시스템 프로그래밍 과제 #1

2016147571 김조현

0. 과제환경

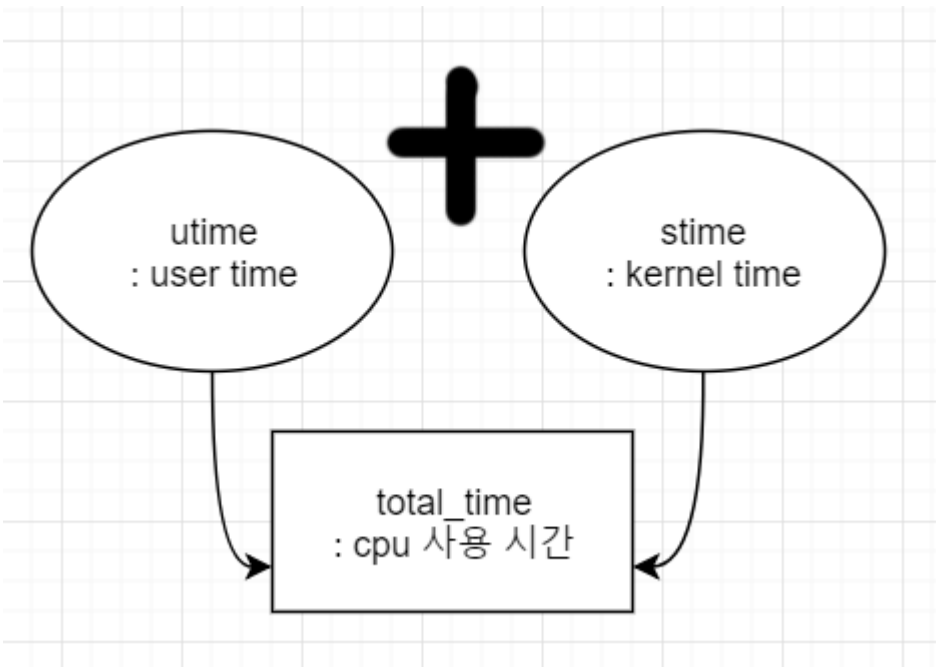
Window, i7-7500U CPU, ram 8GB

```
ubuntu@ubuntu:~/Desktop/hw1/benchmark$ uname -a
Linux ubuntu 2.6.20-15-generic #2 SMP Sun Apr 15 06:17:24 UTC 2007 x86_64 GNU/Linux
```

```
user@user-laptop:~/Desktop/hw1/benchmark$ uname -a
Linux user-laptop 2.6.24-26-generic #1 SMP Tue Dec 1 17:55:03 UTC 2009 x86_64 GNU/Linux
```

1. task_struct 분석

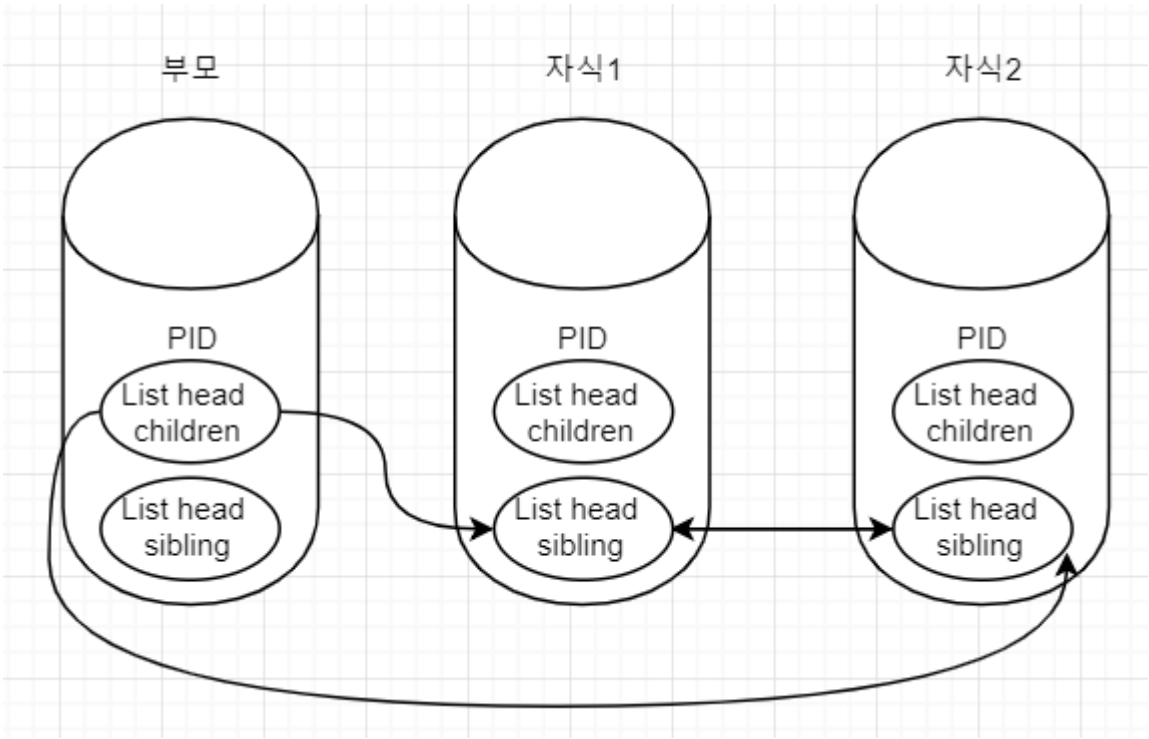
```
unsigned int rt_priority;
cputime_t utime, stime, utimescaled, stimescaled;
cputime_t gtime;
cputime_t prev_utime, prev_stime;
unsigned long nvcsw, nivcsw; /* context switch counts */
struct timespec start_time;         /* monotonic time */
struct timespec real_start_time;    /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
unsigned long min_flt, maj_flt;
```



위 코드에서 **utime** 은 user time으로, 프로세스가 유저모드에서 실행된 tick 횟수를 나타내며, **stime**은 system time으로, 프로세스가 커널모드에서 실행된 tick 횟수를 나타낸다. 커널은 전역적인 jiffies counter를 가지고 있는데, **start_time**은 프로세스가 생성된 시간을 이 jiffies의 값으로 가지고 있다.

```
struct task_struct *real_parent; /* real parent process (when being debugged) */
    struct task_struct *parent;    /* parent process */
    /*
     * children/sibling forms the list of my children plus the
     * tasks I'm ptracing.
     */
    struct list_head children;    /* list of my children */
    struct list_head sibling;    /* linkage in my parent's children list */
    struct task_struct *group_leader;    /* threadgroup leader */
```

