

System Programming 1st Assignment

- kernel analysis & scheduler analysis & kernel module programming -



목차

1. task_struct 분석
2. scheduler code 분석
3. 실습 과제 보고서
 - 가. scheduler 비교 보고서
 - 나. 커널 모듈 작성 보고서

2014147550
컴퓨터과학과
강효림

1. task_struct 분석

가. 실습과 관련된 자료구조와 macro

1) Jiffies와 HZ

jiffies는 전역 시스템 타이머로, 그 초기값은 INITIAL_JIFFIES에 정의되어있고, 부팅 후 1초에 HZ만큼 증가한다. 이하 declaration을 본다

```
649 EXPORT_SYMBOL(get_jiffies_64);
650 #endif
651
652 EXPORT_SYMBOL(jiffies);

9  #ifndef HZ
10  # ifndef CONFIG_ALPHA_RAWHIDE
11  #  define HZ    1024
12  # else
13  #  define HZ    1200
14  # endif
15  #endif

131 /*
132  * Have the 32 bit jiffies value wrap 5 minutes after boot
133  * so jiffies wrap bugs show up earlier.
134  */
135 #define INITIAL_JIFFIES ((unsigned long)(unsigned int) (-300*HZ))
```

즉 jiffes는 매 $\frac{1}{HZ}$ 초마다 갱신되며, HZ는 asm.h에 아키텍처별로 다르게 define 되어 있다.

2) start_time, stime, utime 등 time 관련 정보

```
1012
1013 unsigned int rt_priority;
1014 cputime_t utime, stime, utimescaled, stimescaled;
1015 cputime_t gtime;
1016 cputime_t prev_utime, prev_stime;
1017 unsigned long nvcs, nivcs; /* context switch counts */
1018 struct timespec start_time; /* monotonic time */
1019 struct timespec real_start_time; /* boot based time */
```

cputime_t는 asm-generic/cputime.h파일에 unsigned long으로 정의되어 있다. 따라서 utime, stime은 당해 프로세스가 사용한 user time, system time을 담고 있는 unsigned long형 자료형이다. 이 때 이 자료형의 값은 milisecond 등 시간 단위가 아니라, jiffies로 카운트 되므로, 실제 시간 값을 얻기 위해서는 $\frac{1000}{HZ}$ 를 곱해주어야 한다.

한편, start_time은 프로세스가 생성된 시각으로써, timespec 구조체로 정의되어 있는데, 이는 다음과 같다.

```
11 #ifndef _STRUCT_TIMESPEC
12 #define _STRUCT_TIMESPEC
13 struct timespec {
14     time_t tv_sec; /* seconds */
15     long tv_nsec; /* nanoseconds */
16 };
17 #endif
```

위와 같이 sec 단위, nanoseconds 단위로 구성되어 있기 때문에 jiffies로 count 되는 utime, stime보다 편리하게 바로 값을 꺼내 쓸 수 있다.

3) macro - for_each_process

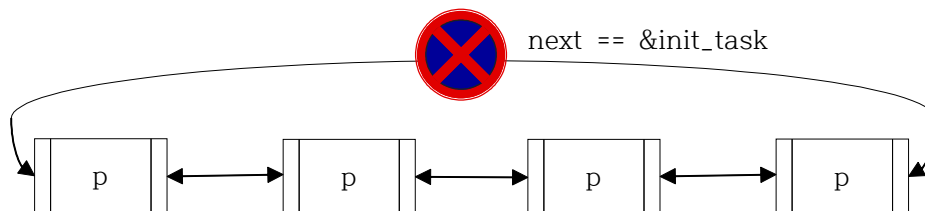
전체 프로세스를 순회하는 매크로이며, for each 문과 비슷하게 작동한다. sched.h에 정의되어 있다.

```
1702 #define next_task(p)    list_entry(rcu_dereference((p)->tasks.next), struct task_struct, tasks)
1703
1704 #define for_each_process(p) \
1705     for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

한편, 여기서의 INIT_TASK의 정의는 다음과 같다.

```
117 /*
118  * INIT_TASK is used to set up the first task table, touch at
119  * your own risk!. Base=0, limit=0x1fffff (=2MB)
120  */
121 #define INIT_TASK(tsk) \
122 {
123     .state          = 0,
124     .stack           = &init_thread_info,
```

이 매크로의 동작 과정을 그림으로 도시하면 다음과 같다.



(next가 init_task의 주소값이면 다 순회한 것이므로 break 됨을 표시함)

나. 프로세스 관리에 관련된 자료구조

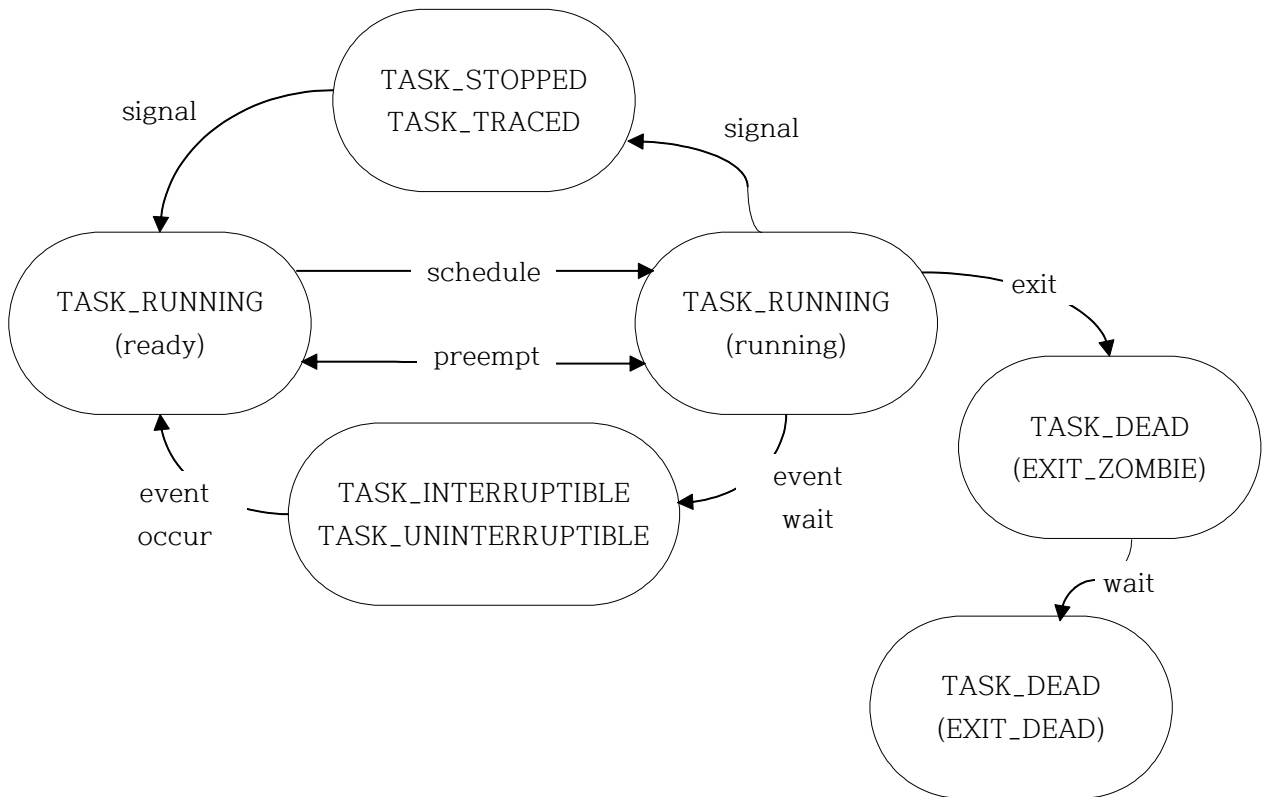
1) state

