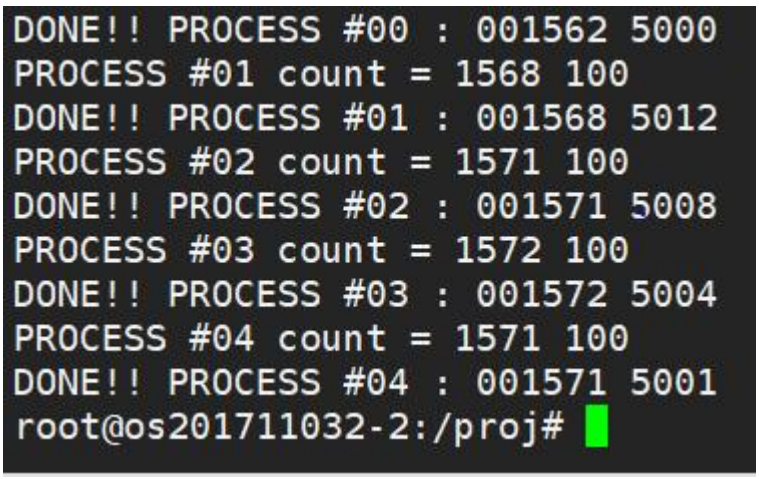
Ctrl+c를 입력했을 때 결과

```
root@os201711032-2:/proj# ./cpu 5 5
Creating Process: #0
Creating Process: #1
Creating Process: #2
Creating Process: #3
Creating Process: #4
```

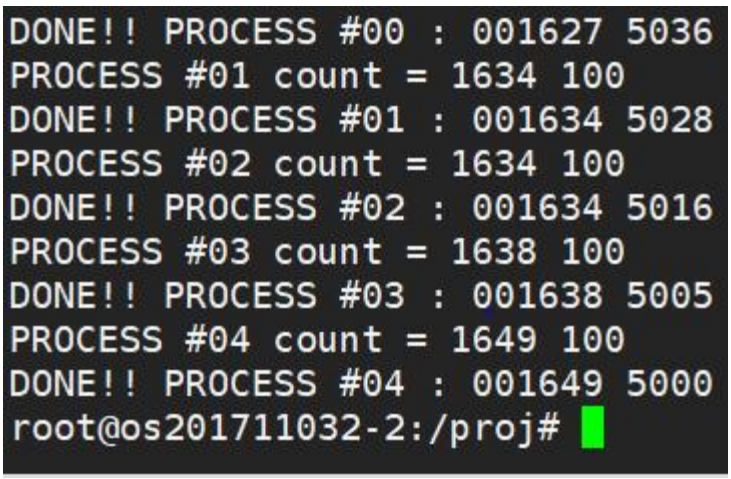
```
PROCESS #02 count = 322 100
PROCESS #02 count = 323 100
PROCESS #02 count = 324 100
^CDONE!! PROCESS #02 : 000328 1005
DONE!! PROCESS #03 : 000314 599
DONE!! PROCESS #04 : 000321 599
DONE!! PROCESS #00 : 000432 1099
DONE!! PROCESS #01 : 000454 1099
root@os201711032-2:/proj#
```

• 캡처 화면 3개 (분량 제한 없음)

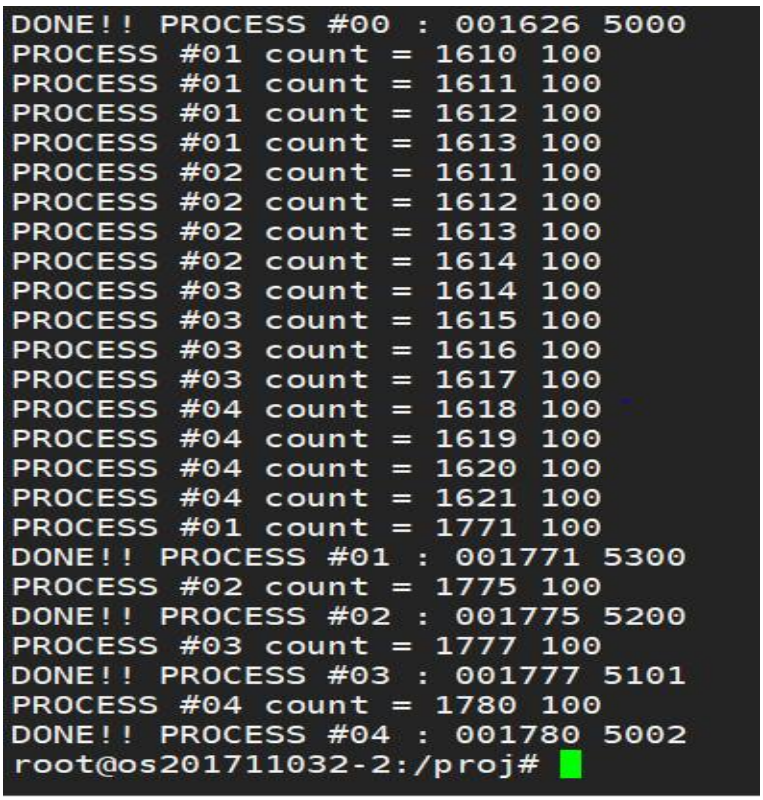
1. 5개 프로세스를 TIMESLICE 1ms 로 5초간 수행시켰을 때의 결과