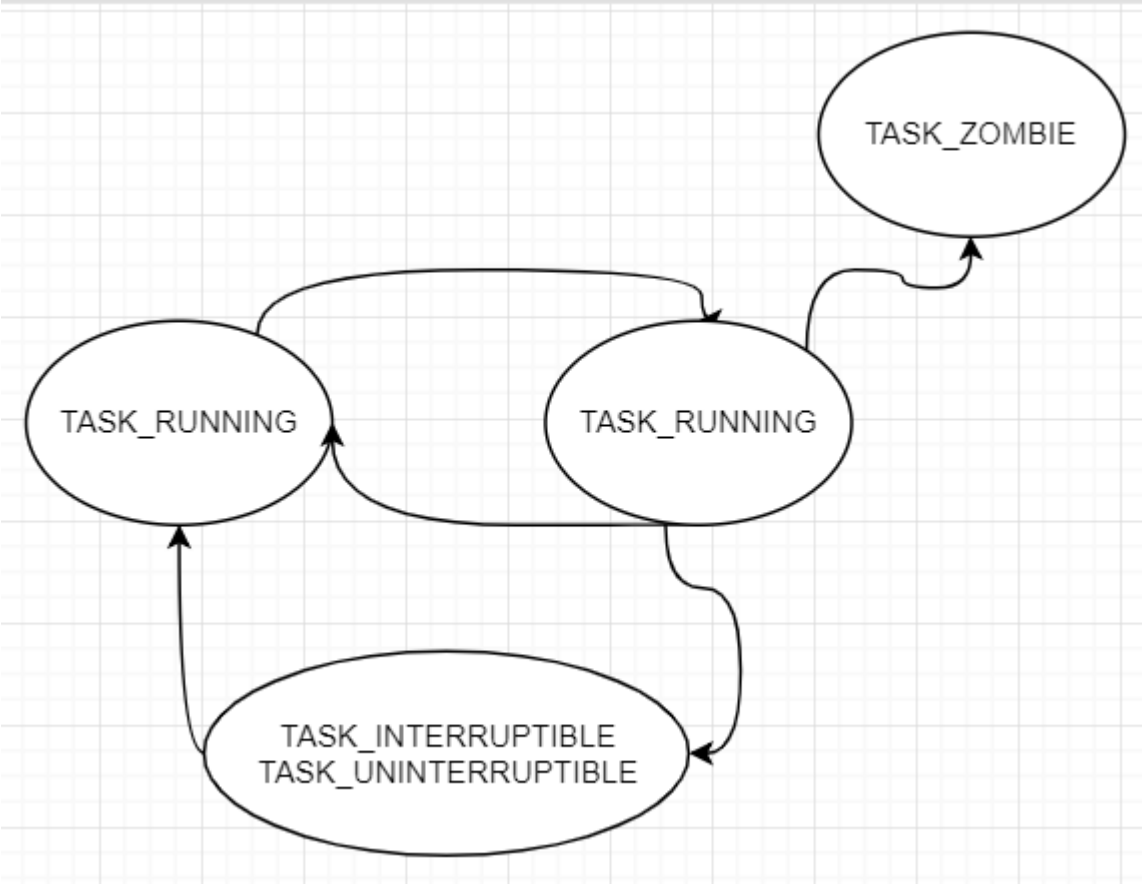
task_struct 구조체는 부모의 task_struct 주소를 parent 로 접근할 수 있게 되어있다. 그리고 children 변수를 통해 자식 프로세스들을 링크드리스트로 연결한 리스트 정보에 접근할 수 있다.



task_struct는 구조체이기 때문에, 부모 프로세스 입장에서 자식 프로세스들이 생길때마다 그것을 변수로 넣어줄 수는 없다. 그렇기 때문에, list_head를 이용해서 형제들끼리 연결되어있고, 부모자식간에도 연결이 되어있어, list_head children list_head sibling 등으로 접근이 가능하다.

```
volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
void *stack;
atomic_t usage;
unsigned int flags;    /* per process flags, defined below */
unsigned int ptrace;
```

각각의 프로세스는 여러가지의 상태를 가지고 있다. 위 코드에서 **state**는 프로세스의 생성에서부터 소멸까지의 상태를 나타낸다. state 변수는 volatile 형식인것을 확인할 수 있는데, 이것은 외부적인 요인으로 그 값이 언제든지 바뀔 수 있다는 것을 뜻한다.



프로세스 상태들 중 주요한것 위주로 순서도를 그려보았다.

```
// sched.h - 2.6.24
#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE  1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED        4
#define TASK_TRACED     8
/* in tsk->exit_state */
#define EXIT_ZOMBIE     16
#define EXIT_DEAD       32
/* in tsk->state again */
#define TASK_DEAD       64
```

위 코드를 보면 2.6.24(CFS) 버전에서의 sched.h 파일에 정의된, 프로세서들의 상태와 그에 따라 정해져있는 값들을 알 수 있다.

```
// sched.h - 2.6.20
#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE  1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED        4
#define TASK_TRACED     8
/* in tsk->exit_state */
#define EXIT_ZOMBIE     16
#define EXIT_DEAD       32
/* in tsk->state again */
#define TASK_NONINTERACTIVE 64
#define TASK_DEAD       128
```

위 코드를 보면 2.6.20(O(1)) 버전에서의 sched.h 파일에 정의된, 프로세서들의 상태와 그에 따라 정해져있는 값들을 알 수 있다. TASK_NONINTERACTIVE 라는 것이 추가된 것을 확인할 수 있다. 그래서 TASK_RUNNING일때 state의 값은 0이 되고 0보다 큰 다른 값들의 경우에는 state의 값이 0보다 크기 때문에 실행되는 상태가 아니게 된다.

```
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next){
    ...
        switch_to(prev, next, prev);
    ...
}
```

sched.c 코드에서 **context_switch** 함수의 모습이다. 이 함수에서는 task_struct 구조체가 prev와 next라는 이름으로 두번 쓰인다. 이 context_switch 함수를 쓰게 되면 그 내부에 **switch_to**라는 매크로에도 도달하게 된다. switch_to 매크로는 CPU의 현재 상태를 이전 커널 스택에 저장하고, esp와 eip를 새로운 프로세스의 것으로 갱신하고 새로운 프로세스를 준비시킨다.

```
asmlinkage void __sched schedule(void)
{
    struct task_struct *prev, *next;
    long *switch_count;
    struct rq *rq;
    int cpu;
}
```

schedule() 함수는 리눅스 커널의 스케쥴링을 담당한다. 위 함수는 현재 실행중인 프로세스의 need_resched가 1이 되고 프로세스가 system call에서 돌아오고 잇을때나, 프로세스의 counter이 0이고 10msec의 타이머가 모두 사용되었을때, 또는 현재 실행중인 프로세스가 중지되었을때 실행된다.

2. 스케줄러 코드 분석

앞서 말했듯이, 리눅스 커널의 스케쥴링을 담당하는 메인 함수는 schedule() 함수이고, 이 함수는 source/kernel/sched.c 파일 안에서 찾을 수 있었다.