```
916
917 struct task_struct {
918     volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped
919     void *stack;
920     atomic_t usage;
921     unsigned int flags;    /* per process flags, defined below */
922     unsigned int ptrace;
```

task_struct의 가정 첫 부분에 정의되어 있으며, task의 상태를 나타내는 long 값이다. 이하와 같은 값을 가질 수 있다.

```
170 #define TASK_RUNNING      0
171 #define TASK_INTERRUPTIBLE 1
172 #define TASK_UNINTERRUPTIBLE 2
173 #define TASK_STOPPED     4
174 #define TASK_TRACED      8
175 /* in tsk->exit_state */
176 #define EXIT_ZOMBIE      16
177 #define EXIT_DEAD       32
178 /* in tsk->state again */
179 #define TASK_DEAD       64
```

task는 여러 가지로 상태가 변화할 수 있는데, 앞서 본 매크로에 따른, state가 가질 수 있는 값을 중심으로 상태 변화도를 도시하면 다음과 같다.



한편, state의 갱신에 대해선 다음 매크로가 정의되어 있어 이를 이용한다.

```

181 #define __set_task_state(tsk, state_value) \
182     do { (tsk)->state = (state_value); } while (0) \
183 #define set_task_state(tsk, state_value) \
184     set_mb((tsk)->state, (state_value)) \
171 #define __set_current_state(state_value) \
172     do { current->state = (state_value); } while (0) \
173 #define set_current_state(state_value) \
174     set_mb(current->state, (state_value))

```

2) task relationship

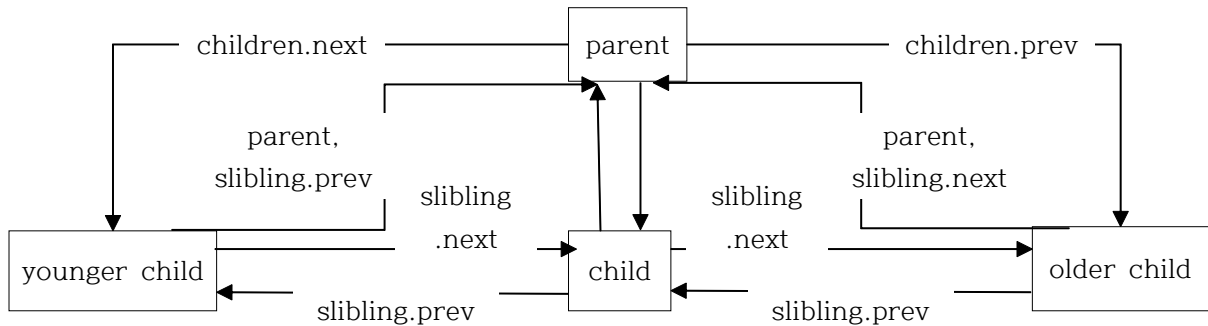
task_struct는 또한 parent process와 child, sibling을 표시하는 자료구조를 갖고 있는데, 이는 아래와 같다.

```

995 struct task_struct *real_parent; /* real parent process (when being debugged) */
996 struct task_struct *parent; /* parent process */
997 /*
998  * children/sibling forms the list of my children plus the
999  * tasks I'm ptracing.
1000  */
1001 struct list_head children; /* list of my children */
1002 struct list_head sibling; /* linkage in my parent's children list */
1003 struct task_struct *group_leader; /* threadgroup leader */
1004

```

이하 그 연결관계를 도시한다.



여기서 sibling, children은 list_head 구조체로서, next와 prev 속성값을 갖고, 그림에서 sibling.prev와 같은 표현은 그 속성값의 접근을 말한다.

다. process scheduling 관련 자료구조

1) sched_class와 sched_entity의 선언

```

931
932     int prio, static_prio, normal_prio;
933     struct list_head run_list;
934     const struct sched_class *sched_class;
935     struct sched_entity se;
  
```

2) sched_class

```

824 struct sched_class {
825     const struct sched_class *next;
826
827     void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
828     void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
829     void (*yield_task) (struct rq *rq);
830
  
```

위와 같이 scheduling 관련 function pointer들을 저장하는 구조체로써, scheduling policy에 따라 저 함수 포인터들은 다른 함수들을 가리킨다.

3) sched_entity

```

866 struct sched_entity {
867     struct load_weight    load;           /* for load-balancing */
868     struct rb_node        run_node;
869     unsigned int          on_rq;
870
871     u64                    exec_start;
872     u64                    sum_exec_runtime;
873     u64                    vruntime;
874     u64                    prev_sum_exec_runtime;
  
```

scheduling에서 필요한 정보들을 종합하여 담은 struct로써, CFS의 특징적인 자료구조인 rb_node, 그리고 vruntime을 볼 수 있다. 위 자료구조의 사용은 scheduler 분석에서 부연한다.

라. Context Switching 관련 Macro, function

```
1929 static inline void
1930 context_switch(struct rq *rq, struct task_struct *prev,
1931               struct task_struct *next)
1932 {
1933     struct mm_struct *mm, *oldmm;
1934
1935     prepare_task_switch(rq, prev, next);
1936     mm = next->mm;
1937     oldmm = prev->active_mm;
```

context_switch() 함수는, process에서 process로 context가 switching될 때 불리는 함수로, prepare_task_switch()를 통하여 switching을 준비하고,

```
1945     if (unlikely(!mm)) {
1946         next->active_mm = oldmm;
1947         atomic_inc(&oldmm->mm_count);
1948         enter_lazy_tlb(oldmm, next);
1949     } else
1950         switch_mm(oldmm, mm, next);
1951
1952     if (unlikely(!prev->mm)) {
1953         prev->active_mm = NULL;
1954         rq->prev_mm = oldmm;
```

위와 같은 과정을 통하여 task image를 바꿨다. 이후 register와 stack을 바꾸는데 다음과 같은 함수를 사용하는데,

```
1966     /* Here we just switch the register state and the stack. */
1967     switch_to(prev, next, prev);
```

그 구현은 다음과 같이 assembly언어로 되어있다. (다만 구체적인 구현은 아키텍처마다 다르다.)

