시스템 프로그래밍 1차 과제

학과: 컴퓨터과학과
학번: 2015147506
이름: 김기현
제출일자: 2020.10.19.

Contents

I. 사전 조사 보고서

1. 각 버전별 schedule() 함수 동작 과정

1-1) v.2.6.23 Schedule Function

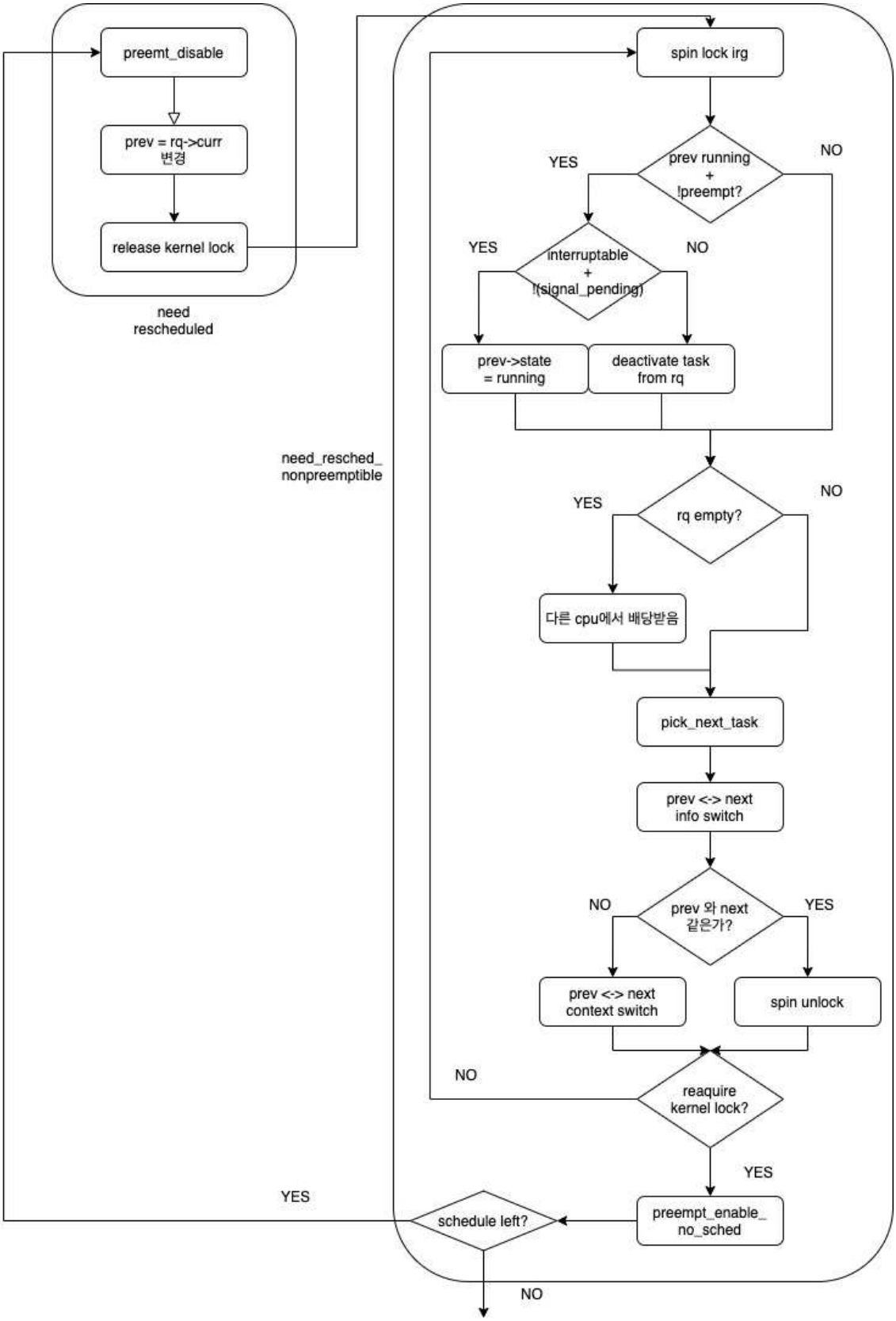
/kernel/sched.c

```
asmlinkagevoid__sched schedule(void)
{
        structtask_struct*prev,*next;
        long*switch_count;
        structrq*rq;
        intcpu;

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);
need_resched_nonpreemptible:

        schedule_debug(prev);

        spin_lock_irq(&rq->lock);
        clear_tsk_need_resched(prev);
        __update_rq_clock(rq);

        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;
        }

        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);

        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);

        if(unlikely(reacquire_kernel_lock(current)<0)){
                cpu=smp_processor_id();
                rq=cpu_rq(cpu);
                goto need_resched_nonpreemptible;
        }
        preempt_enable_no_resched();
        if(unlikely(test_thread_flag(TIF_NEED_RESCHED)))
                goto need_resched;
}
EXPORT_SYMBOL(schedule);
```

2.6.23 버전의 schedule은 다음과 같습니다. 간략하게 설명을 하자면, 2가지 단계로 나뉘는데, need_resched 와, need_resched_nonpreemptible로 나뉩니다. preemptive의 여부로 각각 나뉘는데 간략하게 need_reschedule부터 보면, preempt를 disable 시키고, 이전 prev를 runqueue의 curr로 대체시킵니다. 그후에 prev의 kernel lock을 release 시킵니다. 이후에 바로 need_resched_nonpreemptive로 넘어가는데, 만약에 prev state가 preempt_active일 때, prev state가 task interruptible이 아니라면, 방해할수 없는 상태라면, task running state로 바꿔줍니다. interruptible이라면, task를 변경할 수 있으므로, deactivate task를 합니다. 만약에 runqueue에 할당된 일이 없을 경우에는, 다른 cpu에서 할 일을 idle하게 받아와서 실행을 시킵니다. 그후, 그담에 pick 하는 함수인 pick_next_task를 골라야 하는데, 이는 뒤에 설명하도록 하겠습니다. 만약에 next와 prev가 똑같다면 굳이 switch할 필요는 없으나, 다르다면 context switch를 합니다. 만약에 kernel lock을 다시 풀어주어야 한다면 kernel lock을 이전에 풀어주었던 상기need_resched로 가고, 아니라면, 굳이 풀 필요가 없다면 need_resched_nonpreemptible로 전달해 주면 됩니다. 이러한 과정을 통해서 EXPORT_SYMBOL(schedule)을 통해 schedule을 전역으로 전달 할 수 있게 됩니다.

직관적으로 이해하기 위해 밑의 순서도를 별도 첨부하였습니다.


preemt_disable
prev = rq->curr
변경
release kernel lock
need
rescheduled
need_resched_
nonpreemptible
spin lock irg
prev running
+
!preempt?
YES
NO
interruptable
+
!(signal_pending)
YES
NO
prev->state
= running
deactivate task
from rq
rq empty?
YES
NO
다른 cpu에서 배당받음
pick_next_task
prev <-> next
info switch
prev 와 next
같은가?
NO
YES
prev <-> next
context switch
spin unlock
reaquire
kernel lock?
NO
YES
preempt_enable_
no_sched
schedule left?
YES
NO

1-2) v. 4.19.146 schedule function

/kernel/sched/core.c

```
asmlinkage __visiblevoid__sched schedule(void)
{
        structtask_struct*tsk=current;

        sched_submit_work(tsk);
        do{
                preempt_disable();
                __schedule(false);
                sched_preempt_enable_no_resched();
        }while(need_resched());
}
EXPORT_SYMBOL(schedule);
```