2.1 O(n) 스케쥴러

```
struct schedule_data * sched_data;
struct task_struct *prev, *next, *p;
struct list_head *tmp;
int this_cpu, c;
```

prev는 현재 실행중인 프로세스, next는 다음에 실행될 프로세스이다. tmp는 list_head 구조체로서 리스트 하나를 만들었다고 생각하면 된다. sched_data는 schedule_data 구조체이다.

```
spin_lock_prefetch(&runqueue_lock);
BUG_ON(!current->active_mm)
```

sched.h 파일을 참고해보면 active_mm은 mm_struct 구조체임을 알 수 있었고, active_mm은 현재 실행 가능한 프로세스들이 담겨있는 리스트이고, 만약 current(현재 프로세스)가 이 리스트에 없다면 버그가 발생한 것이므로 BUG_ON 함수를 통해 이를 알 수 있다.

```
need_resched_back:
    prev = current;
    this_cpu = prev->processor;
    if(unlikely(in_interrupt())){
        printk("Scheduling in interrupt\n");
        BUG();
    }
    release_kernel_lock(prev, this_cpu);
    sched_data = & aligned_data[this_cpu].schedule_data;
    spin_lock_irq(&runqueue_lock);
```

need_resched_back 부분은 지금 스케쥴링을 다시 해야하는 경우에 실행되게 된다.

먼저 prev에 current를 넣고, 정수형인 this_cpu 변수에는 prev 구조체에 있는 processor 값을 넣게 된다.

release_kernel_lock()함수를 통해서는 프로세스의 모든 커널들의 잠금을 풀어준다.

```
    // move an exhausted RR process to be last
    if(unlikely(prev->policy == SCHED_RR))
        if(!prev->counter){
            prev->counter = NICE_TO_TICKS(prev->nice);
            move_last_runqueue(prev);
        }
```

if(unlikely(prev->policy == SCHED_RR)) 부분부터는 time quantum을 모두 소진한 프로세스들을 runqueue의 맨 마지막 부분으로 넣어주는 코드들이 있는 것을 확인할 수 있다. 현재 task가 SCHED_RR 정책을 가진 real time task인 경우, counter의 값이 0이 되었다면 타임퀀텀을 새로 할당한 후 runqueue의 맨 마지막으로 추가해준다.

```
    switch(prev->state){
        case TASK_INTERRUPTIBLE:
            if(signal_pendding(prev)){
                prev->state = TASK_RUNNING;
                break;
            }
        default:
            del_from_runqueue(prev);
        case TASK_RUNNING:;
    }
    prev->need_resched = 0;
```

그리고 prev의 상태는 state변수를 통해 볼 수 있는데, 이 변수의 값은 위에서 본것처럼 0부터 64까지 TASK_INTERRUPTIBLE, TASK_RUNNING등의 상태로 나뉜다. 그리고 prev 변수가 TASK_INTERRUPTIBLE 상태인 경우에는, 만약 task에 블록되지 않은 ready상태의 프로세스가 있다면 그것을 TASK_RUNNING으로 바꾼다. 이러한 case가 아니라 default로 들어가는 경우에는 실행큐에서 실행중이던 프로세스를 제거한다.

그리고 prev 프로세스가 스케쥴링이 필요한지 묻는 prev의 need_resched 변수를 0으로 만들어주어 더이상 스케줄링이 필요하지 않음을 알린다.

```
repeat_schedule:
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head){
        p = list_entry(tmp, struct task_struct, run_list);
        if(can_schedule(p, this_cpu)){
            int weight = goodness(p, this_cpu, prev_active_mm);
            if(weight > c)
                c = weight, next = p;
        }
    }
```

next 변수에는 runqueue의 첫번째 항목이 담기게 되고, 이를 통해 runqueue를 탐색할 수 있다. 먼저 c변수를 -1000으로 초기화 해준 후, list_for_each()함수를 사용해서 runqueue를 하나씩 본다. 이 함수내에서 먼저 각 프로세스가 실행될 수 있을지를 can_schedule()을 통해 확인하고, 가능한 경우에는 weight 변수에 그 프로세스의 가중치를 두게 된다. 그리고 그 가중치가 본래의 c보다 큰 경우에 next를 갱신하게 된다. 즉 이 함수가 실행이 끝나고 나면 next에는 가중치 값이 가장 큰 프로세스가 담기게 된다.

```
if(unlikely(!c)){
    ...
    goto repeat_schedule;
}
```

만약 counter을 다시 계산해야할 필요가 있는 경우 위의 repeat_schedule 부분으로 돌아가서 다시 런큐에 있는 모든 프로세스의 counter값을 다시 계산한다.

```
sched_data -> curr = next;
task_set_cpu(next, this_cpu);
spin_unlock_irq(&runqueue_lock);
if(unlikely (prev==next)){
    prev->policy&= ~SCHED_YIELD;
    goto same_process
}
```

scheduling data를 담고있는 sched_data의 현재 프로세스를 next로 지정해주고, 현재 프로세스와 다음으로 실행될 프로세스가 같은 경우에 same_process 부분으로 넘어가게 된다.

```
prepare_to_switch();{
    struct mm_struct *mm = next->mm;
    struct mm_struct *oldmm = prev->active_mm;
    if(!mm){
        BUG_ON(next->active_mm);
        next->active_mm = oldmm;
        atomic_inc(&oldmm->mm_count);
        enter_lazy_tlb(oldmm, next, this_cpu);
    } else{
        BUG_ON(next->active_mm != mm);
        switch_mm(oldmm, mm, next, this_cpu);
    }
    if(!prev->mm){
        prev ->active_mm = NULL;
        mmdrop(oldmm);
    }
}
```

위의 prepare_to_switch 함수를 통해 프로세스에 대한 메모리 주소공간을 switch 하게 된다. switch_mm() 함수를 이용해서 두 프로세스의 메모리 주소공간을 switch 해준다..

```
switch_to(prev, next, prev);
__schedule_tail(prev);
```

그리고 위의 코드를 이용해서 프로세스의 레지스터 상태와 스택을 스위칭한다.