```
19 #define switch_to(prev,next,last) do {
20     unsigned long esi,edi;
21     asm volatile("pushfl\n\t" /* Save flags */
22                 "pushl %%ebp\n\t"
23                 "movl %%esp,%0\n\t" /* save ESP */
24                 "movl %5,%%esp\n\t" /* restore ESP */
25                 "movl $1f,%1\n\t" /* save EIP */
26                 "pushl %6\n\t" /* restore EIP */
27                 "jmp __switch_to\n\t"
28                 "1:\n\t"
29                 "popl %%ebp\n\t"
30                 "popfl"
31                 : "=m" (prev->thread.esp), "=m" (prev->thread.eip),
32                 "=a" (last), "=S" (esi), "=D" (edi)
33                 : "m" (next->thread.esp), "m" (next->thread.eip),
34                 "2" (prev), "d" (next));
35 } while (0)
```

이후 finish_task_switching을 통하여 cleanup하며 context switching을 마무리한다.

```
1975     finish_task_switch(this_rq(), prev);
    그리고 이하는 그 구현이다.
```

```
1872 static void finish_task_switch(struct rq *rq, struct task_struct *prev)
1873     __releases(rq->lock)
1874 {
1875     struct mm_struct *mm = rq->prev_mm;
1876     long prev_state;
1877
1878     rq->prev_mm = NULL;
```


마. 다른 커널 버전과의 차이점

1) O(1) version kernel : 2.6.20

가) thread_info 관련

CFS kernel의 경우는, task_struct에서 **void* stack**을 통하여 thread union을 정의하고, 그 내부에 thread_info를 갖는다. 그러나 2.6.20 kernel은 thread union을 갖지 않고 다음과 같이 thread_info를 task_struct attribute로 갖는다.

```
801 struct task_struct {
802     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
803     struct thread_info *thread_info;
804     atomic_t usage;
805     unsigned long flags; /* per process flags, defined below */
806     unsigned long ptrace;
```

나) scheduling 관련

뒤에서 살피겠지만 O(1)은 bitmap으로 표현되는 일련의 run queue set을 manage 하면서 scheduling을 수행한다. 따라서 scheduling을 위한 요소들이 CFS scheduler를 사용하는 2.6.24와는 상이하다. 예컨대, vruntime을 갖는 sched_entity 대신 다음과 같은 자료구조를 갖는다.

```
815     int load_weight; /* for niceness load balancing purposes */
816     int prio, static_prio, normal_prio;
817     struct list_head run_list;
818     struct prio_array *array;
819
```

이 때 prio_array는 아래와 같이 정의되는 struct이다.

```
192 struct prio_array {
193     unsigned int nr_active;
194     DECLARE_BITMAP(bitmap, MAX_PRIO+1); /* include 1 bit for delimiter */
195     struct list_head queue[MAX_PRIO];
196 };
```

보다시피 비트맵을 갖는다.

2) O(n) version kernel : 2.4.31

scheduling 할 때마다 모든 process의 goodness를 계산하는데, (이에 관해선 다음 목차에서 서술) 그 때 필요한 정보의 빠른 접근을 보장하기 위하여 다음과 같이 task_struct에 직접 그 정보를 갖는다.

```
300 /*
301  * offset 32 begins here on 32-bit platforms. We keep
302  * all fields in a single cacheline that are needed for
303  * the goodness() loop in schedule().
304  */
305     long counter;
306     long nice;
307     unsigned long policy;
308     struct mm_struct *mm;
309     int processor;
```

여기서 nice는 nice value이고, counter는 그 process의 남은 quantum을 나타내는 long 자료형값이다. process의 goodness는 nice값과 counter value에 의해 결정되는 차후 검토할 바이다.

2. scheduler code 분석

가. O(n) Scheduler : linux kernel 2.4.31

1) 개요

새롭게 process를 schedule 할 때, 그 process를 결정하기 위하여 현재 실행중인 모든 process를 순회하여 결정하므로, n개의 process가 실행중일 때 그 결정과정의 시간복잡도가 $O(n)$ 이라는 의미에서 O(n) Scheduler로 불린다. /kernel/sched.c의 코드를 중심으로 분석한다.

2) time slice의 determination

시간은 epoch 단위로 관리되며, 한 epoch에서 각 프로세스들은 자신이 가진 time slice만큼 실행될 수 있다. 이 때 한 epoch에서 프로세스가 자신에게 주어진 time slice를 다 사용하지 않았을 경우, 다음 epoch에서 그 프로세스는 전 epoch에서 사용하지 않았던 시간의 절반만큼의 time slice를 더 갖게 된다.

```
616  if (unlikely(!c)) {
617      struct task_struct *p;
618
619      spin_unlock_irq(&runqueue_lock);
620      read_lock(&tasklist_lock);
621      for_each_task(p)
622          p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
623      read_unlock(&tasklist_lock);
624      spin_lock_irq(&runqueue_lock);
625      goto repeat_schedule;
626  }
```

하이라이트 된 부분은 위에서 실시한 알고리즘을 구현하는 코드이다. 코드에서 counter는 task_struct에 있는 변수로써, 남은 time slice를, shift right 1은 그것을 2로 나누는 것을, NICE_TO_TICKS는 프로세스가 갖는 nice 값에 따른 default time slice를 나타낸다.

3) 우선순위 (goodness) determination

goodness는 scheduler가 순회하면서 조회하는 값으로 가장 높은 goodness 가지는 process가 다음 process로 schedule된다. 즉 process 우선순위의 기준점이다.

```
186  /*
187   * Realtime process, select the first one on the
188   * runqueue (taking priorities within processes
189   * into account).
190   */
191  weight = 1000 + p->rt_priority;
```

이 코드에서 weight는 goodness로써 반환되는 반환값인데, 보다시피 realtime process의 경우 default가 1000이고 거기에 추가적으로 p의 rt_priority가 더해지므로 매우 큰 값이 return됨을 예상할 수 있다.

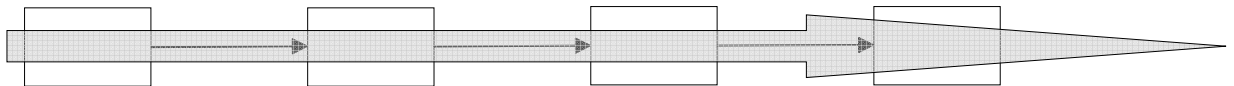

```

157  /*
158  * Non-RT process - normal case first.
159  */
160  if (p->policy == SCHED_OTHER) {
161      /*
162       * Give the process a first-approximation goodness value
163       * according to the number of clock-ticks it has left.
164       *
165       * Don't do any other calculations if the time slice is
166       * over..
167       */
168      weight = p->counter;
169      if (!weight)
170          goto out;
171
172  #ifdef CONFIG_SMP 비활성 전처리기 블록
173  #endif
174
175      /* .. and a slight advantage to the current MM */
176      if (p->mm == this_mm || !p->mm)
177          weight += 1;
178      weight += 20 - p->nice;
179      goto out;
180  }

```

한편, non-realtime process는 위와 같은 방법을 통해 weight가 계산된다. 즉 남은 time slice값(counter)과 당해 process의 nice값에 의해 결정된다.

4) 동작 과정



time complexity: $O(n)$

사각형으로 표시되는 프로세스들의 linked list를 그대로 순회하면서(회색 화살표) 모든 프로세스에 대해 goodness()함수를 호출하여 goodness를 구한 후 최고 높은 goodness를 가진 프로세스를 선택하는 것을 나타낸 그림이다. 이 부분의 코드는 아래와 같다.