2. 5개 프로세스를 TIMESLICE 10ms 로 5초간 수행시켰을 때의 결과



3. 5개 프로세스를 TIMESLICE 100ms 로 5초간 수행시켰을 때의 결과



Context switcing overhead 분석

RR Time slice	1ms		10ms		100ms	
	of calc	Time(ms)	of calc	Time(ms)	of calc	Time(ms)
Process #0	1562	5000	1627	5036	1626	5000
Process #1	1568	5012	1634	5028	1771	5300
Process #2	1571	5008	1634	5016	1775	5200
Process #3	1572	5004	1638	5005	1777	5101
Process #4	1571	5001	1649	5000	1780	5002
Total calc/ Max Time	**7,844**	**5012**	**8,182**	**5036**	**8,729**	**5300**

RR Time slice	1ms	10ms	100ms
Calculation per second(total calc/max time)	**1,565.04** (=7,844/5.012)	**1,624.70** (=8,182/5.036)	**1,646.98** (=8,729/5.300)
Baseline=1ms	100.00%	103.81%	105.23%
Baseline=10ms	96.32%	100.00%	101.37%

1ms와 **10ms** 비교 시 **3.81%**의 성능 향상,
10ms와 **100ms** 비교시 **1.37%**의 성능 향상이 있음.

timeslice 100ms 수행 결과 5.3초까지 수행되는 현상.
5초가 지난 경우 종료해야하지만 cpu가 waiting queue에서 기다리는동안 MONOTONIC의 시간은 흐르고 있으므로 늦게 종료됨. 실제 cpu수행은 5.3초보다 낮게 수행했을 것으로 예상됨.

• 리눅스 CPU 스케줄러 소스 코드 분석 보고서 (분량 제한 없음)

dmesg 결과

```
[   14.990063] random: crng init done
[   14.990118] random: 7 urandom warning(s) missed due to ratelimiting
[   39.436675] sched_setattr syscall
[   39.436677] sched_setattr call
[   39.438457] __sched_setscheduler call
[   39.439346] __setscheduler call
[   39.440165] __setscheduler_params call
[   39.440865] check_class_changed call
[   41.440007] sched: RT throttling activated
root@os201711032-2:/proj#
```

(kernel/sched/core.c)

sched_setattr syscall 정의

```
SYSCALL_DEFINE3(sched_setattr, pid_t, pid, struct sched_attr __user *, uattr,
                unsigned int, flags)
{
        struct sched_attr attr;
        struct task_struct *p;
        int retval;
        printk(KERN_DEBUG "sched_setattr syscall");
        if (!uattr || pid < 0 || flags)
                return -EINVAL;

        retval = sched_copy_attr(uattr, &attr);
        if (retval)
                return retval;

        if ((int)attr.sched_policy < 0)
                return -EINVAL;

        rcu_read_lock();
        retval = -ESRCH;
        p = find_process_by_pid(pid);
        if (p != NULL)
                retval = sched_setattr(p, &attr);
        rcu_read_unlock();

        return retval;
}
```

- 유저 프로그램 cpu.c에서 `return syscall(SYS_sched_setattr, pid, attr, flags);` 했을 때 위의 시스템 콜을 수행
- 프로세스의 정보를 관리하는 PCB인 **task_struct *p** 포인터는 syscall을 호출한 pid를 가리킨다.
- **sched_setattr()** 함수를 호출하기 전에 **rcu_read_lock()**으로 데이터(자료구조)를 참조하기 시작했다는 걸 알림. (RCU : Read Copy Update)

데이터를 참조하기 시작하면 다른 곳에선 접근 금지

- **sched_setattr(p, &attr)** 함수의 호출(setattr syscall과 __sched_setscheduler의 중간 다리 역할)
- **rcu_read_unlock()** : rcu에서 데이터 참조가 끝났음을 알림.

sched_setattr() call

```
int sched_setattr(struct task_struct *p, const struct sched_attr *attr)
{
        printk(KERN_DEBUG "sched_setattr call");
        return __sched_setscheduler(p, attr, true, true);
}
```