2.2 O(1) 스케쥴러

```
struct task_struct *prev, *next;
struct prio_array *array;
struct list_head *queue;
unsigned long long now;
unsigned long run_time;
int cpu, idx, new_prio;
long *switch_count;

struct rq *rq;
```

O(n)과 마찬가지로 prev, next라는 task_struct 구조체가 존재한다. 그리고 우선순위 큐인 array 변수가 존재한다. cpu는 프로세스의 번호를 받기 위한 변수이다.

```
need_resched:
    preempt_disable();
    prev = current;
    release_kernel_lock(prev);
```

need_resched는 다시 스케쥴링이 필요할때 들어오게 되는 부분이다. 이러한 경우에 스케쥴링 요청이 또 오는 것을 막기 위해 preempt_disable()으로 스케쥴링 요청을 금지해준다.. 현재 태스크를 의미하는 current 변수는 prev에 저장한다. release_kernel_lock() 함수를 통해 prev 프로세스에 있는 모든 커널의 잠금을 해제해준다.

```
need_resched_nonpreemptible:
    rq = this_rq();

    if (unlikely(prev == rq->idle) && prev->state != TASK_RUNNING) {
        printk(KERN_ERR "bad: scheduling from the idle thread!\n");
        dump_stack();
    }

    schedstat_inc(rq, sched_cnt);
    now = sched_clock();
```

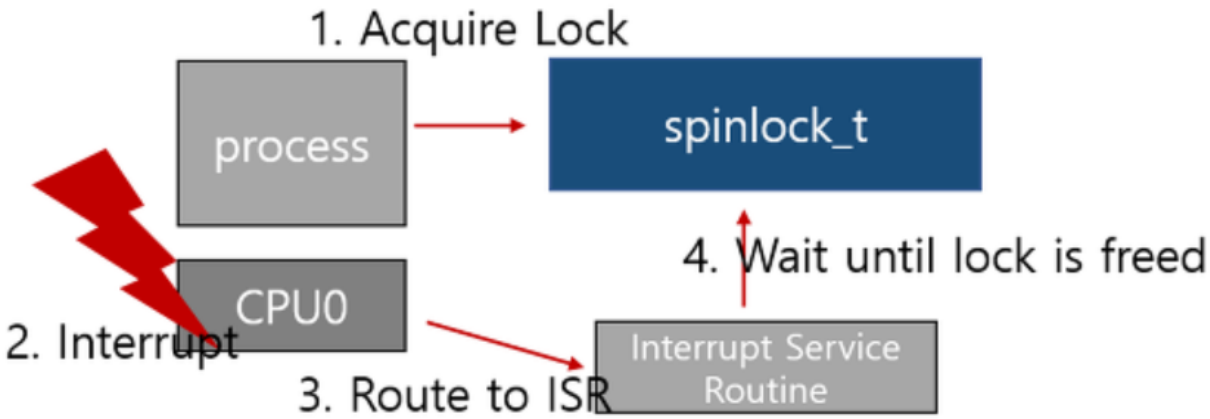
비 선점형으로 스케쥴링이 될때는 위 코드부분으로 들어오게 된다. rq는 이 프로세스 스케줄러의 실행 큐이다. 만약 rq의 idle이 이전 프로세스와 같고, 이전 프로세스의 상태가 TASK_RUNNING이 아니라면, 에러메시지를 내보낸다.

```
    if (likely((long long)(now - prev->timestamp) < NS_MAX_SLEEP_AVG)) {
        run_time = now - prev->timestamp;
        if (unlikely((long long)(now - prev->timestamp) < 0))
            run_time = 0;
    } else
        run_time = NS_MAX_SLEEP_AVG;
```

만약 now - (이전 프로세스의 타임스탬프 값)이 NS_MAX_SLEEP_AVG보다 작다면, run_time은 now-prev->timestamp 값이 되며, 혹시라도 그 값이 0보다 작다면 runtime은 0으로 설정된다. 그렇지 않은 경우에는 run_time 값이 NS_MAX_SLEEP_AVG 값을 run_time값으로 설정해 준다.

```
    run_time /= (CURRENT_BONUS(prev) ? : 1);
    spin_lock_irq(&rq->lock);
    switch_count = &prev->nivcsw;
```

spin_lock_irq 함수를 이용해서 run queue에 lock을 설정하여 동기화 문제를 해결한다.



위 그림은 spinlock에 대해 더 알아보고자 찾아보던 중, 보게 된 그림이다. 예시를 들어서 A 프로세스가 값을 수정하기 이전에 B 프로세스가 읽는 것을 막기 위하여 spin_lock을 사용한다고 한다. A 프로세스는 spin_lock을 걸어두면 되는데, B 프로세스는 그동안 busy waiting 상태로 기다린다. A가 수정작업이 모두 끝나면 spin_lock을 해제하고, 이후에 B 프로세스는 lock을 획득하여 읽을 수 있게 된다.

```
//O(1)참고-1
if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
    switch_count = &prev->nvcsw;
    if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
        unlikely(signal_pending(prev))))
        prev->state = TASK_RUNNING;
    else {
        if (prev->state == TASK_UNINTERRUPTIBLE)
            rq->nr_uninterruptible++;
        deactivate_task(prev, rq);
    }
}
```

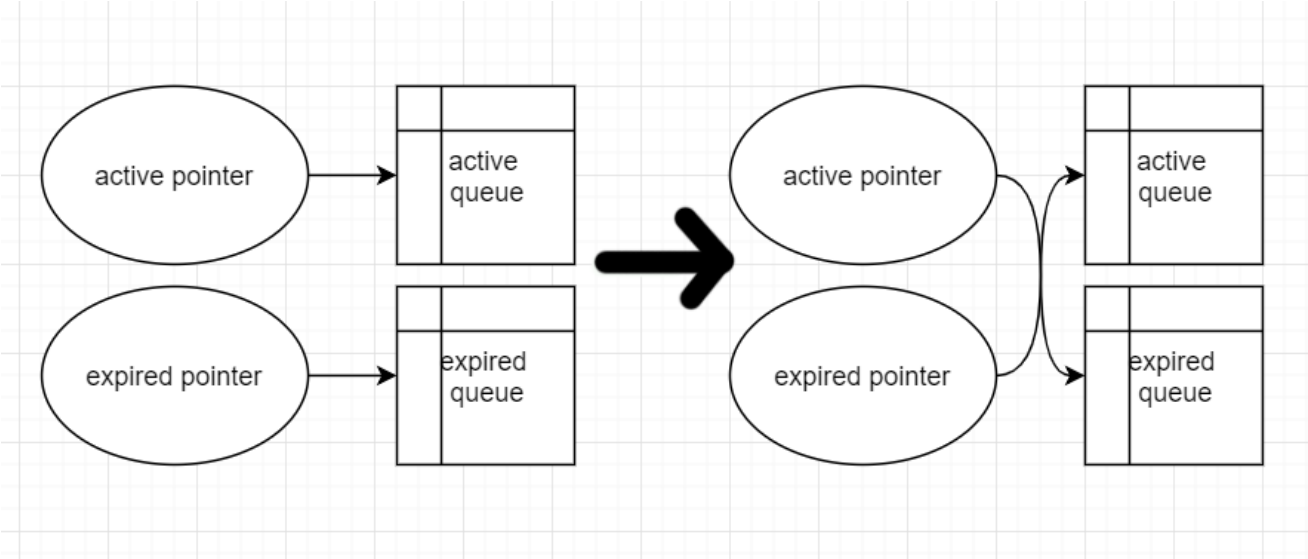
두번째 줄을 보면, 만약 현재 프로세스의 상태가 TASK_INTERRUPTIBLE이며 시그널을 수신한 경우라면, 현재 프로세스 prev의 상태는 TASK_RUNNING이 된다. 그렇지 않은 경우에는 deactivate_task() 함수를 사용해서 prev라는 현재 실행중이었던 프로세스를 run queue에서 제거해준다.

```
cpu = smp_processor_id();
    if (unlikely(!rq->nr_running)) {
        idle_balance(cpu, rq);
        if (!rq->nr_running) {
            next = rq->idle;
            rq->expired_timestamp = 0;
            wake_sleeping_dependent(cpu);
            goto switch_tasks;
        }
    }
```