```

601  /*
602  * Default process to select..
603  */
604  next = idle_task(this_cpu);
605  c = -1000;
606  list_for_each(tmp, &runqueue_head) {
607      p = list_entry(tmp, struct task_struct, run_list);
608      if (can_schedule(p, this_cpu)) {
609          int weight = goodness(p, this_cpu, prev->active_mm);
610          if (weight > c)
611              c = weight, next = p;
612      }
613  }

```

process를 순회하며 weight를 구하고, weight가 기존 c보다 클 경우 c와 next를 갱신하는 것을 볼 수 있다. 이 부분의 실행 시간은 process가 많아지면 그 수에 선형적으로 비례하여 늘어나므로 process switching에 있어 많은 오버헤드를 야기했다.

나. $O(1)$ Scheduler : linux kernel 2.6.20

1) 개요

프로세스가 가질 수 있는 모든 우선순위마다 그에 해당하는 큐가 있고, 그 큐의 접

근을 통하여 스케줄링할 프로세스의 선택을 피한다. 이를 통하여 스케줄링 시 프로세스 선택작업의 time complexity를 $O(1)$ 로 줄일 수 있었고, 이러한 특징에 따라 $O(1)$ scheduler로 불린다.

2) process 우선순위 determination

실시간 프로세스의 경우는 static priority만을 갖고, 오로지 일반 프로세스들만 dynamic priority를 갖는다. 이하 dynamic priority를 정하는 함수를 본다.

```

727
728 static inline int normal_prio(struct task_struct *p)
729 {
730     int bonus, prio;
731
732     bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;
733
734     prio = p->static_prio - bonus;
735     if (prio < MAX_RT_PRIO)
736         prio = MAX_RT_PRIO;
737     if (prio > MAX_PRIO-1)
738         prio = MAX_PRIO-1;
739     return prio;
740 }

```

그리고, 여기서 CURRENT_BONUS는,

```

#define CURRENT_BONUS(p) \
    (NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / \
     MAX_SLEEP_AVG)

```

로 정의된다. 즉 CURRENT_BONUS는 process의 sleep_avg의 0~MAX_BONUS사이의 mapping이다. 따라서 sleep_avg가 길수록 bonus 값도 커지고, 특히 CURRENT_BONUS(p)가 MAX_BONUS/2보다 크다면 당해 process의 prio는 기존 static_prio보다 더 작은 값(더 중요하게 취급된다는 의미)을 갖게 된다. 반대의 경우도 마찬가지이다.

이와 같이 $O(1)$ Scheduler는 일반 프로세스에게 CPU bounded job에는 penalty를, IO bounded job에는 reward를 주어가면서 fairness를 추구한다.

3) scheduling process

가) schedule될 새로운 프로세스 선택

```

188 /*
189  * These are the runqueue data structures:
190  */
191
192 struct prio_array {
193     unsigned int nr_active;
194     DECLARE_BITMAP(bitmap, MAX_PRIO+1); /* include 1 bit for delimiter */
195     struct list_head queue[MAX_PRIO];
196 };

```

bitmap 자료구조를 통하여 MAX_PRIO개의 run queue를 나타낸다.

이 bitmap 자료구조를 통하여 빠르게 가장 높은 priority에 대응되는 run queue를 찾아낸다. 아래는 그 함수이다.

```

7 /*
8  * Every architecture must define this function. It's the fastest
9  * way of searching a 140-bit bitmap where the first 100 bits are
10  * unlikely to be set. It's guaranteed that at least one of the 140
11  * bits is cleared.
12  */
13 static inline int sched_find_first_bit(const unsigned long *b)

```

그리고 이 함수를 이용하여, 아래와 같은 코드로써 next에 새롭게 schedule할

process를 담는다.

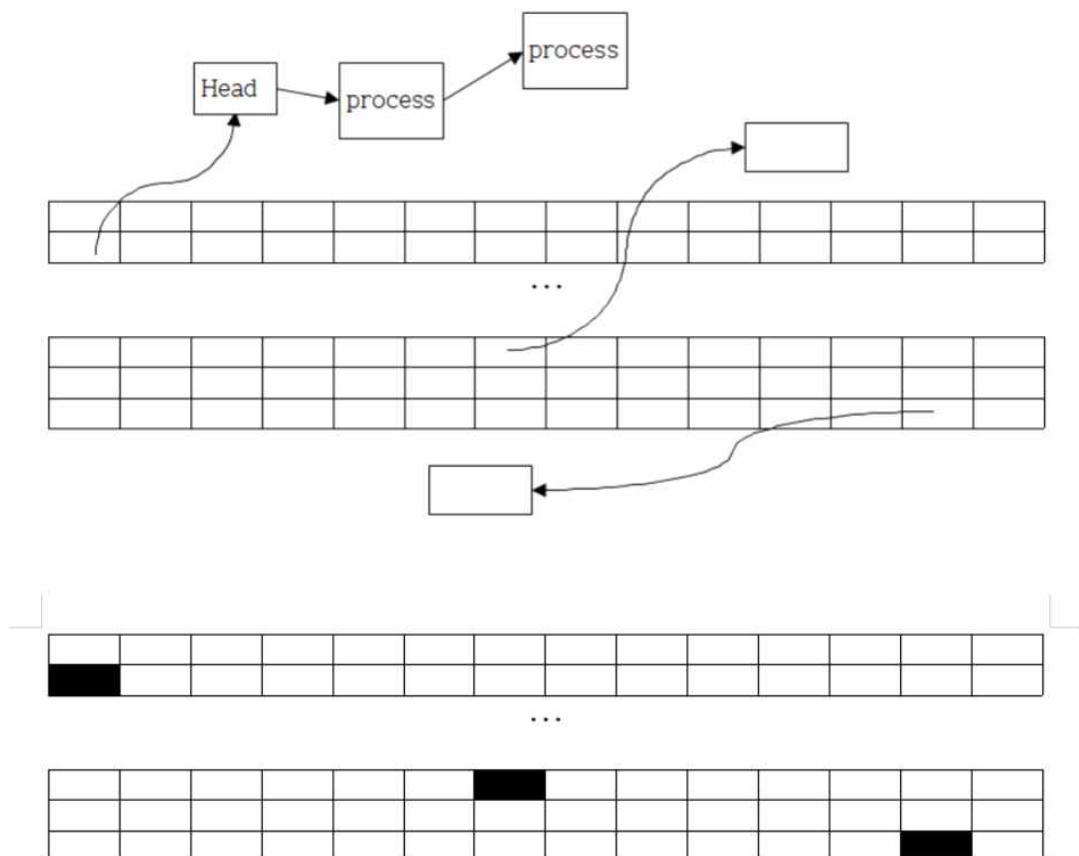
```
3508
3509     idx = sched_find_first_bit(array->bitmap);
3510     queue = array->queue + idx;
3511     next = list_entry(queue->next, struct task_struct, run_list);
3512
```

나) active array와 expired array의 pointer 교환