__sched_setscheduler()의 반환값을 반환하는 함수
인자로 PCB, 스케줄링 정보가 담긴 **attr 구조체를** 넘긴다.

__sched_setscheduler() call

```
static int __sched_setscheduler(struct task_struct *p,
                                const struct sched_attr *attr,
                                bool user, bool pi)
{
        int newprio = dl_policy(attr->sched_policy) ? MAX_DL_PRIO - 1 :
                      MAX_RT_PRIO - 1 - attr->sched_priority;
        int retval, oldprio, oldpolicy = -1, queued, running;
        int new_effective_prio, policy = attr->sched_policy;
        const struct sched_class *prev_class;
        struct rq_flags rf;
        int reset_on_fork;
        int queue_flags = DEQUEUE_SAVE | DEQUEUE_MOVE | DEQUEUE_NOCLOCK;
        struct rq *rq;
        printk(KERN_DEBUG "__sched_setscheduler call");
```

```
        __setscheduler(rq, p, attr, pi);
```

rq에는 현재 런큐의 상태를 표현
라운드 로빈을 사용할 프로세스의 경우 struct rt_rq 필드에 프로세스가 큐잉 됨.
이 보고서에서는 한 개의 코어를 사용하기 때문에 rt_rq에만 프로세스 큐잉
__setscheduler함수에 인자로 rq, PCB, attr, pi를 넘김.

__setscheduler() call

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

        /*
         * Keep a potential priority boosting if called from
         * sched_setscheduler().
         */
        p->prio = normal_prio(p);
        if (keep_boost)
                p->prio = rt_effective_prio(p, p->prio);

        if (dl_prio(p->prio))
                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;
}
```

task_struct의 prio 필드에 따라 스케줄러 클래스를 등록.
프로세스의 우선순위에 따라서 스케줄러 클래스를 등록. 라운드 로빈의 경우 rt_prio(p->prio)에서 1이 반환되어 PCB에 rt_sched_class(라운드 로빈) 등록.

__setscheduler_params call

(프로세스에 policy 등록하는 함수)

```
static void __setscheduler_params(struct task_struct *p,
                const struct sched_attr *attr)
{
        int policy = attr->sched_policy;
        printk(KERN_DEBUG "__setscheduler_params call");
        if (policy == SETPARAM_POLICY)
                policy = p->policy;

        p->policy = policy;

        if (dl_policy(policy))
                __setparam_dl(p, attr);
        else if (fair_policy(policy))
                p->static_prio = NICE_TO_PRIO(attr->sched_nice);

        /*
         * __sched_setscheduler() ensures attr->sched_priority == 0 when
         * !rt_policy. Always setting this ensures that things like
         * getparam()/getattr() don't report silly values for !rt tasks.
         */
        p->rt_priority = attr->sched_priority;
        p->normal_prio = normal_prio(p);
        set_load_weight(p, true);
}
```

task_struct(PCB)의 policy 변수에 cpu.c에서 기록한 policy, priority 저장.

set_load_weight() 함수는 프로세스가 가지고 있는 우선순위를 load weight로 변환.

스케줄러의 관리

(kernel/sched/sched.h)

struct sched_class

```
struct sched_class {
        const struct sched_class *next;

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*yield_task)   (struct rq *rq);
        bool (*yield_to_task)(struct rq *rq, struct task_struct *p, bool preempt);

        void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);

        /*
         * It is the responsibility of the pick_next_task() method that will
         * return the next task to call put_prev_task() on the @prev task or
         * something equivalent.
         *
         * May return RETRY_TASK when it finds a higher prio class has runnable
         * tasks.
         */
        struct task_struct * (*pick_next_task)(struct rq *rq,
                                               struct task_struct *prev,
                                               struct rq_flags *rf);
        void (*put_prev_task)(struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
        int  (*select_task_rq)(struct task_struct *p, int task_cpu, int sd_flag, int flags);
        void (*migrate_task_rq)(struct task_struct *p);

        void (*task_woken)(struct rq *this_rq, struct task_struct *task);
```

이 스케줄링 클래스는 총 5가지 스케줄러(stop, deadline, rt, cfs, idle)의 세부 동작을 모듈화 한 클래스이다.

(kernel/sched/rt.c)

rt class 스케줄러의 선언

```
const struct sched_class rt_sched_class = {
        .next                   = &fair_sched_class,
        .enqueue_task           = enqueue_task_rt,
        .dequeue_task           = dequeue_task_rt,
        .yield_task             = yield_task_rt,

        .check_preempt_curr     = check_preempt_curr_rt,

        .pick_next_task         = pick_next_task_rt,
        .put_prev_task          = put_prev_task_rt,