smp_processor_id()에 있는 현재 실행중인 프로세스의 번호를 cpu 변수에 넘겨준다. 만약 runqueue의 nr_running이 Flase라면(rq내에 실행 가능한 프로세스가 없는 경우), idle_balance() 함수를 사용해서 다른 프로세스 스케쥴러의 run queue에서 프로세스를 가져온다. 그리고 switch_tasks로 넘어가게 된다.

```
array = rq->active;
    if (unlikely(!array->nr_active)) {
        schedstat_inc(rq, sched_switch);
        rq->active = rq->expired;
        rq->expired = array;
        array = rq->active;
        rq->expired_timestamp = 0;
        rq->best_expired_prio = MAX_PRIO;
    }
```

위 코드를 그림으로 표현해 보았다.

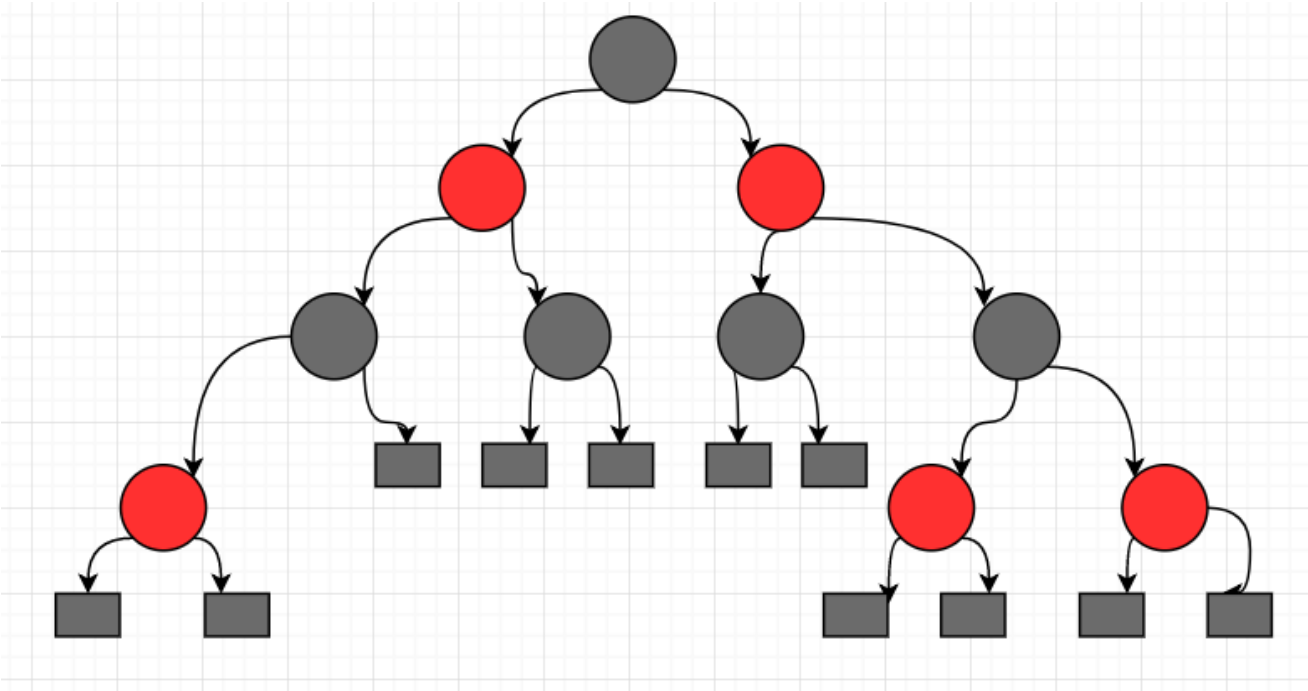


먼저 array 변수에 runqueue 중 active를 넣어둔다. 강의자료를 참고해 보면, O(1) 스케쥴러에서는 runqueue 구조체 안에 prio_array_t라는 자료구조를 가진 객체 active, expired를 가진다. 그리고 O(1) 스케쥴러의 경우에 active에 있는 모든 작업을 처리하면 active와 expired를 바꾼다. 위의 if문은 active안에 실행가능한 프로세스가 없는 경우에 들어가게 된다. 그런 경우에 active는 expired가 되고, expired는 active가 된다. 이것은 포인터를 바꿈으로서 가능하다.

```
    idx = sched_find_first_bit(array->bitmap);
    queue = array->queue + idx;
    next = list_entry(queue->next, struct task_struct, run_list);

    if (!rt_task(next) && interactive_sleep(next->sleep_type)) {
        unsigned long long delta = now - next->timestamp;
        if (unlikely((long long)(now - next->timestamp) < 0))
            delta = 0;

        if (next->sleep_type == SLEEP_INTERACTIVE)
            delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;
```

```
        array = next->array;
        new_prio = recalc_task_prio(next, next->timestamp + delta);

        if (unlikely(next->prio != new_prio)) {
            dequeue_task(next, array);
            next->prio = new_prio;
            enqueue_task(next, array);
        }
    }
    next->sleep_type = SLEEP_NORMAL;
    if (dependent_sleeper(cpu, rq, next))
        next = rq->idle;
```

2.3 CFS 스케줄러



CFS 스케쥴러는 레드블랙트리를 기반으로 한다.

```
struct task_struct *prev, *next;
    long *switch_count;
    struct rq *rq;
    int cpu;
```

위와 마찬가지로, prev는 현재 실행중인 프로세스, 즉 이제 다시 스케쥴링 될 프로세스를 말하고, next는 이제 다음으로 실행시킬 프로세스를 의미한다. rq또한 위와 마찬가지로 run-queue 즉 실행 큐를 의미한다. cpu 또한 마찬가지로 프로세스의 번호를 받을 변수이다.

```
need_resched:
    preempt_disable();
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    rcu_qsctr_inc(cpu);
    prev = rq->curr;
    switch_count = &prev->nivcsw;
    release_kernel_lock(prev);
```

