

운영체제 2차 과제

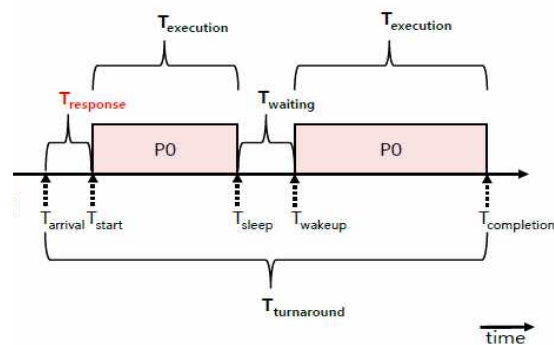
<Linux process scheduler의 이해와 구현>

정보대학 컴퓨터학과
2014210007 서영남

1. 구현한 스케줄러 설계 구조 및 동작 방식

a) Process scheduling

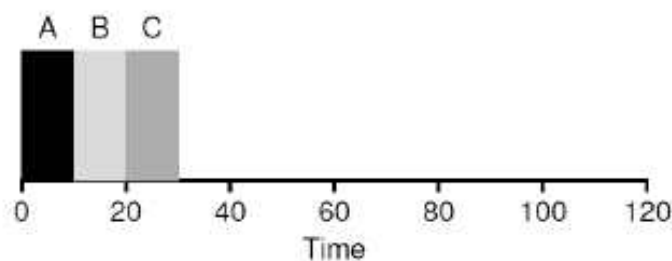
프로세스 스케줄링이란, 다수의 프로세스가 존재하는 경우, OS의 프로세스 관리 목표(scheduling goal)를 극대화하기 위해서 프로세스를 스케줄링하는 것이다. 각각의 정책마다 극대화하고자 하는 목적이 있다. 예를 들어, 우선순위에 따르거나 평균 실행시간을 최소화하는 등의 목표를 가지고 스케줄링 정책을 정한다. 이를 정확히 비교하기 위해서는 여러 개념에 대한 정의가 필요할 것이다.



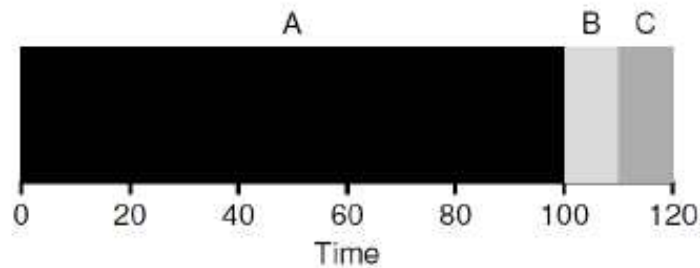
먼저, Turnaround time(이하 TAT)은 어떤 작업이 제출된 후에, 완료되기까지 걸린 시간을 의미한다. 즉, 프로세스가 생성된 후 실행을 끝마치기까지 걸린 시간이다. Response time이란, 어떤 작업이 제출된 후에, 시작까지 걸린 시간을 의미한다. 즉, 프로세스가 생성된 후, 처음 실행하기까지 걸린 시간이다. Priority란, 각 프로세스마다 주어진 우선순위를 말한다. 즉, 원하는 결과에 따라 프로세스에는 우선순위가 할당되어있을 수 있으므로 각각의 프로세스 사이에 우선순위 서열이 있을 수 있다.

이 3가지 기준 이외에도 Fairness, I/O 작업 등 고려할 만한 요소들이 있지만 여기서는 이 3가지 기준을 중심으로 설명을 이어나갈 것이다.

b) FIFO

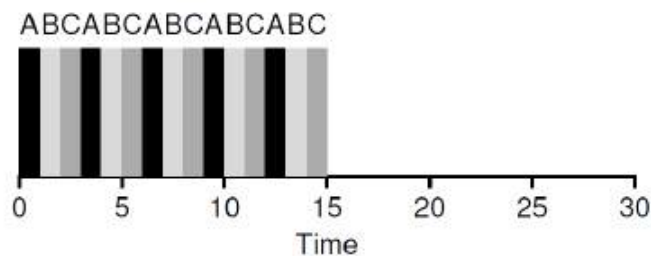


FIFO(First In First Out) 정책은 말 그대로, 먼저 들어온 프로세스가 먼저 나가는 방식이다. 위에서 A, B, C 순으로 프로세스가 실행되었음을 알 수 있는데, FIFO 스케줄링 정책이 적용된 것이라면, 들어간 순서 또한 A, B, C 순서이다. FIFO 방식은 가장 단순한 방식인 만큼 구현하기 쉽다는 장점이 있지만 모든 프로세스들이 늘 같은 실행시간을 가지기는 어렵다는 점에서 여러 문제를 가지고 있다.



위의 예시처럼 먼저 들어온 프로세스가 가장 긴 실행시간을 가질 경우, 전체적으로 평균 비용이 증가하므로 효율적이지 못한 정책이 된다. 즉, 위의 경우에서 평균 TAT를 계산하면, $(100+110+120)/3 = 110\text{s}$ 이다. 만약 프로세스에 들어간 순서가 B->C->A라면, 평균 TAT는 $(10+20+120)/3 = 50\text{s}$ 이다. 즉, 들어온 순서에 따라 평균 TAT가 크게 차이날 수 있고 긴 실행시간을 가진 프로세스가 짧은 것보다 먼저 들어오면 매우 비효율적인 정책이 된다.

c) RR



RR(Round Robin) 정책은 실행 중인 작업을 다 마치지 않았어도, 주어진 시간(time slice)이 만료되면 다른 프로세스로 전환하는 스케줄링 방식이다. 프로세스들 사이에 우선순위를 두지 않고, 시간 단위(time slice) 동안 수행한 프로세스는 준비 큐(waiting queue)의 끝으로 밀려나게 된다. RR 정책의 가장 큰 장점은 평균 Response time을 줄일 수 있다는 점이다. 위의 경우의 Response time은 $(0+1+2)/3 = 1\text{s}$ 이다. 만약 위와 같은 프로세스를 FIFO 정책으로 수행했다면, $(0+5+10)/3 = 5\text{s}$ 이다. 이렇게 Response time이 짧아지는 장점이 있는 만큼 실시간 시스템에 적용되기 유리하다는 특징이 있다. 또한 모든 프로세스를 짧은 response time 내에 실행하기 때문에 fairness(공평성) 측면에서도 장점을 보이는 정책이다.