```
3495
3496     array = rq->active;
3497     if (unlikely(!array->nr_active)) {
3498         /*
3499          * Switch the active and expired arrays.
3500          */
3501         schedstat_inc(rq, sched_switch);
3502         rq->active = rq->expired;
3503         rq->expired = array;
3504         array = rq->active;
3505         rq->expired_timestamp = 0;
3506         rq->best_expired_prio = MAX_PRIO;
3507     }
```

active한 process가 없다면 rq->active와 rq->expired의 포인터를 바꾸어 다
는 획기적인 방법으로서 상수시간만에 새로운 epoch을 시작하게 할 수 있다. 위
코드에서 array는 새롭게 할당된 (즉 이전에는 expired에 있던) run queue 집합
을 가리킨다.

다) 참고도면 - run queue set의 snapshot과 그에 대응되는 bitmap



다. CFS : linux kernel 2.6.24

1) 개요

vruntime을 기준으로 우선순위 큐(Red-Black Tree로 implement)를 구축하여 더 공정한 시간안배를 추구한 scheduler로, Completely Fair Scheduler의 약자이다. 우선순위 큐 구축으로 인하여 다음 실행할 process 선별작업이 $O(\log n)$ 이 되지만, 여러 corner case들을 공정하게 처리해 주기 때문에 latest version도 CFS를 사용한다.

2) process 우선순위(vruntime) && time slice determination

vruntime은 CFS에서 새로이 도입된 개념으로 아래와 같이 정의되는 수치이며, process scheduling 우선순위의 기준이 된다. 즉 run queue에 존재하는 프로세스 중 vruntime이 가장 작은 process가 다음에 schedule될 process로 선택된다.

$$vruntime(\tau, t) = \frac{Weight_0}{Weight\tau} \times ExecutedRuntime(\tau, t)$$

이를 갱신하기 위하여 아래와 같이 일반적 함수로서 calc_delta_mine()함수가 존재하고, (이후 version에서는 time slice의 계산에도 쓰임)

```
751 static unsigned long
752 calc_delta_mine(unsigned long delta_exec, unsigned long weight,
753                struct load_weight *lw)
754 {
755     u64 tmp;
756
757     if (unlikely(!lw->inv_weight))
758         lw->inv_weight = (WMULT_CONST - lw->weight/2) / lw->weight + 1;
759
760     tmp = (u64)delta_exec * weight;
761     /*
762      * Check whether we'd overflow the 64-bit multiplication:
763      */
764     if (unlikely(tmp > WMULT_CONST))
765         tmp = SRR(SRR(tmp, WMULT_SHIFT/2) * lw->inv_weight,
766                  WMULT_SHIFT/2);
767     else
768         tmp = SRR(tmp * lw->inv_weight, WMULT_SHIFT);
769
770     return (unsigned long)min(tmp, (u64)(unsigned long)LONG_MAX);
771 }
```

calc_delta_fair()가 실질적으로 vruntime을 계산하게 된다.

```
772
773 static inline unsigned long
774 calc_delta_fair(unsigned long delta_exec, struct load_weight *lw)
775 {
776     return calc_delta_mine(delta_exec, NICE_0_LOAD, lw);
777 }
```

그리고 이 calc_delta_fair() 함수는 __update_curr()함수에 의해 호출되어 current process의 vruntime을 갱신하게 된다. (line 320)

```

304 static inline void
305 __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
306              unsigned long delta_exec)
307 {
308     unsigned long delta_exec_weighted;
309     u64 vruntime;
310
311     schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));
312
313     curr->sum_exec_runtime += delta_exec;
314     schedstat_add(cfs_rq, exec_clock, delta_exec);
315     delta_exec_weighted = delta_exec;
316     if (unlikely(curr->load.weight != NICE_0_LOAD)) {
317         delta_exec_weighted = calc_delta_fair(delta_exec_weighted,
318                                             &curr->load);
319     }
320     curr->vruntime += delta_exec_weighted;
321
322     /*
323      * maintain cfs_rq->min_vruntime to be a monotonic increasing
324      * value tracking the leftmost vruntime in the tree.
325      */
326     if (first_fair(cfs_rq)) {
327         vruntime = min_vruntime(curr->vruntime,
328                                __pick_next_entity(cfs_rq)->vruntime);
329     } else
330         vruntime = curr->vruntime;
331
332     cfs_rq->min_vruntime =
333         max_vruntime(cfs_rq->min_vruntime, vruntime);
334 }

```

한편, time slice는 다음과 같이 결정된다.

$$TimeSlice_{\tau_i} = \frac{Weight_{\tau_i}}{\sum_{j \in \phi} Weight_{\tau_j}} \times P$$

여기서 P는 다음 __sched_period()함수에 의해 구해지고,

```

245 static u64 __sched_period(unsigned long nr_running)
246 {
247     u64 period = sysctl_sched_latency;
248     unsigned long nr_latency = sched_nr_latency;
249
250     if (unlikely(nr_running > nr_latency)) {
251         period *= nr_running;
252         do_div(period, nr_latency);
253     }
254
255     return period;
256 }

```


sched_slice() 함수는 이렇게 구해진 period를 통하여 time slice를 구한다.

※do_div는 나누기를 정의하는 함수이다.

```

264 static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
265 {
266     u64 slice = __sched_period(cfs_rq->nr_running);
267
268     slice *= se->load.weight;
269     do_div(slice, cfs_rq->load.weight);
270
271     return slice;
272 }

```

3) 동작 과정

real time process들은 기존의 bitmap 자료구조를 이용한 O(1) scheduling 정책을 사용하고, 일반 사용자 process들은 새로히 도입된 CFS를 사용한다. 사용자 process 중심으로 검토한다.