스케쥴링이 필요한 경우에 need_resched 안으로 들어오게 된다. 위와 마찬가지로 preempt_disable()을 통해 스케쥴링 요청을 금지해준다. 현재 실행되고 있는 프로세스의 번호를 cpu 변수에 넣어준다. 그리고 rq에 cpu에서 실행중인 프로세스들의 목록을 넘겨준다. prev변수를 rq 안의 curr으로 바꿔주고, switch_count에는 현재 실행중인 prev 프로세스의 nivcsw를 넘겨준다. release_kernel_lock() 함수는 프로세스가 가진 모든 커널의 잠금을 해제하는 역할을 한다.

```
need_resched_nonpreemptible:
    schedule_debug(prev);
    local_irq_disable();
    __update_rq_clock(rq);
    spin_lock(&rq->lock);
    clear_tsk_need_resched(prev);
```

need_resched_nonpreemptible에 들어오게 되면, 먼저 __update_rq_clock() 함수를 통해 실행되고 있는 rq의 time quantum을 업데이트 해준다. 그리고 spin_lock() 함수를 통해 rq->lock에 접근한다.
clear_tsk_need_resched() 함수를 호출하면, 이제 프로세스가 스케쥴링이 되어야 한다는 flag를 1에서 0으로 바꿔주게 된다.

```
if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
                unlikely(signal_pending(prev)))) {
            prev->state = TASK_RUNNING;
        } else {
            deactivate_task(rq, prev, 1);
        }
        switch_count = &prev->nvcsw;
    }
```

위 코드는 **O(1)참고-1** 부분과 동일하다.

```
if (unlikely(!rq->nr_running))
    idle_balance(cpu, rq);

prev->sched_class->put_prev_task(rq, prev);
next = pick_next_task(rq, prev);
sched_info_switch(prev, next);
```

만약 runqueue의 nr_running이 Flase라면(rq내에 실행 가능한 프로세스가 없는 경우), idle_balance() 함수를 사용해서 다른 프로세스 스케쥴러의 run queue에서 프로세스를 가져온다.

prev의 sched_class에 put_prev_task 함수를 사용해서 rq와 prev 를 넣고, next 변수에 pick_next_task 함수를 통해 반환되는 다음 프로세스에 대한 정보를 넣어준다. 그리고 sched_info_switch() 함수를 통해 현재 실행중이었던 프로세스와 다음 프로세스로 정해진 next의 정보를 바꿔준다

```
if (likely(prev != next)) {
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        context_switch(rq, prev, next); /* unlocks the rq */
    } else
        spin_unlock_irq(&rq->lock);
```

prev가 next와 같지 않으면, 즉 다음으로 실행될 다른 프로세스가 next 변수에 잘 설정되었다면, rq의 curr, 즉 이제 실행될 프로세스를 next로 바꾸어준다. 또한 context_switch() 함수를 이용해서 prev 의 context를 next로 바꾸어준다. 만약 prev와 next가 같다면, spinlock을 했던 것을 없애준다.

```
preempt_enable_no_resched();
if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
    goto need_resched;
```

preempt_enable_no_resched() 함수를 통해서 스케쥴링이 일어나는 동안에 스케줄링 요청이 또 들어오지 않도록 해준다. 그리고 TIF_NEED_RESCHED 을 확인한 다음 1인 경우에는 다시 스케쥴링이 필요한 것이므로 need_resched 부분으로 가서 스케쥴링을 다시 시작하게 된다.

3. 스케줄러 비교

O(1) 스케줄러와 CFS의 성능 비교를 위해 5가지의 벤치마크 프로그램을 작성하여 사용하였다. 작성한 프로그램 목록은 다음과 같다

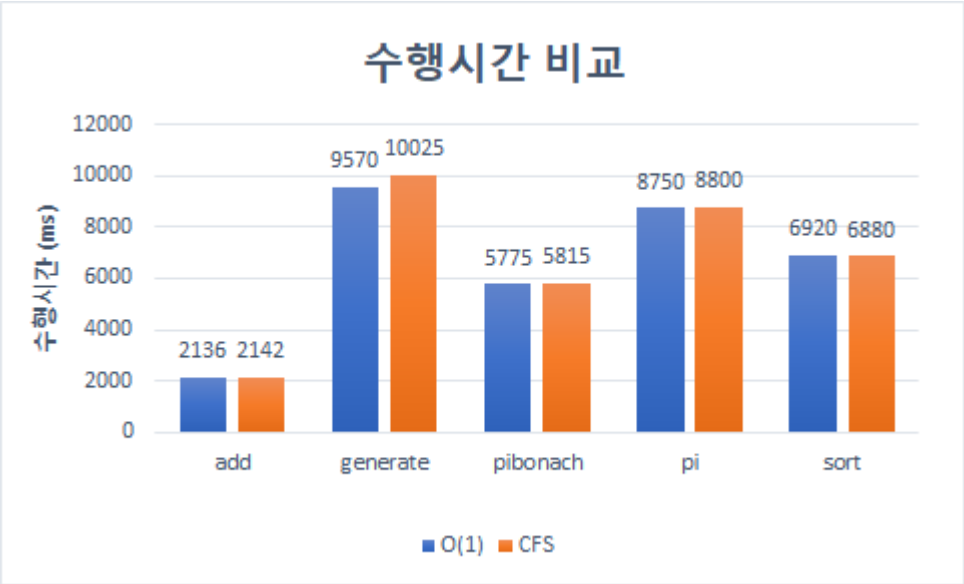
- add.c : 1을 계속 더하는 프로그램
- generate.c : 임의의 수를 만들어 내는 프로그램
- pibonach.c : 피보나치 수열을 일정 길이까지 발생시키는 프로그램
- pi.c : 파이 소수점 자리를 일정 길이까지 계산하는 프로그램
- sort.c : 위의 generate.c를 통해 만들어진 수들을 정렬하는 프로그램

위의 다섯가지 프로그램을 만들어서 성능 비교를 하였고, 각 프로그램은 발생 개수나 범위등을 조정하여 1-10초 사이에 실행이 끝나도록 해두었다.

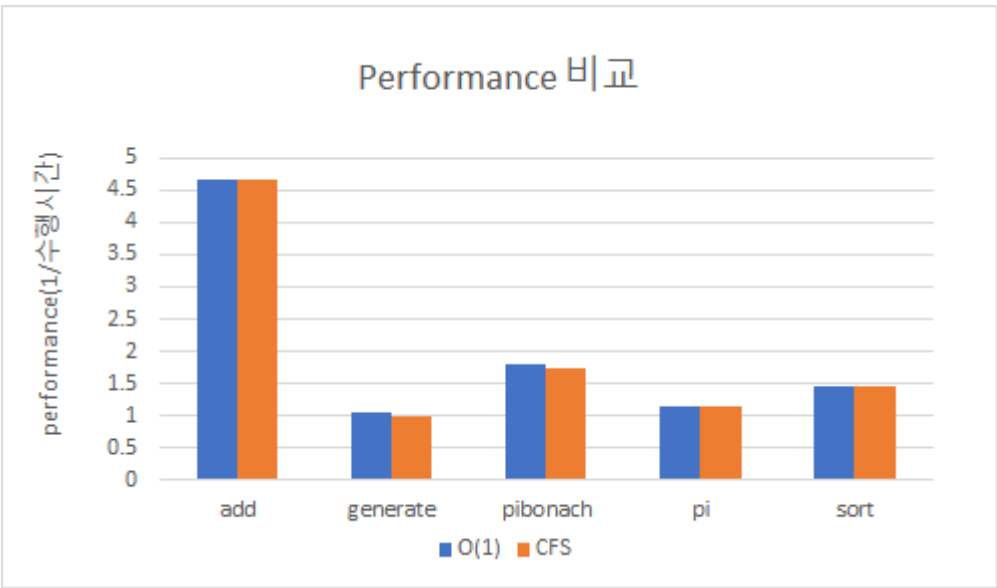
3.1 Performance 비교

	add	generate	pibonach	pi	sort
O(1)	2136	9570	5775	8750	6920
CFS	2142	10025	5815	8800	6880

위 표는 O(1)스케쥴러와 CFS 스케쥴러가 5가지의 벤치마크 프로그램을 돌렸을때 걸린 시간을 ms 단위로 나타낸 표이다.