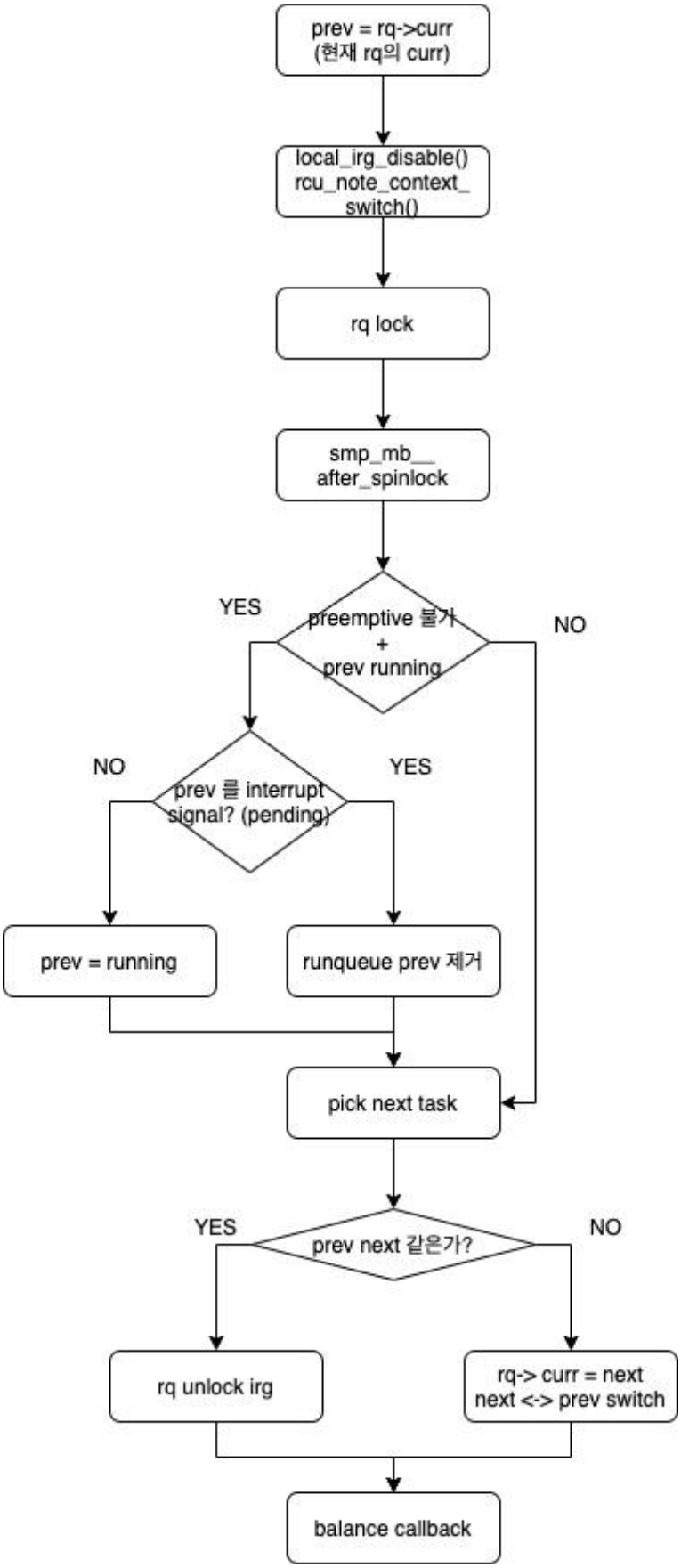
4.19.146 버전에서는 이전 버전과 상당히 함수들이 유사함을 볼 수 있습니다. 같이 preempt_disable함수를 통해 시작을 하지만, __schedule()로 넘어가기 때문에, schedule()함수의 주요 기능을 __schedule()에서 볼 수 있습니다.

```
staticvoid__sched notrace __schedule(bool preempt)
{
        structtask_struct*prev,*next;
        unsignedlong*switch_count;
        structrq_flags rf;
        structrq*rq;
        intcpu;

        cpu=smp_processor_id();
        rq=cpu_rq(cpu);
        prev=rq->curr;

        schedule_debug(prev);

        if(sched_feat(HRTICK))
                hrtick_clear(rq);

        local_irq_disable();
        rcu_note_context_switch(preempt);

        rq_lock(rq,&rf);
        smp_mb__after_spinlock();

        /* Promote REQ to ACT */
        rq->clock_update_flags<<=1;
        update_rq_clock(rq);

        switch_count=&prev->nivcsw;
        if(!preempt&&prev->state){
                if(unlikely(signal_pending_state(prev->state,prev))){
                        prev->state=TASK_RUNNING;
                }else{
                        deactivate_task(rq,prev,DEQUEUE_SLEEP|DEQUEUE_NOCLOCK);
                        prev->on_rq=0;
```

```
                        if(prev->in_iowait){
                                atomic_inc(&rq->nr_iowait);
                                delayacct_blkio_start();
                        }


                        if(prev->flags&PF_WQ_WORKER){
                                structtask_struct*to_wakeup;

                                to_wakeup=wq_worker_sleeping(prev);
                                if(to_wakeup)
                                        try_to_wake_up_local(to_wakeup,&rf);
                        }
                }
                switch_count=&prev->nvcsw;
        }

        next=pick_next_task(rq,prev,&rf);
        clear_tsk_need_resched(prev);
        clear_preempt_need_resched();

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

                ++*switch_count;

                trace_sched_switch(preempt,prev,next);

                /* Also unlocks the rq: */
                rq=context_switch(rq,prev,next,&rf);
        }else{
                rq->clock_update_flags&=~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
                rq_unlock_irq(rq,&rf);
        }

        balance_callback(rq);
}
```

__schedule 함수를 분석해 보면, 이전과 마찬가지로 prev next가 있습니다. 먼저 prev를 runqueue의 cur을 대입해서 시작 합니다. 로컬 irq를 disable함으로서 인터럽트를 막는 것을 방지하게 합니다. runqueue를 lock 시킨 후, 이전 2.6.23버전과 마찬가지로 prev가 preemtive 불가인 상태인지 확인을 하고, 그렇다면, interrupt 가능한지에 대해서 확인합니다. 만약 그렇지 않다면 running state로 지정해 주며, 그렇지 않다면 runqueue에서 prev를 제거합니다. 그 후, pick next task를 검색하며, prev 와 next 상태가 같은지, 그렇다면 runqueue를 unlock을 하고, 그렇지 않다면, runqueue의 curr = next로 지정해 주고, next와 prev를 context swtich를 하는 방식으로 작동합니다. 크게 보면 다른 점이 많이 부각되지는 않지만 switch count에서 nivcsw를 씀과 nvcsw를 쓰는 것과 같은 이름만 바뀐 경우도 있지만,,기능적으로 4.19.146버전에만 있는 local irq를 이용하여 데이터 interrupt 방지할 때 차이점을 보이게 됩니다. 그러나 큰 틀에서는 많이 다르지 않습니다.

직관적인 이해를 위해 순서도를 첨부합니다.


prev = rq->curr
(현재 rq의 curr)
local_irg_disable()
rcu_note_context_
switch()
rq lock
smp_mb__
after_spinlock
YES
preemptive 불가
+
prev running
NO
NO
prev 를 interrupt
signal? (pending)
YES
prev = running
runqueue prev 제거
pick next task
YES
prev next 같은가?
NO
rq unlock irg
rq-> curr = next
next <-> prev switch
balance callback

schedule 함수에서의 중요한 점

1) 프로세스 우선순위 판단

task struct의 state에 따라서, running이냐 blocked이냐의 우선순위에 대해서 정하고, 그 후에는 priority를 따져서 cfs인지 rt인지 따져, 그중에서 cfs면 red black tree의 맨 왼쪽, rt면 그냥 priorty만을 봄으로 우선순위를 정합니다.

2) task 선택 기준

cfs 의 경우 vruntime을 계산해, 가장 작은 것을 red black tree에 맨 왼쪽에 두어 task를 선택합니다. (red black tree란, 각 노드가 red/black 의 색으로 구분되며, 가장 왼쪽의 노드를(높이 상관없는 좌우기준) 가장 작은 값을 가지고 있는 것으로 지정해 놓습니다)

3) runqueue

각각의 schedule에서 runqueue를 가지고 있으며, runqueue에서 또 cfs_rq라는 또다른 runqueuef를 가지고 있어, 각각이 우선순위가 변경되어 task 선택이 변경될 때, runqueue에서 prev next가 변경되는 것을 볼 수 있습니다. 각각의 우선순위가 변경되는 것은 두 버전 모두 preemptive한가? interruptible 한 상태인가?에 대해서 비교를 해본 후에 변경을 가능하게 하여 runqueue를 변경, scheduling에서 변동을 줄 수 있도록 하였습니다.