```

230 /* CFS-related fields in a runqueue */
231 struct cfs_rq {
232     struct load_weight load;
233     unsigned long nr_running;
234
235     u64 exec_clock;
236     u64 min_vruntime;
237
238     struct rb_root tasks_timeline;
239     struct rb_node *rb_leftmost;

```

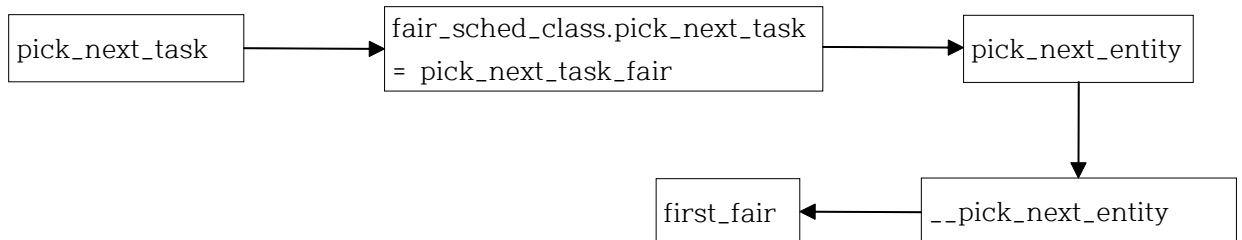
cfs_rq 구조체를 보면 알 수 있든, process를 저장하는 자료구조로는 우선순위 큐의 일종인 red-black tree를 사용하고 있다. 이 rb tree의 key는 각 프로세스의 vruntime이며, rb_leftmost는 가장 작은 vruntime을 갖는 프로세스를 wrapping하는 rb node이다. 그리고 이하 코드는 schedule() function에서 next에 다음 schedule할 process를 선택하는 코드이다.

```

3657 if (unlikely(!rq->nr_running))
3658     idle_balance(cpu, rq);
3659
3660 prev->sched_class->put_prev_task(rq, prev);
3661 next = pick_next_task(rq, prev);

```

이 때 pick_next_task의 함수 호출 관계가 복잡하므로 아래와 같이 도면으로 나타내었다. (화살표는 호출관계)



그리고 마지막 두 단계의 코드는 아래와 같다.

```

196 static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
197 {
198     return rb_entry(first_fair(cfs_rq), struct sched_entity, run_node);
199 }
200

```



```

191 static inline struct rb_node *first_fair(struct cfs_rq *cfs_rq)
192 {
193     return cfs_rq->rb_leftmost;
194 }
195

```

3. 실습 과제 보고서

가. OS, CPU, 메모리 정보

OS는 O(1)의 경우 linux kernel 2.6.20 version을, CFS의 경우는 linux kernel 2.6.24 version을 사용하였고, CPU는 각각 하나씩, 메모리는 각각 4GB씩 할당하였다.

나. uname -a

```

root@aya-laptop:~/Desktop/sharing# uname -a
Linux aya-laptop 2.6.24-32-generic #1 SMP Mon Dec 3 15:48:02 UTC 2012 x86_64 GNU/Linux
root@aya-laptop:~/Desktop/sharing#

```

```

root@aya-laptop:~# uname -a
Linux aya-laptop 2.6.20-17-generic #2 SMP Wed Aug 20 15:14:36 UTC 2008 x86_64 GNU/Linux
root@aya-laptop:~#

```

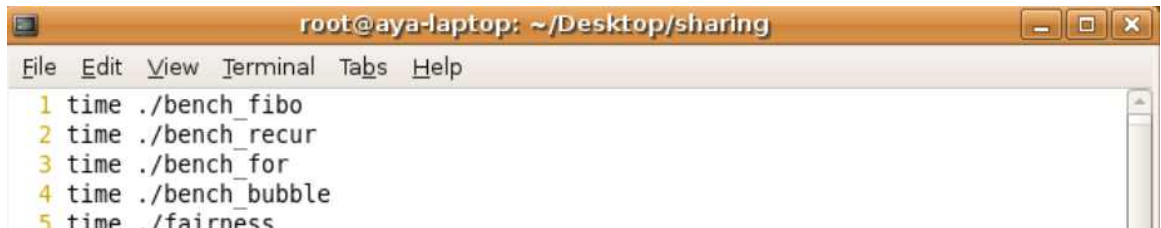
다. Scheduler 비교 보고서

1) performance 비교

가) 실험방법

5개의 benchmark program을 각각의 OS에서 돌렸을 때 시간을 shell에서 time 명령어를 통하여 알아낸 후 서로 비교한다. (shell programming을 사용하여 자동화함)

나) 사용한 shell program : performance.sh



```

root@aya-laptop: ~/Desktop/sharing
File Edit View Terminal Tabs Help
1 time ./bench_fibo
2 time ./bench_recur
3 time ./bench_for
4 time ./bench_bubble
5 time ./fairness

```

다) 사용한 benchmark program의 설명

- (1) bench_fibo : recursive하게 fibonacci number을 계산하는 프로그램으로, n=51
- (2) bench_recur : recursive 3중 fibonacci 계산 프로그램으로, n=40
- (3) bench_for : for loop를 돌며 cnt를 늘려가는 프로그램으로 n=100000000000
- (4) bench_bubble: n개 정수로 이루어지는 무작위 수열을 생성한 뒤, bubble sort를 행하는 프로그램으로, n=200000
- (5) fairness : bench_for과 같은 기능을 하는 자식 process들을 CHILD개 fork하고 이를 wait(&status)로 기다리는 프로그램으로, n=500000000, CHILD=100

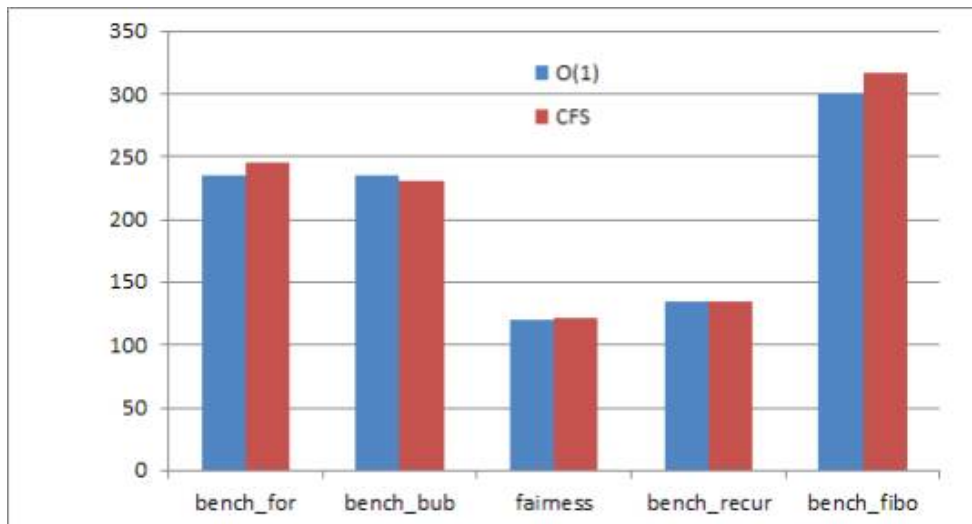
※ 참고: fairness.c의 코드