도표로 나타내면 위와 같다.



이 도표는 걸린 수행시간으로 (ms단위) 10000을 나눴을때의 몫을 가지고 그린 그래프이다. 이 그래프를 보면 성능이 O(1)이 근소한 차이로 더 좋은 것을 확인할 수 있다. 파랑색 바가 O(1)인데, 수행시간 비교 그래프를 보면 앞에서 4개까지의 벤치마크 프로그램의 경우 O(1) 스케쥴러가 더 빨랐고, 단 하나 sort.c 프로그램에서만 CFS 스케줄러가 더 빠른것을 확인할 수 있었다.

red-black tree를 사용하여 스케쥴링하는 CFS는 O(logn)정도로 시간 자체는 조금 더 든다. 그래서 위의 그래프에서도 O(1)이 수행시간은 더 적게들고 빠른 것을 알 수 있다. 그러나 그 차이가 무척 근소하기 때문에 이러한 점을 감수하고도, 공정하게 처리하는것에 중점을 둔 cfs가 쓰이고 있다.

3.2 Fairness 비교

fairness 비교를 하는데에는 pi.c와 sort.c 두가지 파일을 사용하였다. 몇십초간 지속될 정도의 규모로 파일을 수정하였고, 각각 pi_long.o, sort_long.o의 실행파일을 만든 다음, 실행을 시켜둔 상태에서 top을 통해 proc filesystem 내의 정보를 읽어왔다

```
top -b -d 1 -S -p 12414,12415
```

top 명령어와 함께 옵션들을 설정해 주어 분석에 용이하게 했다.

-b를 해주어 갱신되는 정보들이 모두 프린트 되도록 하였고, -d 1을 선언해서 1초 간격으로 정보를 수집하도록 하였으며 -S로 cpu 사용시간이 누적되도록 설정해주었다. -p로는 보고싶은 프로세스들의 pid를 설정해서 볼 수 있었는데 12414는 sort_long의 pid, 12415는 pi_long의 pid였다.

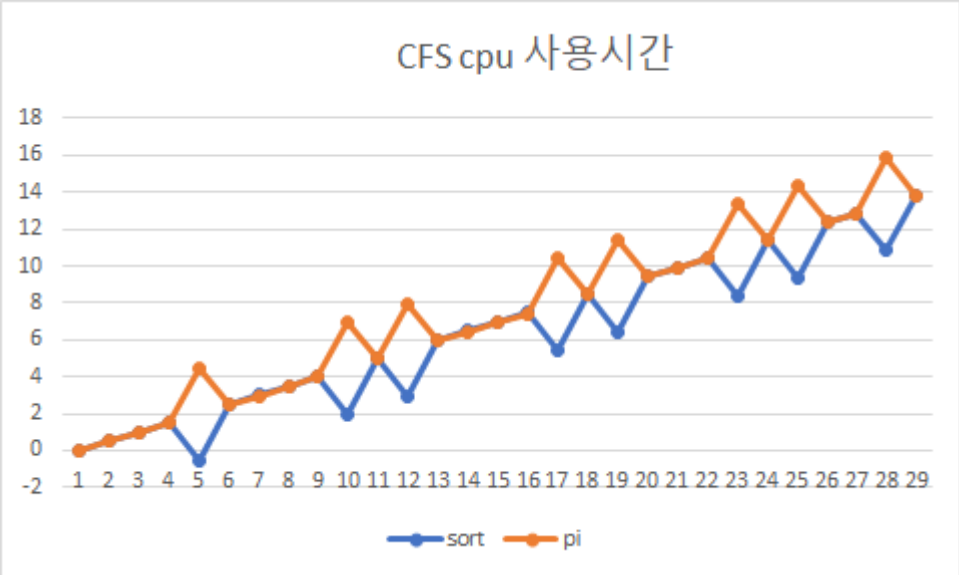
```
Mem:   1029120k total,   642120k used,   387000k free,    82500k buffers
Swap:   489940k total,        0k used,   489940k free,   301200k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12415 user      20   0  9116 5836  288 R 50.0  0.6   0:46.69 pi_long
12414 user      20   0  5600 2320  288 R 49.0  0.2   0:49.17 sort_long


top - 17:54:59 up  3:59,  4 users,  load average: 1.95, 1.23, 0.52
Tasks:   2 total,   2 running,   0 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1029120k total,   642120k used,   387000k free,    82500k buffers
Swap:   489940k total,        0k used,   489940k free,   301200k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12414 user      20   0  5600 2320  288 R 50.0  0.2   0:49.67 sort_long
12415 user      20   0  9116 5836  288 R 49.0  0.6   0:47.18 pi_long


top - 17:55:00 up  3:59,  4 users,  load average: 1.95, 1.23, 0.52
Tasks:   2 total,   2 running,   0 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1029120k total,   642120k used,   387000k free,    82500k buffers
Swap:   489940k total,        0k used,   489940k free,   301200k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12415 user      20   0  9116 5836  288 R 50.0  0.6   0:47.68 pi_long
12414 user      20   0  5600 2320  288 R 49.0  0.2   0:50.16 sort_long
```

위와 같이 출력된 결과를 text 파일로 긁어와서 간단한 코딩을 통해 1초마다 사용한 cpu 사용량을 정리할 수 있었다.