2. 각 버전 별 CFS 분석

2-1) 2.6.23 버전

/kernel/sched_fair.c (이하 동일)

```
structsched_class fair_sched_class __read_mostly={
        .enqueue_task           =enqueue_task_fair,
        .dequeue_task           =dequeue_task_fair,
        .yield_task             =yield_task_fair,

        .check_preempt_curr     =check_preempt_curr_fair,

        .pick_next_task         =pick_next_task_fair,
        .put_prev_task          =put_prev_task_fair,

        .load_balance           =load_balance_fair,

        .set_curr_task          =set_curr_task_fair,
        .task_tick              =task_tick_fair,
        .task_new               =task_new_fair,
};
```

put_prev_task_fair에서는 se를 통해 se의 부모로 이동 가능하며 put prev_entity를 통해 cfs_rq에 삽입하는 것을 알 수 있습니다. 그러나 cfs의 흐름을 알 수 있는 것은 pick_next_task 이므로, pick_next_task를 통해 다음 cfs의 함수 흐름을 알 수 있으므로,pick next task fair 함수를 따라가 봤습니다.

```
staticstructtask_struct*pick_next_task_fair(structrq*rq)
{
        structcfs_rq*cfs_rq=&rq->cfs;
        structsched_entity*se;

        if(unlikely(!cfs_rq->nr_running))
                returnNULL;

        do{
                se=pick_next_entity(cfs_rq);
                cfs_rq=group_cfs_rq(se);
        }while(cfs_rq);

        returntask_of(se);
}
```

pick next task fair 함수에서, pick prev task fair에서 update한 cfs_rq가 running인지 판별을 하고, pick next entity, 즉 그다음 entity를 고르는 방법을 제시합니다. 따라서, pick next entity 함수를 따라가게 됩니다.

```
staticstructsched_entity*pick_next_entity(structcfs_rq*cfs_rq)
{
        structsched_entity*se=__pick_next_entity(cfs_rq);

        set_next_entity(cfs_rq,se);

        returnse;
}
```

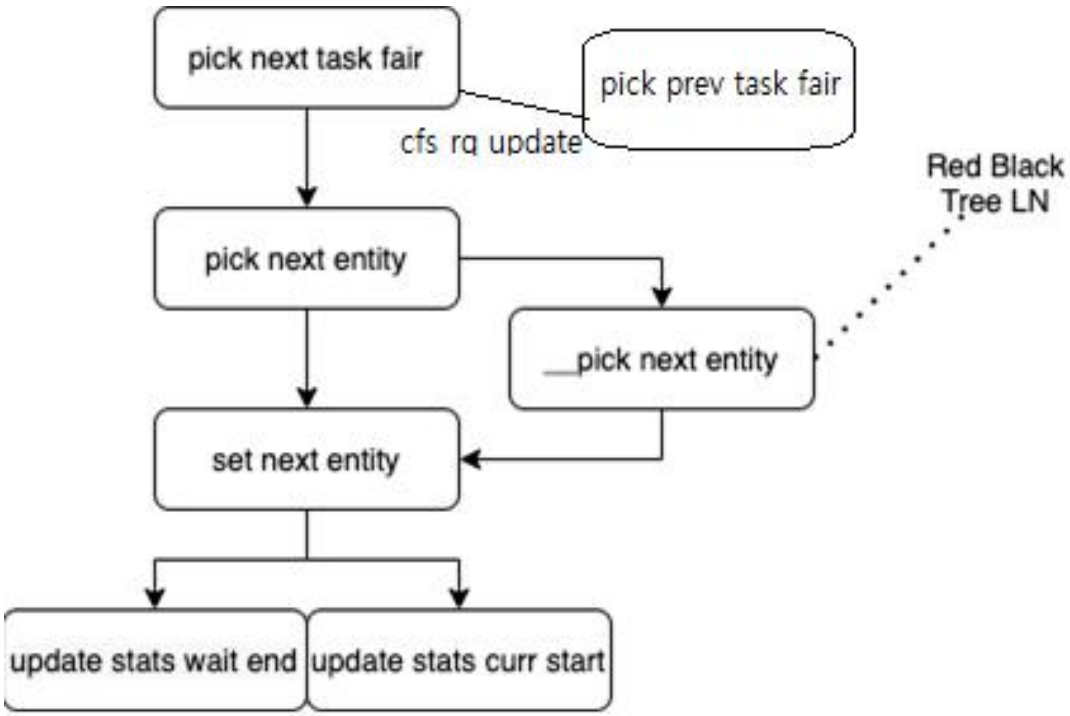
pick next entity에서는 __pick_next_entity가 있고, set_next_entity가 있습니다.
아래의 __pick next entity에서는 red black tree가 있으며, 가장 왼쪽의 node에서 가져옴을 말합니다. (우선순위) (vruntime 적은 순으로 정렬)

```
staticstructsched_entity*__pick_next_entity(structcfs_rq*cfs_rq)
```

```
{
        return rb_entry(first_fair(cfs_rq), struct sched_entity, run_node);
}
static inline void
set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{

        update_stats_wait_end(cfs_rq, se);
        update_stats_curr_start(cfs_rq, se);
        set_cfs_rq_curr(cfs_rq, se);
        se->prev_sum_exec_runtime = se->sum_exec_runtime;
}
```

__pick_next_entity에서 vruntime이 적은 se를 반환함을 알 수 있습니다.

이러한 runqueue와 se를 반환 한 것을 전달한 set_next_entity에서는 update_stats_wait_end 함수와, update_sats_curr_start함수로 이루어져있습니다. 이 두 함수에서 위에 전달한 runqueue와 se를 전달하여 fair 값을 계산하여 더할 수 있도록 실행합니다.



2-2) 4.19.146 버전
/kernel/sched/fair.c (이하 동일)

```
conststructsched_class fair_sched_class={
        .next                   =&idle_sched_class,
        .enqueue_task           =enqueue_task_fair,
        .dequeue_task           =dequeue_task_fair,
        .yield_task             =yield_task_fair,
        .yield_to_task          =yield_to_task_fair,

        .check_preempt_curr     =check_preempt_wakeup,

        .pick_next_task                 =pick_next_task_fair,
        .put_prev_task          =put_prev_task_fair,

#ifdef CONFIG_SMP
        .select_task_rq         =select_task_rq_fair,
        .migrate_task_rq =migrate_task_rq_fair,

        .rq_online              =rq_online_fair,
        .rq_offline             =rq_offline_fair,

        .task_dead              =task_dead_fair,
        .set_cpus_allowed       =set_cpus_allowed_common,
#endif

        .set_curr_task          =set_curr_task_fair,
        .task_tick              =task_tick_fair,
        .task_fork              =task_fork_fair,

        .prio_changed           =prio_changed_fair,
        .switched_from          =switched_from_fair,
        .switched_to            =switched_to_fair,

        .get_rr_interval   =get_rr_interval_fair,

        .update_curr            =update_curr_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
        .task_change_group      =task_change_group_fair,
#endif
};
```

pick_next_task를 통해 CFS 함수의 흐름을 따라 갈 수 있기 때문에, pick_next_task의 함수가 되는 pick_next_task_fair 함수를 찾아봤습니다.

```
staticstructtask_struct*
pick_next_task_fair(structrq*rq,structtask_struct*prev,structrq_flags*rf)
{
        structcfs_rq*cfs_rq=&rq->cfs;
        structsched_entity*se;
        structtask_struct*p;
```

```
        intnew_tasks;

again:
        if(!cfs_rq->nr_running)
                gotoidle;

#ifdef  CONFIG_FAIR_GROUP_SCHED
        if(prev->sched_class!=&fair_sched_class)
                gotosimple;

        /*
         * Because of the set_next_buddy() in dequeue_task_fair() it is rather
         * likely that a next task is from the same cgroup as the current.
         *
         * Therefore attempt to avoid putting and setting the entire cgroup
         * hierarchy, only change the part that actually changes.
         */

        do{
                structsched_entity*curr=cfs_rq->curr;

                /*
                 * Since we got here without doing put_prev_entity() we also
                 * have to consider cfs_rq->curr. If it is still a runnable
                 * entity, update_curr() will update its vruntime, otherwise
                 * forget we've ever seen it.
                 */
                if(curr){
                        if(curr->on_rq)
                                update_curr(cfs_rq);
                        else
                                curr=NULL;

                        /*
                         * This call to check_cfs_rq_runtime() will do the
                         * throttle and dequeue its entity in the parent(s).
                         * Therefore the nr_running test will indeed
                         * be correct.
                         */
                        if(unlikely(check_cfs_rq_runtime(cfs_rq))){
                                cfs_rq=&rq->cfs;

                                if(!cfs_rq->nr_running)
                                        gotoidle;

                                gotosimple;
                        }
                }

                se=pick_next_entity(cfs_rq,curr);
                cfs_rq=group_cfs_rq(se);
        }while(cfs_rq);

        p=task_of(se);
```

```
        /*
         * Since we haven't yet done put_prev_entity and if the selected task
         * is a different task than we started out with, try and touch the
         * least amount of cfs_rqs.
         */
        if(prev!=p){
                structsched_entity*pse=&prev->se;

                while(!(cfs_rq=is_same_group(se,pse))){
                        intse_depth=se->depth;
                        intpse_depth=pse->depth;

                        if(se_depth<=pse_depth){
                                put_prev_entity(cfs_rq_of(pse),pse);
                                pse=parent_entity(pse);
                        }
                        if(se_depth>=pse_depth){
                                set_next_entity(cfs_rq_of(se),se);
                                se=parent_entity(se);
                        }
                }

                put_prev_entity(cfs_rq,pse);
                set_next_entity(cfs_rq,se);
        }

        gotodone;
simple:
#endif

        put_prev_task(rq,prev);

        do{
                se=pick_next_entity(cfs_rq,NULL);
                set_next_entity(cfs_rq,se);
                cfs_rq=group_cfs_rq(se);
        }while(cfs_rq);

        p=task_of(se);

done:__maybe_unused;
#ifdef CONFIG_SMP
        /*
         * Move the next running task to the front of
         * the list, so our cfs_tasks list becomes MRU
         * one.
         */
        list_move(&p->se.group_node,&rq->cfs_tasks);
#endif

        if(hrtick_enabled(rq))
                hrtick_start_fair(rq,p);
```

```
        return p;

idle:
        new_tasks = idle_balance(rq, rf);

        /*
         * Because idle_balance() releases (and re-acquires) rq->lock, it is
         * possible for any higher priority task to appear. In that case we
         * must re-start the pick_next_entity() loop.
         */
        if (new_tasks < 0)
                return RETRY_TASK;

        if (new_tasks > 0)
                goto again;

        return NULL;
}
```