```

6 #define NUM 500000000
7 #define CHILD 100
8
9 void time_consuming(){
10 //     printf("time_consuming called\n");
11     long long n = NUM;
12     long long i, j=0;
13     for (i=0;i<n;i++){
14         j++;
15     }
16 }
17
18 int main(void){
19     int c = CHILD;
20     int status;
21     int i, pid;
22     int cnt = 0;
23     for (i=0;i<c;i++){
24         pid = fork();
25         if (pid == 0){
26             time_consuming();
27             break;
28         }
29     }
30     if (pid > 0){
31         printf("waiting\n");
32         while(wait(&status)>0){
33             cnt++;
34             printf("%d ", cnt);
35         }
36         printf("\nwaiting end\n");
37     }
38 }

```

라) 실험 결과



세로축은 실행 시간(단위: sec)을 나타낸다.

마) 결과 분석

유의미한 정도의 시간 차이를 보이진 않으나, 대체적으로 O(1) scheduler가 근소하게 좋은 성능을 보였다. 생각건대, scheduling 과정에서 CFS는 red-black tree를 활용하여 다음 process를 select 하므로 그 시간복잡도가 $O(\log n)$ 인데 반하여 O(1) scheduler는 rq에 대응하는 bitmap에 고도로 최적화된 매크로 함수를 통하여 빠르게 접근하여 다음 process를 select 하는, $O(1)$ 시간복잡도를 갖는 알고리즘을 쓰므로 CFS보다 근소하게나마 process scheduling overhead가 적게 걸리므로(big-O notation을 쓰는 시간복잡도는 분명히 유의미한 차이가 있으나 실제

돌고 있는 process의 개수는 100개 남짓으로, 큰 차이가 나진 않는다.) 위와 같은 결과가 나왔을 것이다.

2) fairness 비교

가) 실험 방법

A_bench와 B_bench, 그리고 위에서 소개한 fairness를 동시에 실행시키고, A_bench와 B_bench의 cpu 점유시간을 작성한 커널 모듈을 단위시간(1초)마다 조회하여 각각 얻고, 이를 텍스트 파일에 기록한다. 이를 자동화하기 위하여 shell programming을 하였다.

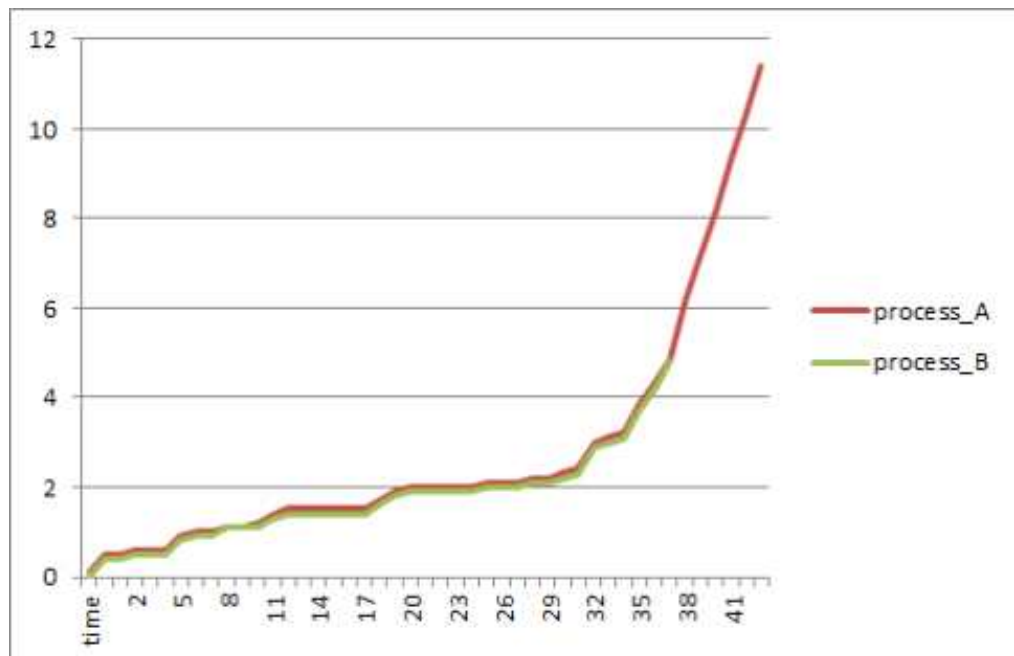
나) 사용한 shell program - benchmark_shell.sh