	0	1
0	0:31.00	0:28.52
1	0:31.50	0:29.02
2	0:32.00	0:29.50
3	0:32.50	0:30.00
4	0:30.50	0:32.99
5	0:33.49	0:31.00
6	0:33.99	0:31.50
7	0:34.49	0:32.00
8	0:34.99	0:32.49

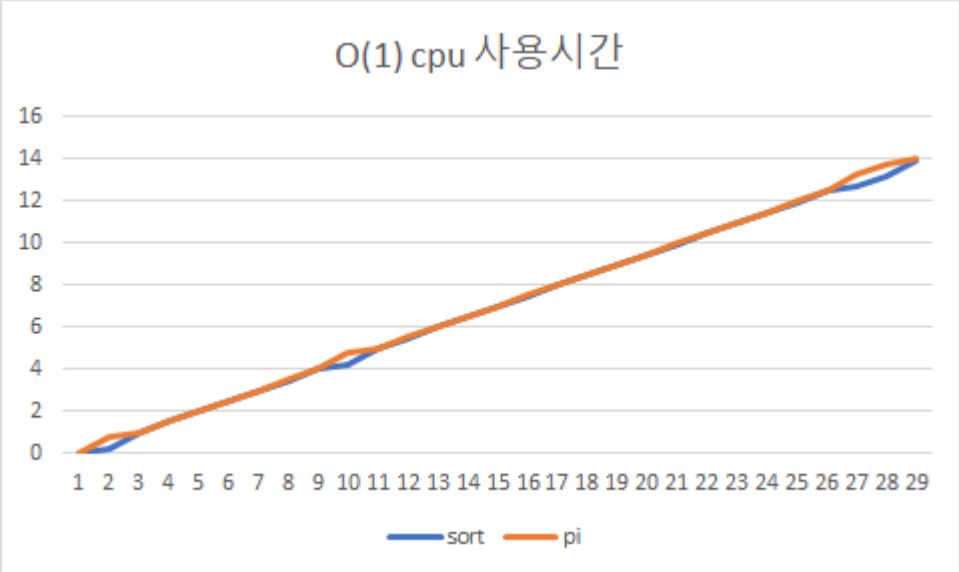
	0	1	sort	pi
0	0:31.00	0:28.52	0.00	0.00
1	0:31.50	0:29.02	0.50	0.50
2	0:32.00	0:29.50	1.00	0.98
3	0:32.50	0:30.00	1.50	1.48
4	0:30.50	0:32.99	-0.50	4.47
5	0:33.49	0:31.00	2.49	2.48
6	0:33.99	0:31.50	2.99	2.98
7	0:34.49	0:32.00	3.49	3.48
8	0:34.99	0:32.49	3.99	3.97

cpu의 사용량을 알기위해 처음 기록된 부분 사용량을 0이라고 가정하고 정리하였다.



그린 그래프는 위와 같다. 파란선은 sort, 주황선은 pi를 실행시켰을때 29초까지 각 초마다 수집된 누적 cpu 사용시간을 나타낸 것이다. CFS의 경우 sort와 pi에 거의 비슷하게 cpu 사용 시간을 부여한것을 확인할 수 있다.

마찬가지 방식으로 O(1) 스케쥴러의 경우에도 top을 통해 proc filesystem 내부의 정보를 얻고, 정리한 후 그래프를 그려보았다.



놀랍게도 오히려 O(1)의 경우에 cpu 사용시간의 분포가 더 고르게 나타났다. 이런 결과가 나온 이유를 추측해 보자면 다음과 같다.

```
  PID USER      PR
12414 user      20
12415 user      20
```

위 그림을 보면, top을 통해 출력되는 정보중에 PR이라는 정보도 있다. 이것은 Priority, 즉 우선순위를 나타낸다. 그런데 이 두 pi 프로그램과 sort 프로그램은 둘다 우선순위값이 20인것을 확인할 수 있다. 이 두 벤치마크 프로그램이 우선순위가 같았기 때문에, 이러한 결과가 나온것으로 해석된다.

O(1) 스케쥴러는 cfs에 비해서 우선순위 차이가 날때 더 크게 cpu 사용시간에 차이를 두게 되는데, 이 두가지 벤치마크 프로그램을 사용했을때 이러한 점을 부각시킬 수 없었기 때문에 이러한 결과가 나온 것 같다. 우선순위가 다른 벤치마크 프로그램을 사용하면 더 명확한 결과를 얻을 수 있었을 것 같다.

4. 커널 모듈 프로그래밍

먼저 기본 코드는 과제 공지사항에 있었던 커널 모듈 프로그래밍 가이드를 보고 작성하였다. Example 5-4 를 보고 작성했으며, 그 이유는 그 코드가 iter라는 이름의 모듈을 만들고 cat /proc/iter을 통해 그 내용을 볼 수 있는 구조로 만들었기 때문이다. 따라서 그부분의 코드를 가져다가 수정해서 코드를 쓰게 되었다.

먼저 모듈에 프린트를 하기 위해서 seq_printf 함수를 썼으며, seq_file *s 변수를 설정해주었다. task_struct 구조체인 task_list를 만들어서 거기서 모든 정보를 출력할 수 있었다.

먼저 for_each_process(task_list)로 프로세스의 개수도 셀 수 있었고, 또한 각 프로세스의 정보를 출력해야할때 사용하였다.

user과 kernel의 시간을 합쳐서 total에 나타내야 했고, 1ms 단위까지 초단위로 출력해야 했다.

```
ts = task_list ->time_slice;
ti1 = tempint2/ ts;
ti2 = task_list->utime /ts;
ti3 = task_list->stime /ts;
ti11 = (tempint2 - ti1*ts)*1000/ts;
ti22 = (task_list->utime - ti2*ts)*1000 /ts;
ti33 = (task_list->stime - ti3*ts)*1000/ts;
```

위 코드는 user, kernel, total을 출력형식에 맞추기 위한 코드이다. seq_printf()함수 내에서 %f를 하면 잘 출력이 되지 않았기 때문에, 반드시 정수를 출력해야만 했다. 그래서 정수부분과 소수부분을 정수로 따로 만들어 준 후, 그 사이에 .을 출력하여 실수처럼 보이도록 해주었다.

ts는 time_slice인데, CFS에서는 250, O(1)에서는 125였다. 이 값이 달랐기 때문에, 양쪽에서 모두 잘 동작하게 하기 위해서 정수를 넣지 않고 task_list->timeslice를 통해 해결하였다.

문제점

O(1)과 CFS 모두 오래 전 스케줄러이기에 그런지, 메일등 통신이 안되어 어려웠다. 그래서 aws ec2 서비스를 이용해서 서버를 파고, pemkey없이 비밀번호로 통신이 가능하도록 설정한 후, 비밀번호를 이용해서 scp로 파일을 보내며 cfs에서 o(1)으로 때로는 host 컴퓨터에서 cfs 가상머신으로 파일을 보내는등을 할 수 있었다.

참고자료

https://libsora.so/posts/system-prog-timing-measurements/

[jiffies] http://www.morenice.kr/120

[volatile] http://skyul.tistory.com/337

[kernel code] https://elixir.bootlin.com/linux/v2.6.24/source/include/linux/sched.h

[context switch] https://libsora.so/posts/system-prog-linux-processes/

[spinlock] http://selfish-developer.com/tag/spin_lock_irq

[커널모듈 프로그래밍] https://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf

[커널모듈 for_each_process] https://gist.github.com/anryko/c8c8788ccf7d553a140a03aba22cab88