2.6.23 버전에서는 put_prev_entity가 put_prev_task_fair에 있었는데 pick_next_task_fair에 있는 것을 알 수 있었으며, 대기 상태의 기능 추가가 있음을 알 수 있습니다. pick prev task fair에서 하고 있었던 put prev entity의 함수의 역할처럼 prev를 cfs_rq에 삽임함을 알 수 있었습니다.

```
static struct sched_entity *
pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        struct sched_entity *left = __pick_first_entity(cfs_rq);
        struct sched_entity *se;

        /*
         * If curr is set we have to see if its left of the leftmost entity
         * still in the tree, provided there was anything in the tree at all.
         */
        if (!left || (curr && entity_before(curr, left)))
                left = curr;

        se = left; /* ideally we run the leftmost entity */

        /*
         * Avoid running the skip buddy, if running something else can
         * be done without getting too unfair.
         */
        if (cfs_rq->skip == se) {
                struct sched_entity *second;

                if (se == curr) {
                        second = __pick_first_entity(cfs_rq);
                } else {
                        second = __pick_next_entity(se);
                        if (!second || (curr && entity_before(curr, second)))
                                second = curr;
                }
```

```
                if(second&&wakeup_preempt_entity(second,left)<1)
                        se=second;
        }

        /*
         * Prefer last buddy, try to return the CPU to a preempted task.
         */
        if(cfs_rq->last&&wakeup_preempt_entity(cfs_rq->last,left)<1)
                se=cfs_rq->last;

        /*
         * Someone really wants this to run. If it's not unfair, run it.
         */
        if(cfs_rq->next&&wakeup_preempt_entity(cfs_rq->next,left)<1)
                se=cfs_rq->next;

        clear_buddies(cfs_rq,se);

        returnse;
}
```

```
/*
 * Pick the next process, keeping these things in mind, in this order:
 * 1) keep things fair between processes/task groups
 * 2) pick the "next" process, since someone really wants that to run
 * 3) pick the "last" process, for cache locality
 * 4) do not run the "skip" process, if something else is available
 */
```

다음 프로세스를 고르기 위해서는 다음 방법을 알고 있어야 합니다.
1) 프로세스와 태스크 그룹간에 fair하게 고르기
2) 다음 프로세스를 누가 정말 원하면 고르기
3) 마지막 프로세스를 캐시의 지역성을 위해 고르기
4) 다른 것이 가능하다면 (runable) skip 하지 않기

```
structsched_entity*__pick_first_entity(structcfs_rq*cfs_rq)
{
        structrb_node*left=rb_first_cached(&cfs_rq->tasks_timeline);

        if(!left)
                returnNULL;

        returnrb_entry(left,structsched_entity,run_node);
}
```

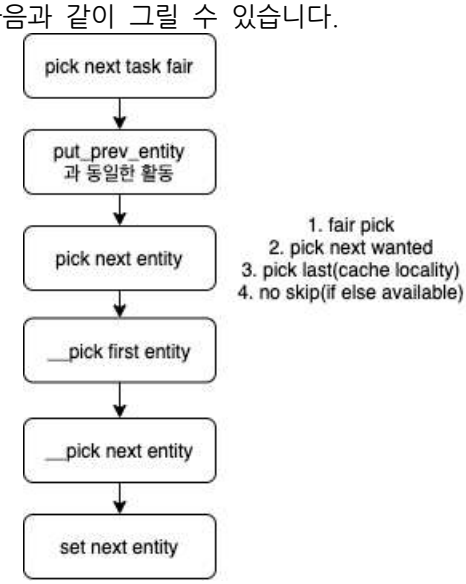
이전 버전과 마찬가지로 red black tree에 대해 가장 왼쪽 노드에서 se를 반환하게 됩니다.

```
staticstructsched_entity*__pick_next_entity(structsched_entity*se)
{
        structrb_node*next=rb_next(&se->run_node);

        if(!next)
                returnNULL;

        returnrb_entry(next,structsched_entity,run_node);
}
```

```
static void
set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        /* 'current' is not kept within the tree. */
        if (se->on_rq) {
                /*
                 * Any task has to be enqueued before it get to execute on
                 * a CPU. So account for the time it spent waiting on the
                 * runqueue.
                 */
                update_stats_wait_end(cfs_rq, se);
                __dequeue_entity(cfs_rq, se);
                update_load_avg(cfs_rq, se, UPDATE_TG);
        }

        update_stats_curr_start(cfs_rq, se);
        cfs_rq->curr = se;

        /*
         * Track our maximum slice length, if the CPU's load is at
         * least twice that of our own weight (i.e. dont track it
         * when there are only lesser-weight tasks around):
         */
        if (schedstat_enabled() && rq_of(cfs_rq)->load.weight >= 2*se->load.weight) {
                schedstat_set(se->statistics.slice_max,
                        max((u64)schedstat_val(se->statistics.slice_max),
                            se->sum_exec_runtime - se->prev_sum_exec_runtime));
        }

        se->prev_sum_exec_runtime = se->sum_exec_runtime;
}
```

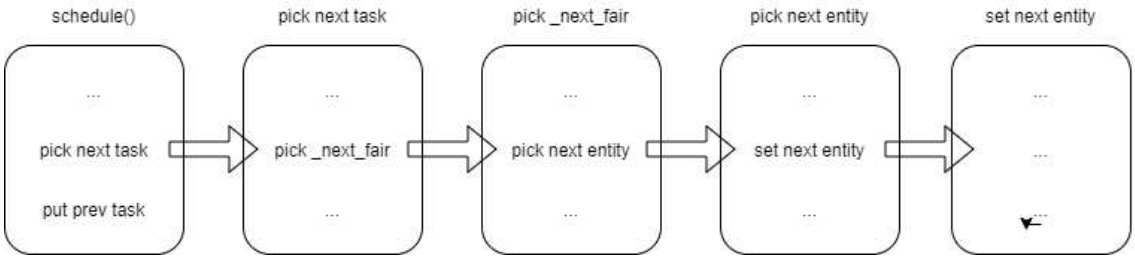
이전 2.6.23 버전과 마찬가지로, set_next_entity를 통해, 다음으로 고를 것을 어떻게 고르는지에 대해서 cfs_rq에 적용이 된 위의 두 함수를 통해 cfs_rq를 전달받아 next entity를 고르면서 scheduling이 완료가 됩니다.

큰 틀에서 cfs 순서도는 다음과 같이 그릴 수 있습니다.