```
1 #!/bin/bash
2 number=0
3 rm ./bench_report.txt
4 insmod ./hw1_module.ko
5 ./fairness & ./A_mybench & ./B_mybench &
6 while [ "$number" -lt 100 ]
7 do
8     echo -n 'sec: ' >> bench_report.txt
9     echo "$number" >> bench_report.txt
10    echo -n 'A cpu time: ' >> bench_report.txt
11    grep 'A_mybench' /proc/hw1 | egrep -o '[0-9]+\.[0-9]+' | sed '2q;d' >> bench_report.txt
12    echo -n 'B cpu time: ' >> bench_report.txt
13    grep 'B_mybench' /proc/hw1 | egrep -o '[0-9]+\.[0-9]+' | sed '2q;d' >> bench_report.txt
14    echo >> bench_report.txt
15    sleep 1s
16    ((number += 1))
17 done
18 rmmod hw1_module
19 echo 'fairness test end'
```

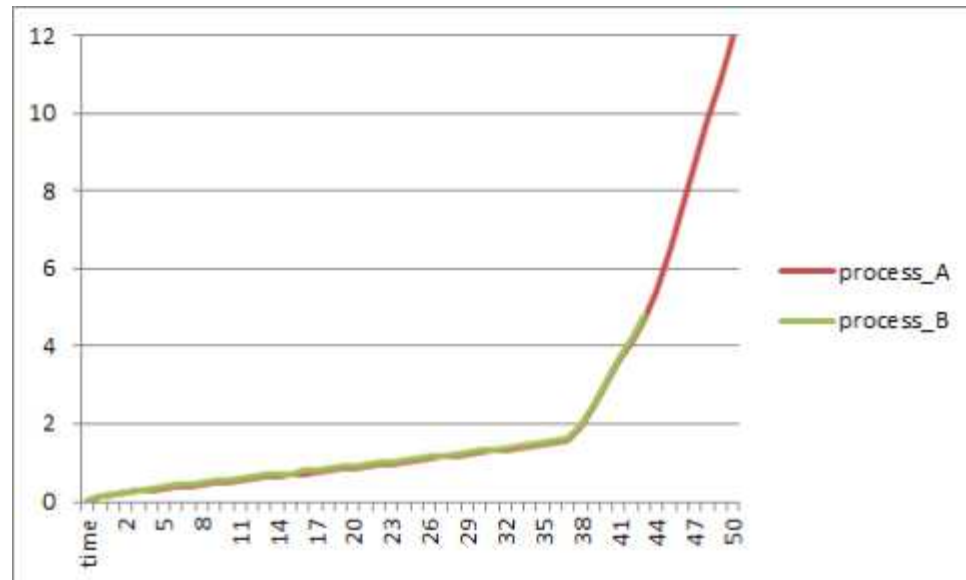
다) A_mybench, B_mybench의 내용 : 둘 모두 bench_for이고, 각각
n=5000000000, n=2000000000

라) 실험 결과

(1) O(1) scheduler



(2) CFS



(3) $fairness = avg(|A_t - B_t|)$

(가) O(1) : 0.100718

(나) CFS : 0.044844

마) 결과 분석

fairness는 유의미하게 CFS가 좋게 나왔고(위에서 정의한 *fairness*는 값이 작을수록 더 공평한, 다시 말해 더 좋은 scheduling이 있었다고 볼 수 있음), 그래프의 개형도 유의미한 차이가 있었다.

생각건대, 앞에서 소개한, 100개의 child를 fork하는 fairness program을 백 그라운드에서 실행시켰으므로, O(1) scheduler의 특정 nice value(default nice value는 0이므로 0)에 대응하는 run queue(아마 120번째)는 **과포화**에 이르렀을 것이다. 비록 dynamic priority가 적용되어 조금 완화 되겠지만, 그럼에도 불구하고 거의 **round robin**방식과 유사하게 동작 할 것이고, 따라서 O(1) scheduler 그래프의 11~17sec과 같은, cpu 시간을 전혀 할당받지 못하는 plateau period가 나타난다. 자연히 이러한 현상 때문에 $avg(|A_t - B_t|)$ 값도 높게 나오게 된다.

한편, CFS는 유저 process scheduling 정책으로 CFS를 사용하고, 이는 process를 **vruntime**을 **key**로 갖는 **red-black tree**로 관리하기 때문에 위와 같은 특정 run queue 과포화로 인한 round robin化의 우려가 없다. 실험 결과도 이를 뒷받침하듯 매 초 꾸준히 양 task 모두 cpu 시간을 할당받아 O(1)과 같은 plateau period가 나타나지 않았고, $avg(|A_t - B_t|)$ 도 작게 나왔다.

라. 커널 모듈 작성 보고서

1) seq_file

원래 `proc_dir_entry`가 읽힐 때는 `proc_dir_entry->read_proc`함수가 불리면서, `read_proc`함수가 인수로 받는 `char*` buffer에 담긴 문자열이 터미널에 출력된다. 그러나 이러한 출력방식은 불편한 점이 있어, 출력의 새로운 interface로써 `seq_file`이 제시되었는데, 이러한 `seq_file`을 활용하면 `seq_printf()`함수를 통하여 일반 어플리케이션에서 `printf`를 쓰듯 터미널에 표시할 수 있게 된다.

`seq_file`은 iterator로도 사용이 가능하나, 본 과제에서는 iterator 기능은 사용하지 않고, `seq_file`의 `show`만을 사용하는 방식을 취한다. - `single_open()`함수

2) 주요 함수, 구조체

가) `my_show()` : 본인이 만든 kernel module에 의해 만들어진 proc entry가 읽힐 때 호출되는 함수로써, 터미널에 표시할 내용들을 `seq_printf()`로 코딩하였다.

```
25 static int my_show(struct seq_file *s, void *unused){
26     int proc_cnt = 0;
27     int constant = 1000/HZ;
28     int system_uptime = (jiffies-INITIAL_JIFFIES)*constant;
29     int start_time, start_time_ms, total_time, total_time_ms, user_time, user_time_ms, kernel_time, kernel_time_ms;
30     int user_time_mil, kernel_time_mil, total_time_mil;
31     printk(KERN_INFO "my_show called\n");
32     print_bar(s);
33     seq_printf(s, "CURRENT SYSTEM INFORMATION >\n");
34     for_each_process(task){
35         proc_cnt++;
36     }
```

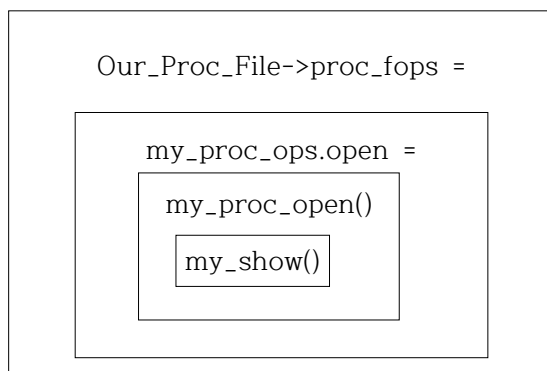
my_show()의 일부

나) `my_proc_open()` : `my_show()`를 인수로 받는 `single_open()`이라는 커널 함수의 리턴값을 리턴하는 함수이다. `return single_open(file, my_show, NULL);`

다) `my_proc_ops` : 구조체로서, 구조체 내부 open 함수포인터에 `my_proc_open()`함수가 할당된다.

라) `Our_Proc_File` : 구조체로서, 이 프로그램이 만드는 proc file entity를 나타낸다. 그 구조체 내부 `proc_fops`에 `my_proc_ops` 구조체의 주소값이 할당된다.

3) 프로그램 구조



이와 같은 계층구조로서, 결국 사용자 정의 함수인 `my_show()`함수는 `Our_Proc_File`에서 접근 가능하게 된다. 따라서 `Our_Proc_File`이 가리키는 proc file(여기에서는 `/proc/hw1`)이 읽혔을 때, 정상적으로 `my_show()`함수가 call되게 된다.

4) my_show() 내부에서 접근한 kernel 내 자료구조들 - 전술함

5) 문제점 및 해결

가) 관점의 문제

커널 모듈 프로그래밍의 경우, 커널 내부에 있어서 커널 내부 자료구조들에 자유롭게 접근할 수 있음을 잊고, 일반 프로그램과 같이 생각하여 my_show()내부의 코딩에 어려움이 있었다. 예컨대 처음에는 모든 process 개수를 세는데, /proc내의 '숫자로 된' 폴더들의 개수를 세면된다고 생각하였다.

그러나 곧 for_each_process를 통하여 모든 process의 task_struct에 자유롭게 접근할 수 있음을 알게 되었고, 커널 내 자료구조들을 활용하여 보다 간결하게 원하고자 하는 바를 알아낼 수 있었다.

나) 커널 자료구조에 익숙치 않음에 의한 문제

그러나, '어떤' 커널 자료구조가 내가 원하는 정보를 담고 있는지 알아내는 과정이 힘들었다. task_struct 내의 start_time, stime, utime같은 경우에는 비교적 쉽게 찾았으나, 부팅 후 milisecond로 시스템 시간을 알아내기 위해 필요한 커널 자료구조가 jiffies와 INITIAL_JIFFIES임은 알아내기가 어려웠다.

장기간의 intensive한 구글링과 동기들과의 협업을 통하여 이를 해결하였다.