#ifdef CONFIG_SMP
        .select_task_rq         = select_task_rq_rt,

        .set_cpus_allowed       = set_cpus_allowed_common,
        .rq_online              = rq_online_rt,
        .rq_offline             = rq_offline_rt,
        .task_woken             = task_woken_rt,
        .switched_from          = switched_from_rt,
#endif

        .set_curr_task          = set_curr_task_rt,
        .task_tick              = task_tick_rt,

        .get_rr_interval        = get_rr_interval_rt,

        .prio_changed           = prio_changed_rt,
        .switched_to            = switched_to_rt,

        .update_curr            = update_curr_rt,
};

#ifdef CONFIG_RT_GROUP_SCHED
```

- sched_class 구조체 필드에 RT스케줄러 세부 함수를 선언.(모듈화)
- 직접 스케줄링 클래스를 만들기 위해선 kernel/sched에 내가 만들고 싶은 스케줄러 my_sched.c를 만들고 위와 같이 스케줄링 클래스를 정의해야함.

enqueue_task:프로세스가 실행 가능한 상태로 진입

dequeue_task:프로세스가 더 이상 실행 가능한 상태가 아닐때
yield_task:프로세스가 스스로 yield() 시스템콜을 실행했을 때
check_preempt_curr:현재 실행 중인 프로세스를 선점(preempt)할 수 있는지 검사
pick_next_task:실행할 다음 프로세스를 선택
put_prev_task:실행중인 태스크를 다시 내부 자료구조에 큐잉
load_balance:코어 스케줄러가 태스크 부하를 분산하고자 할때
set_curr_task:태스크의 스케줄링 클래스나 태스크 그룹을 바꿀때
task_tick:타이머 틱 함수가 호출

정리하자면 sched_setattr을 호출했을 때 스케줄러를 변경할 때 스케줄러 클래스의 정보를 담고있는 PCB의 프로세스의 우선순위에 따라서 스케줄러 클래스를 변경한다.

• 특히 어려웠던 점 및 해결 방법 (1장 이내)

①printk 수행했을 때 문제점

include/asm-generic/uaccess.h의 __access_ok()
include/linux/rcupdate.h의 rcu_read_lock(), rcu_read_unlock()

위의 함수들을 printk 찍어보았습니다. 지금 보면 당연히 문제가 있던 것입니다.
커널이 부팅될 때 위의 함수들은 주요하게 사용되는 함수들이기 때문에 로그 버퍼가 넘쳐나서 커널이 잘 부팅되지 않는 문제가 발생 할 수 있다는 걸 알게 되었습니다.
인스턴스를 hard reboot하고 부트 메뉴를 보려고 5회 이상 reboot 수행했지만 결국 보지 못했고 교수님께서 도와주셔서 잘 해결할 수 있었습니다.
(커널 패닉된 문제도 해결해주셔서 감사합니다.)

②CLOCK_MONOTONIC일 때 5초에 종료 되어야할 프로세스가 5.3초에 종료

스스로 생각해보았을 때 두가지 문제 중 하나일 거라고 생각했습니다.

① 시간을 측정하는 구현 방법에 문제

만약 구현 방법에 문제가 있다면 코어를 1개로 설정하기 전에 멀티코어 일 때 cpu 수행 했을 때 문제가 똑같이 발생할 거라고 생각했습니다. 하지만 멀티 코어 일 때는 5초를 정확하게 출력했고 구현 방법에는 이상이 없다고 생각했습니다.

② MONOTONIC이기 때문에 문제

CLOOCK_MONOTONIC은 부팅 후로부터의 단조시간이기 때문에 프로세스가 wating queue에서 기다릴 때도 시간이 흐르고 있다고 생각했습니다. 5초를 cpu 연산하라고 지시 했을 때 실제로는 5초에 못 미치는 연산을 수행했을 것이고, wating queue에서 기다리는 시간이 포함된 시간일 것이라고 생각했습니다.
CPUTIME_ID를 사용했을 때는 5초에 정확한 연산을 수행하여 이게 올바른 방식이라고 생각했지만 교수님께서 지적해주셨고 이럴 경우 context switching overhead를 측정할 수 없다고 말씀해주셨습니다.
생각을 해보니 CPUTIME_ID를 사용하게 되면 switcing 할 때의 시간이 포함되지 않았기 때문에 측정할 수 없다는 걸 깨닫게 되었습니다. 그래서 기존의 방식대로 보고서를 작성했습니다.

과제를 하면서 느낀 것은, 누군가에겐 쉬운 개념이고 과제였을 수도 있지만 이해가 잘 되지 않았을 때 시간이 남들보다 오래 걸리더라도 스스로 고민하고, 문제의 해결방안을 찾아보게 되었던 좋은 경험이었습니다.

감사합니다.