3. 각 버전의 공통점, 차이점 분석

두 버전 모두 큰 틀에서는 크게 차이가 나지 않는 스케줄링기법을 보입니다.
스케줄에서, preempt와 interrupt에 변동이 있어 runqueue에서 변동이 있다면, pick next task -> pick next fair -> pick next entity -> set next entity의 큰 흐름을 따라가면서 runqueue를 바꾸며, 변동된 vruntime에 따른 task의 우선순위 비교를 통해 scheduling을 합니다. 두 스케줄 함수 모두 cpu가 가장 우선적으로 실행이 필요한 task를 실행하기 위해 priority로만 이루어진 것이 아닌 weight와 vruntime을 스케줄링에 이용하며, red black tree를 이용하여 스케줄링에 이용한다는 공통점이 있습니다. 큰 틀에서는 같지만 다른 점도 있습니다. CFS 함수에서 본 것처럼, 4.19.146 버전은 2.6.23버전과 비교해 put prev_task 함수에서, put prev entity를 작동시키지 않고 pick next fair이라는 함수에서 돌리게 되어 sched entity 뿐만 아니라, cfs runqueue에 영향을 주게 만듭니다. 이전 버전에서는 preempt disable만 이용했지만, local irq disable을 통해 cpu interrupt를 제어할 수 있도록 다르게 이용하고 있습니다. 또한, 2.6.23버전에서는 폴더 안에 여러 개의 파일들이 분산되어서 한눈에 파악하기 힘들지만, 4.19.146 버전에는 하나의 폴더에 잘 정리되어 있습니다.



4. References

System Programming 2nd Lecture Process scheduling
https://opensource.com/article/19/2/fair-scheduling-linux
https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
https://elixir.bootlin.com/linux/v2.6.23/source/kernel/sched_fair.c#L133
https://elixir.bootlin.com/linux/v2.6.23/source/include/linux/sched.h
https://elixir.bootlin.com/linux/v2.6.23/source/kernel/sched.c
https://elixir.bootlin.com/linux/v4.19.146/source/include/linux/sched.h
https://elixir.bootlin.com/linux/v4.19.146/source/kernel/sched/fair.c
http://jake.dothome.co.kr/cfs/

Ⅱ. 실습 과제 보고서

1. 과제 수행 시스템 환경

```
billygoat97@ubuntu:~$ uname -a
Linux ubuntu 4.19.146-2015147506 #15 SMP Sun Oct 18 18:13:15 PDT 2020 x86_64 x86
_64 x86_64 GNU/Linux
```

```
billygoat97@ubuntu:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
stepping        : 10
microcode       : 0x96
cpu MHz         : 2903.999
cache size      : 9216 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable no
nstop_tsc cpuid pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ib
rs ibpb stibp fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves arat flush_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass litf mds swapgs itlb_multihit srbds
bogomips        : 5807.99
clflush size    : 64
cache_alignment : 64
address sizes   : 45 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
stepping        : 10
microcode       : 0x96
cpu MHz         : 2903.999
cache size      : 9216 KB
physical id     : 2
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 2
initial apicid  : 2
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable no
nstop_tsc cpuid pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ib
rs ibpb stibp fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves arat flush_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass litf mds swapgs itlb_multihit srbds
bogomips        : 5807.99
clflush size    : 64
cache_alignment : 64
address sizes   : 45 bits physical, 48 bits virtual
power management:
```

```
billygoat97@ubuntu:~$ cat /proc/meminfo
MemTotal:        4012724 kB
MemFree:         1965632 kB
MemAvailable:    2388900 kB
Buffers:           63140 kB
Cached:           556340 kB
SwapCached:            0 kB
Active:          1151280 kB
Inactive:         342824 kB
Active(anon):     875948 kB
Inactive(anon):    24096 kB
Active(file):     275332 kB
Inactive(file):   318728 kB
Unevictable:           16 kB
Mlocked:               16 kB
SwapTotal:       2097148 kB
SwapFree:        2097148 kB
Dirty:                 0 kB
```

〈가상머신 2코어 4gb〉

Windows 버전

Windows 10 Pro

© 2019 Microsoft Corporation. All rights reserved.

시스템

프로세서:	Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz 2.90 GHz
설치된 메모리(RAM):	16.0GB
시스템 종류:	64비트 운영 체제, x64 기반 프로세서
펜 및 터치:	이 디스플레이에 사용할 수 있는 펜 또는 터치식 입력이 없습니다.

〈본체 4코어 16gb〉

2. 커널 수정 보고서

저는 이번 과제를 코딩함에 있어서 sched.h에는 변수 추가를 core.c 에는 코드수정을 하였습니다. 상기 내용은 다음과 같습니다.

```
unsigned long long last_sched_time; // sched recently
unsigned long long cpu_exec_time[255]; // for each cpu time
```

sched.h 에서는 가장 최근에 schedule된 시간을 저장하기 위한 last_sched_time을 가지고 있으며, cpu별 실행 시간을 저장하기 위해 cpu_exec_time이라는 배열을 추가하였습니다.

```
int cpu_num;
int max_cpu = 0;
struct task_struct* prev_task[255][10]; // send task info
unsigned long long per_cpu_time[255][10]; // send task scheduled info
unsigned long long imsi;
int cpu_count[255] = {0,};

void save_task(struct task_struct* prev, int cpu, unsigned long long itime){

    for_each_online_cpu(cpu_num){
        if(cpu_num > max_cpu)max_cpu = cpu_num;
    }
    if(cpu>max_cpu) return;
    if(cpu_count[cpu]>=10){ // style of queue
        cpu_count[cpu] = 9;
        prev_task[cpu][0] = prev_task[cpu][1];
        prev_task[cpu][1] = prev_task[cpu][2];
        prev_task[cpu][2] = prev_task[cpu][3];
        prev_task[cpu][3] = prev_task[cpu][4];
        prev_task[cpu][4] = prev_task[cpu][5];
        prev_task[cpu][5] = prev_task[cpu][6];
        prev_task[cpu][6] = prev_task[cpu][7];
        prev_task[cpu][7] = prev_task[cpu][8];
        prev_task[cpu][8] = prev_task[cpu][9];
    }

    prev_task[cpu][cpu_count[cpu]++] = prev; // remember task
}

EXPORT_SYMBOL(prev_task);
EXPORT_SYMBOL(per_cpu_time);
```

```
next -> last_sched_time = rq_clock(rq);
int zzz;
for(zzz=0;zzz<9;zzz++){
    per_cpu_time[cpu][zzz] = per_cpu_time[cpu][zzz+1];
}
per_cpu_time[cpu][9] = next ->last_sched_time;
save_task(next, cpu, imsi);
```

core.c에서는 task_struct 배열 선언을 하여 module로 전송을 하도록 하였습니다. 과제 명세에서는 cpu 개수가 총 255개를 넘지 않는다고 하여 최대 개수를 받을 정도로 배열을 생성하였으며, 생성한 배열에서 가용 가능한 cpu만큼 task를 받아오도록 하였습니다. 받는 task 객체는 reschedule 될 때마다 받아오게 하였습니다. task 객체가 10개가 넘어가는 경우, queue와 같은 형식으로 뒤에서 다시 재배열 하도록 하였습니다.

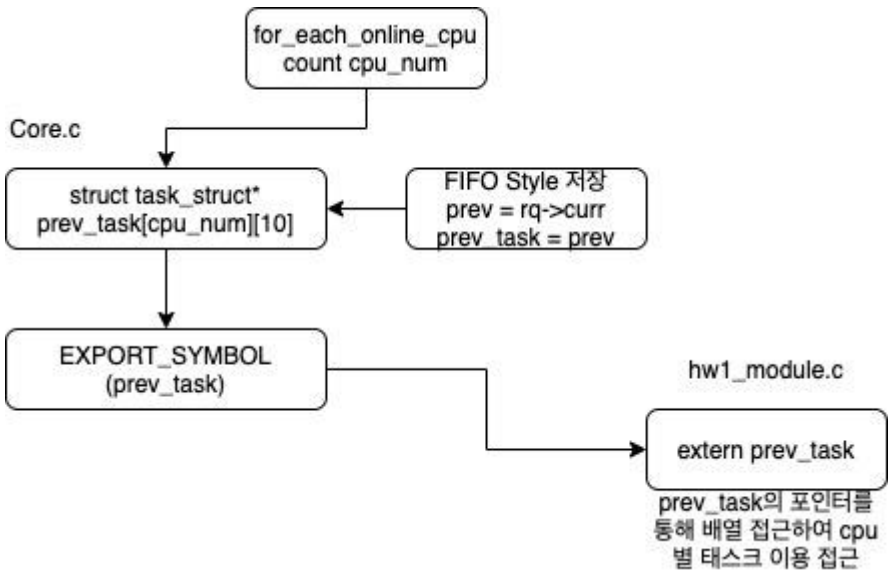
.task 배열의 포인터와 cpu sched time을 export_symbol를 이용하여 전송을 해줍니다. per_cpu_time은 cpu별 task 이용을 schedule시간을 받아오도록 하였는데 처음에 save _task 함수에 넣으려고 하였으나, 그 결과 포인터가 전달이 되어 시간이 export 할 때 원하는 시간이 전달이 되지 않았습니다. 그래서 그 자리에서 바로 변수화 시켜 전역 배열에 저장을 하여 export를 하였습니다.

```
prev->cpu_exec_time[cpu] += prev->se.sum_exec_runtime - prev ->se.prev_sum_exec_runtime;
```

또한 cpu별 실행 시간을 다음과 같이 전체 exec time - (전) exec time을 통해 각각 더하여 배열에 저장할 수 있었습니다.

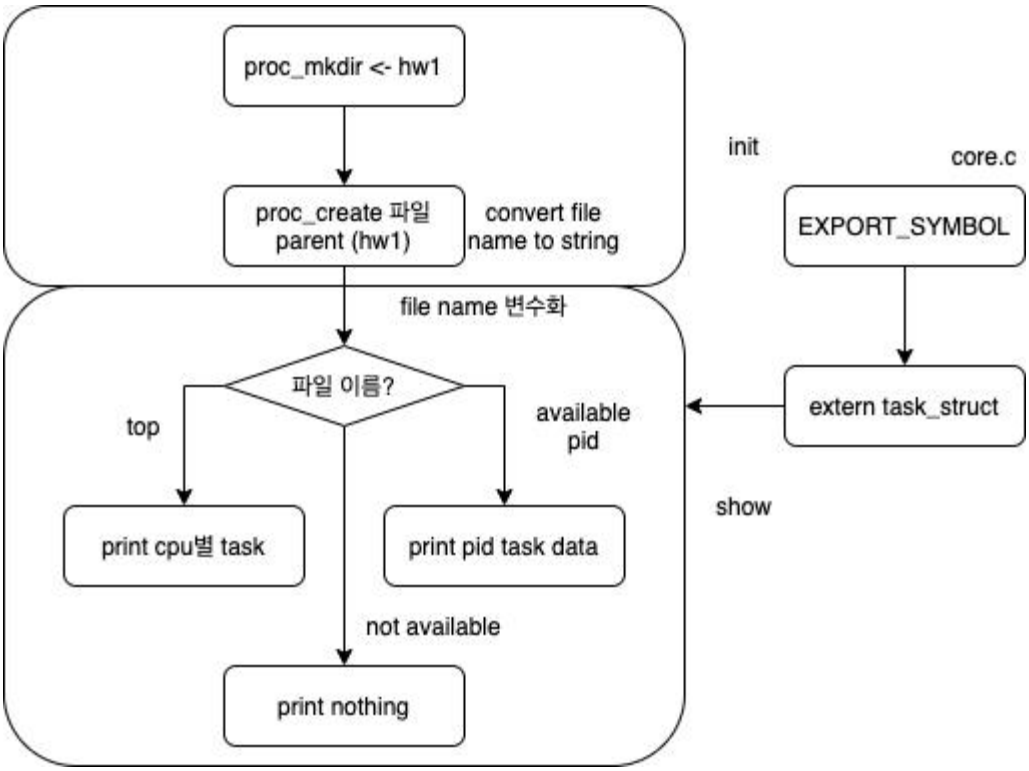
```
char * fname = "";
int imsi = 0;
extern struct task_struct* prev_task[][10];
extern unsigned long long per_cpu_time[][10];
```

보낸 export_symbol의 값은 hw1_module.c에서
extern을 통해 이용합니다 이는 뒤의 모듈 프로그래밍에서 다루도록 하겠습니다.



core.c에서 cpu별 태스크를 저장하여 가져오는 순서도는 크게 다음과 같습니다.

3. 커널 모듈 작성 보고서



크게 모듈의 구성을 보여준 순서도는 다음과 같습니다. 크게, 경로와 파일을 만들고, 파일이름을 비교하여 적절한지 비교 후, core.c에서 받아온 데이터를 적합하게 출력을 합니다.

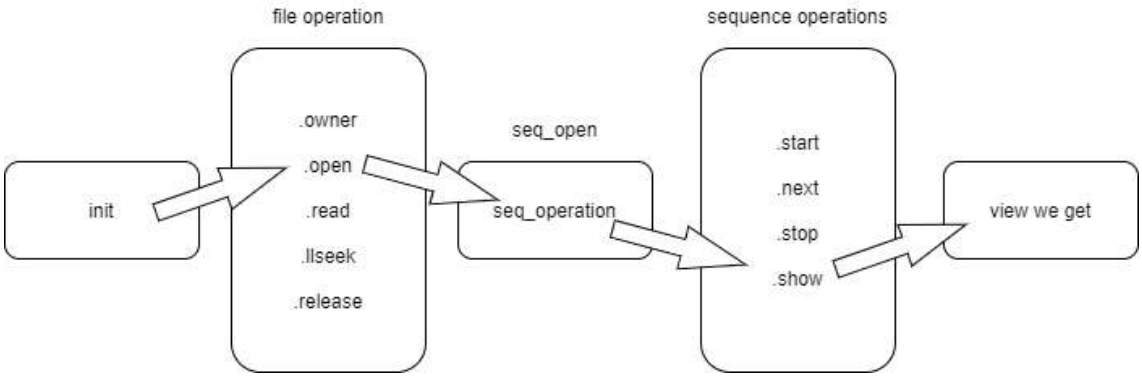
```
#include <linux/kernel.h> // for kernel create
#include <linux/module.h> // kernel create
#include <linux/proc_fs.h> // proc file
#include <linux/seq_file.h> // seq_file
#include <linux/init.h> // init, exit function
#include <linux/sched.h> // task_struct, sched_entity
#include <linux/sched/signal.h> //for_each_process
#include <linux/cpumask.h> // for_each_online_cpu
#include <linux/string.h> // sprintf
```

가장 처음에는 다음과 같은 header을 선언을 했습니다. 각 header 별로 가장 필요해서 추가한 것들을 주석한것과 같이 표시해 두었습니다.

```c
static int __init sched_init(void) { //initialize file
    struct proc_dir_entry *proc_directory; // directory
    proc_directory = proc_mkdir("hw1",NULL);
    struct proc_dir_entry *proc_file_entry; // top file
    proc_file_entry = proc_create("top", 0, proc_directory, &sched_file_ops);


    struct proc_dir_entry *proc_pid[32768]; // create all pid file possible does not ensure availability
    int i;
    for(i = 0; i<32768;i++){ // pid possible #
        char str[50] = "";
        sprintf(str, "%d",i); // change to string
        proc_pid[i] = proc_create(str, 0, proc_directory, &sched_file_ops); // create
    }
    return 0;
}
```

본격적 모듈 프로그래밍은 다음과 같습니다. 제일 먼저 sched_init을 통해 경로와 file을 생성을 합니다. hw1이라는 경로를 다음과 같이 mkdir을 이용해 만들었으며, 전체를 볼 수 있는 top이라는 파일을 만들었습니다. 두 번째 사항인 pid 파일을 만드는 애로사항이 있었는데, 이는, for each process를 통해 pid를 받아와, sprintf 함수를 통해 문자열 형태로 바꿔서 파일 생성을 완료하였습니다. 파일 생성을 했을 때, hw1이라는 경로를 부모로 받을 수 있도록 proc create 세 번째 칸에 proc_directory를 추가하여 지정해 주었습니다.

우리가 화면을 보는 show function까지 가기 위해서는 간단히 설명하자면,



다음과 같은 그림의 형태로 우리가 보는 view까지 갈 수 있습니다.

```c
static int sched_proc_open(struct inode *inode, struct file *file) {
    fname = file->f_path.dentry->d_name.name; // grab file name super duper important ***
    return seq_open(file, &sched_seq_ops);
}
```

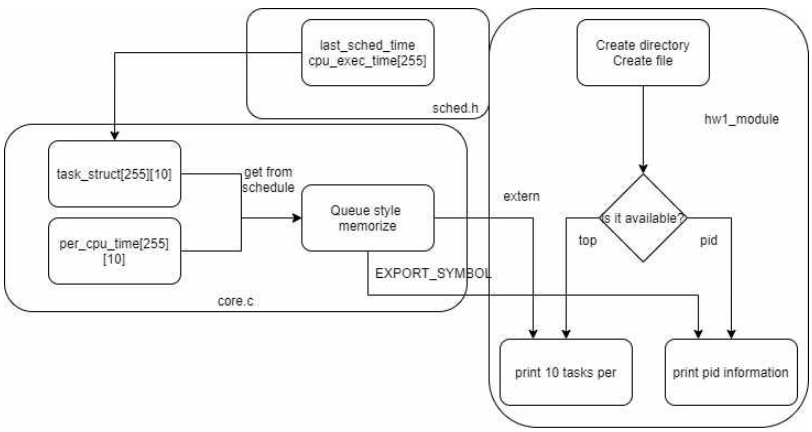
파일 생성을 하였을 때, top의 경우에는 이름이 명확하게 정해져 있어서 코딩을 할 수 있었으나, pid 파일의 경우, 그 상황에 맞는 파일 이름을 특정화하기 위해서 fname이라는 문자열을 만들어, file -> f_path.dentry->d_name.name;을 통해 파일 이름을 변수로 사용 가능하게 만들었습니다.

```c
static int sched_seq_show(struct seq_file *s, void *v)
{

    int count = 0;
    int c = 0;
    struct task_struct *task;
    for_each_process(task){
        count++;
        if(task->state == 0 ||task->state == 1||task->state == 2){ // running? blocked?
            c++;
        }
    }
```

가능한 task들의 수를 모두 파악하기 위해서 for_each_process를 통해 task 수를 확인해 보았습니다. 저는 running 혹은 blocked라는 것만 세라는 qna를 보고, 0,1,2의 state를 갖고 있는 것만이 running 혹은 blocked라고 판단했습니다. count는 총 process를, 그중에서 running 혹은 blocked는 c로 count를 하였습니다.

```c
static void print_task(struct seq_file *s, struct task_struct *task, unsigned long long pct){ // get task
        if(task -> static_prio >= 100){ // CFS
            seq_printf(s, "%16s %5d %12lld CFS\n", task->comm, task->pid, pct/NM); // name, pid
        }else{  // RT
            seq_printf(s, "%16s %5d %12lld  RT\n", task->comm, task->pid, pct/NM);
        }
}
```

top 파일을 호출한 경우, 총 몇 개의 task가 있는지, 그리고 각 cpu당 어떤 task가 recently scheduled 되었는지를 표시하기 위해 print_task를 for_each_online_cpu를 통해서 호출하였습니다. 모듈과, 커널을 통한 모든 정보 flow는 다음과 같습니다.



```
//seq_printf(s, "Total %d tasks\n", c);
if(strcmp("top", fname)==0){

    //loff_t *spos = (loff_t *) v;
    print_bar(s);
    seq_printf(s, "System Programming Assignment 1]\n");
    seq_printf(s, "ID: 2015147506, Name: Kim, KiHyun\n");
    seq_printf(s, "Total %d tasks, %dHz\n", c, HZ);
    print_bar(s);
    int cpu;
    static int max_cpu = 0;
    for_each_online_cpu(cpu){
        if(cpu > max_cpu)max_cpu = cpu;
    }
    //seq_printf(s,"%d",max_cpu);
    //print_bar(s);
    for_each_online_cpu(cpu){

        seq_printf(s, "CPU %d\n", cpu);
        print_bar(s);
        print_task(s, prev_task[cpu][0], per_cpu_time[cpu][0]);
        print_task(s, prev_task[cpu][1], per_cpu_time[cpu][1]);
        print_task(s, prev_task[cpu][2], per_cpu_time[cpu][2]);
        print_task(s, prev_task[cpu][3], per_cpu_time[cpu][3]);
        print_task(s, prev_task[cpu][4], per_cpu_time[cpu][4]);
        print_task(s, prev_task[cpu][5], per_cpu_time[cpu][5]);
        print_task(s, prev_task[cpu][6], per_cpu_time[cpu][6]);
        print_task(s, prev_task[cpu][7], per_cpu_time[cpu][7]);
        print_task(s, prev_task[cpu][8], per_cpu_time[cpu][8]);
        print_task(s, prev_task[cpu][9], per_cpu_time[cpu][9]);
        print_bar(s);

    }
```

print task의 설명입니다. 해당되는 task를 상기 함수에서 받아오면, 이를 통해 task의 이름, pid, 그리고 최근 schedule된 시간을 지정해 줍니다. 그리고 각 타입을 출력해 주는데 priority에 ᄯᅳ라서 cfs 혹은 rt로 구분되게 됩니다.

```
    }
}else{
    struct task_struct *t;
    int flag = 0;
    for_each_process(t){
        int pid;

        char pname[50] = "";
        sprintf(pname, "%d",t -> pid);
        if(strcmp(pname, fname) == 0){// match available pid?
            flag = 1;
            break;
        }

    }
    if(flag == 1){
        print_pid(s,t);
    }

}
```

만약에 파일이 top과 다르다면, pid와 비교를 하게 됩니다. 만약에 pid와 이름이 같다면, 그중에서도 available한 pid와 같다면 print_pid 함수를 출력을 하도록 하였습니다.

```c
static void print_pid(struct seq_file *s, struct task_struct *task){
    print_bar(s);
    seq_printf(s, "System Programming Assignment 1]\n");
    seq_printf(s, "ID: 2015147506, Name: Kim, KiHyun\n");
    print_bar(s);
    seq_printf(s, "Command: %s\n", task->comm);
    seq_printf(s, "Type: ");
    if(task -> static_prio >= 100){ // CFS
        seq_printf(s, "CFS\n");
    }else{  // RT
        seq_printf(s, "RT\n");
    }
    seq_printf(s, "PID: %d\n", task->pid); //pid
    seq_printf(s, "Start time: %lld (ms)\n", task ->start_time/NM);
    seq_printf(s, "Last Scheduled Time: %lld (ms)\n", task -> last_sched_time/NM);// get from core.c
    seq_printf(s, "Last CPU #: %d\n", task ->recent_used_cpu); // recent cpu
    seq_printf(s, "Priority: %d\n", task -> static_prio); //pid
    unsigned long long tEt =0;
    int cpu;
    for_each_online_cpu(cpu){
        tEt += task->cpu_exec_time[cpu]/NM;  // sum up all exec time
    }
    seq_printf(s, "Total Execution time: %lld (ms)\n", tEt);
    for_each_possible_cpu(cpu);
    for_each_online_cpu(cpu){
        seq_printf(s, "- CPU %d: %lld(ms)\n", cpu, (task->cpu_exec_time[cpu]/NM)); // each exec time
    }
    if(task -> static_prio >= 100){ // CFS
        seq_printf(s, "Weight: %ld\n", task->se.load.weight); //weight
        seq_printf(s, "Virtual Runtime: %lld\n", task->se.vruntime); //runtime
    }else{  // RT

    }
}
```

이번 과제의 두 번째 핵심이자 가장 난이도가 있는 printpid함수입니다. 위에서부터 설명을 하지면, 이름, pid type과 sched는 다 같지만, start_time과, last cpu, priority 출력은 받아온 task에 내장되어 있는 변수를 이용하여 출력하였습니다. cpu별 exec time은, sched.h에서 저장한 cpu별 exec time을 core.c에서 계산한 듯이, sum_exec_time과 직전 sum_exec_time의 차를 통해 점진적으로 더한 값을 표현 할 수 있습니다. 이를 통해 task별 cpu 이용시간과 그 합을 출력할 수 있었습니다.
마지막으로는 task가 우선순위의 척도에 따라 CFS인 경우, task->se.load.weight과, task->se.runtime을 통해서 출력하도록 하였습니다.

MakeFile & 컴파일 과정

```
obj-m = hw1_module.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
	rm -rf *.ko *.mod.* .*.cmd *.o *.symvers *.order *.mod
```

기존에 과제 명세에서 제시된 make file을 조금 modify 하였습니다. obj -m는 hw1_module로 변경을 하였으며, clean을 할 때, .mod 파일이 하나 남는 것을 발견하여 이를 방지하고자 *.mod를 하나 더 추가해 주었습니다.

```
billygoat97@ubuntu:~/module$ make clean
rm -rf *.ko *.mod.* .*.cmd *.o *.symvers *.order *.mod
billygoat97@ubuntu:~/module$ make
billygoat97@ubuntu:~/module$ sudo rmmod hw1_module
[sudo] password for billygoat97:
billygoat97@ubuntu:~/module$ sudo insmod hw1_module.ko
billygoat97@ubuntu:~/module$ cat /proc/hw1/top
```

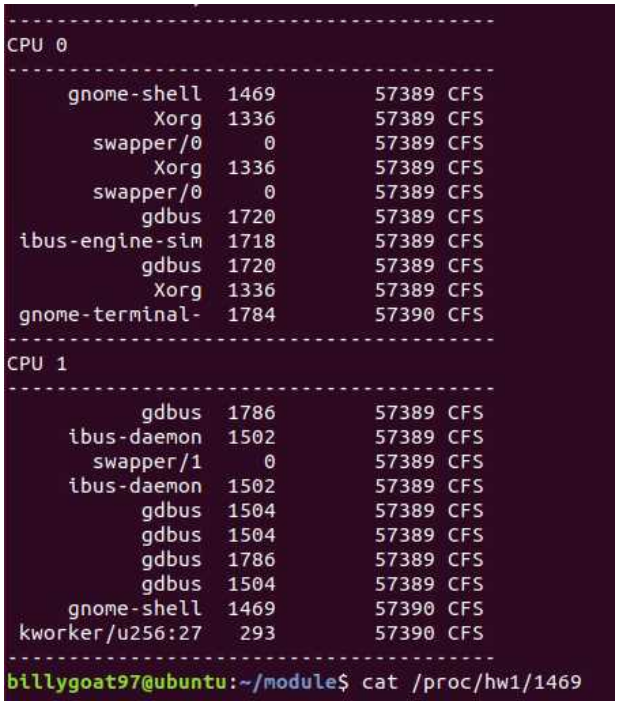
make clean을 하여 있을지 모르는 불필요한 파일을 삭제하였으며, make를 통해 필요한 모듈 파일을 생성하였습니다. rmmod를 통해 기존에 있었던 hw1_module이 중복으로 있는 것을 detach하였고, insmod를 통해 hw1_moduel.ko를 attach하였습니다.

그 이후에는 명령어를 입력 할 수 있었습니다.

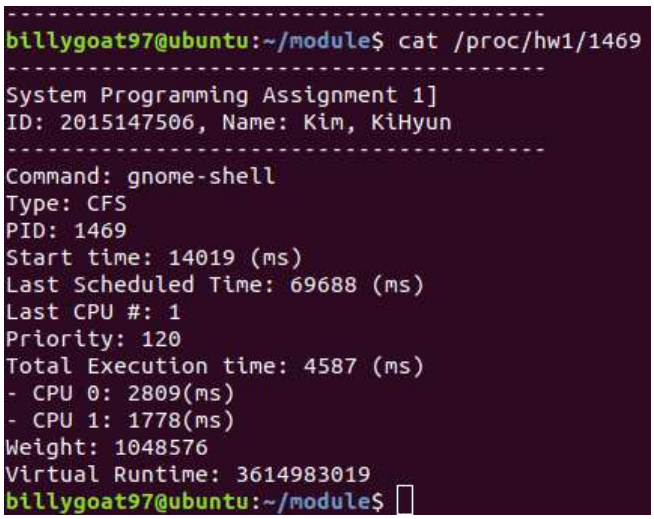
```
billygoat97@ubuntu:~$ sudo cp /boot/config-$(uname -r) ./.config
billygoat97@ubuntu:~$ make menuconfig
billygoat97@ubuntu:~$ sudo fakeroot make-kpkg --initrd kernel_image kernel_headers -j 2
billygoat97@ubuntu:~$ sudo dpkg -i linux-image-4.19.146-2015147506_4.19.146-2015147506-10.00.Custom_amd64.deb
billygoat97@ubuntu:~$ sudo dpkg -i linux-headers-4.19.146-2015147506_4.19.146-2015147506-10.00.Custom_amd64.deb
```

커널 컴파일을 위해서는 다음과 같이 주어진 명령어를 이용했으며, 다운그레이드가 안되는 것은 grub 파일에서 default를 변경하여 실행하였습니다. grub 파일 관련은 애로사항에 자세히 작성해 놓았습니다.

실행 결과



cat /proc/hw1/top을 실행 시켰을 때는, 각 cpu별로 task가 출력이 되었으며, 각 task 별로 이름, pid last_sched_time 시간이 잘 출력이 됨을 목격할 수 있으며 type도 확인 할 수 있었습니다.



그 후에, gnome-shell라는 이름을 가진 1469 pid를 입력 한 결과, 이름, type, pid, start time, last scheduled time을 잘 목격할수 있으며, 최근에 이용한 cpu, 우선순위를 목격 할 수 있습니다. 각 cpu별 execution 시간을 확인할 수 있으며, 이 합이 total execution 타임과 같음을 볼 수 있었습니다. 이 pid의 경우 cfs이므로, weight와 virtual runtime도 출력이 되었습니다.

4. 문제점, 해결방안, 애로사항

- pid 파일 생성을 하기 위해 pid를 string으로 변경하여 생성을 해야 했는데, char*배열로 생성을 하려고 해서 계속 모듈을 돌릴 때 killed가 떴었습니다. sprintf를 linux/string 헤더를 추가해서 만들 수 있었습니다.
- 구글에 검색해봤을 때, 생소한 부분이기 때문에 원하는 내용이 잘 나와있지 않아서 검색 뿐만 아니라 원서 책을 찾아보는 일이 많았으며, core.c를 변경시킬때마다 컴파일 시간이 너무 오래 걸린 문제가 있었습니다.
- 파일명을 특정화 하여서 변수로 이용하려고 했는데, top을 제외한 pid 파일들을 특정할 수 없었습니다. 이에, 상기 설명했던 file ->f_path.dentry->d_name.name를 통해 변수화 하여서 출력 가능하게 하였습니다.
- EXPORT_SYMBOL()을 처음 이용하려고 하였을 때, 포인터 전달을 위해서 함수를 만들어서 export 하려고 했는데, 제대로 전달이 되지 않았습니다. 따라서 배열 그 자체를 전달을 해주는 방안을 통해서 task 전달을 할 수 있었습니다.
- for_each_process라던지, for_each_online_cpu같은 가용 가능한 부분을 이용하기 위해서 다른 header들을 검색해서 추가하는 과정을 찾기 쉽지 않았습니다.

```
GRUB_DEFAULT=4
#GRUB_TIMEOUT_STYLE=hidden
#GRUB_TIMEOUT=0
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet"
GRUB_CMDLINE_LINUX="find_preseed=/preseed.cfg auto noprompt priority=critical locale=en_US"
```

- 커널을 다시 설치하려고 하였을 때, grub 파일을 수정하여 자동으로 최신 image로 열리는 것을 방지하여 원하는 4.19.146 버전으로 실행할 수 있었습니다. 문제가 됐던 부분은 memtest86을 계속적으로 부팅할 때 시작하여 운영체제를 돌리지 못했습니다. 이를 위해서 1초동안 잠시 나타나는 초기 부팅 화면에서 f12를 눌러 임의로 선택을 하도록 했습니다.
- top 파일 출력 시에 task는 잘 받아오는 것에 반해 last_sched_time을 배열로 10개씩 받아오려고 하는데 시간이 잘 안받아와 졌습니다. 이는 pointer을 전달했지 값을 전달한 것이 아니였기 때문입니다. 따라서 전역 변수를 선언, 함수에 넣지 않고 바로 값을 대입하도록 변경하였습니다.

5. References

System Programming 2nd Lecture Process scheduling
https://opensource.com/article/19/2/fair-scheduling-linux
https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
http://rousalome.egloos.com/10012007
http://rousalome.egloos.com/9990378
http://rousalome.egloos.com/9990377
http://rousalome.egloos.com/9990163
https://www.cs.fsu.edu/~baker/opsys/examples/task_struct.html
https://ljhh.tistory.com/entry/struct-taskstruct%EC%9D%98-%EA%B5%AC%EC%A1%B0
https://hidekuma.github.io/ec2/ubuntu/linux/remount-ebs/
http://esos.hanyang.ac.kr/tc/david/i/entry/file-%EA%B5%AC%EC%A1%B0%EC%B2%B4%EC%97%90%EC%84%9C-file-name-%EC%B6%9C%EB%A0%A5%ED%95%98%EA%B8%B0#_post_62
http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch12lev1sec7.html
https://modoocode.com/66