그렇다면, Time slice가 짧을수록 Response time이 줄어들기 때문에 좋은 정책이라고 할 수 있는가? 그것은 아니다. 왜냐하면 실행하는 프로세스를 전환하는 context switching을 하는 데 비용이 들기 때문이다. 따라서 RR 정책에서는 time slice의 길이를 설정하는 것이 성능에 결정적인 영향을 미친다고 할 수 있다.

d) WRR

WRR(Weighted Round Robin) 정책은 RR 정책에 Priority를 추가로 고려하는 정책을 말한다. 각 프로세스의 우선순위에 따라 서로 다른 Queue에 분류 및 할당하고 우선순위가 높은 Queue에 있는 프로세스들부터 RR 정책에 따라 실행하는 스케줄링 정책이다. 또는 우선순위에 따라 프로세스를 서로 다른 Queue에 넣지 않더라도 실행 비율이 우선순위에 따른 가중치 비율을 따르도록 할 수 있다.

2. 구현한 스케줄러 코드

a) FIFO

I) enqueue_task_fifo()

```
57 static void
58 enqueue_task_fifo(struct rq *rq, struct task_struct *p, int flags)
59 {
60     Node *newNode = (Node*)kalloc(sizeof(Node), GFP_KERNEL);
61     newNode->next = NULL;
62     newNode->p = p;
63     if (QIsEmpty(pq)) {
64         pq.front = newNode;
65         pq.rear = newNode;
66     }
67     else {
68         pq.rear->next = newNode;
69         pq.rear = newNode;
70     }
71     printk("MYMOD: enqueue_task_fifo CALLED\n");
72     add_nr_running(rq, 1);
73 }
```

이 함수는 프로세스가 생성되어 fork될 때 실행되는 함수이다. FIFO를 구현하기 위해 연결 리스트를 기반으로 한 queue를 구현하였다. Queue에 새 노드를 만들어서 주어진 프로세스 p를 할당한 후, queue의 rear 부분에 넣는다. 이 방식을 통해 n개의 프로세스들이 queue에 삽입될 수 있다.

II) dequeue_task_fifo()

```
75 static void
76 dequeue_task_fifo(struct rq *rq, struct task_struct *p, int flags)
77 {
78     Node *temp, *temp2;
79     temp = pq.front;
80     temp2 = NULL;
81
82     while (p->pid != temp->p->pid) {
83         temp2 = temp;
84         temp = temp->next;
85     }
86     if (temp->next == NULL) {
87         if (temp2 != NULL) {
88             pq.rear = temp2;
89             temp2->next = NULL;
90             kfree(temp);
91         }
92         else {
93             pq.front = NULL;
94             pq.rear = NULL;
95             kfree(temp);
96         }
97     }
98     else {
99         if (temp2 != NULL) {
100             temp2->next = temp->next;
101             kfree(temp);
102         }
103         else {
104             pq.front = temp->next;
105             kfree(temp);
106         }
107     }
108
109     printk("MYMOD: dequeue_task_fifo CALLED\n");
110     sub_nr_running(rq, 1);
111 }
```

dequeue_task_fifo() 함수는 queue에 있는 프로세스를 꺼내 삭제하는 함수이다. 주어진 프로세스 p의 pid와 같은 값을 가진 프로세스를 찾는 작업을 거친 후에, 그 노드가

front, rear, 그 사이에 위치했을 때의 각각의 경우와 꺼내고자 하는 노드의 양옆 노드의 Null 여부에 따라 다르게 코드를 짰다.

III) pick_next_task_fifo()

```
37 static struct task_struct *
38 pick_next_task_fifo(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
39 {
40     //printk("MYMOD: pick_next_task_fifo CALLED");
41
42     if (pq.front->p != NULL)
43         p = pq.front->p;
44     else
45         p = fair_sched_class.pick_next_task(rq, prev, rf);
46
47     if (p != NULL)
48     {
49         put_prev_task(rq, prev);
50         printk("MYMOD: pick_next_task_fifo [prev: %u] [next: %u]\n", prev->pid, p->pid);
51         return p;
52     }
53
54     return NULL;
55 }
56
```

pick_next_task_fifo() 함수는 다음으로 스케줄링 될 프로세스를 고르는 함수이다. Queue의 front 노드에 위치한 프로세스를 꺼내서 FIFO 정책에 따라 수행한다. 만약 해당 노드가 비었다면, mysched_class를 삽입하기 전 이용하던 fair class에 따라 수행한다.

b) RR

1) task_tick_fifo()

```
124 static void task_tick_fifo(struct rq *rq, struct task_struct *curr, int queued)
125 {
126     Node *temp3;
127
128     if (pq.front != pq.rear) {
129         pq.rear->next = pq.front;
130         temp3 = pq.front->next;
131         pq.rear = pq.front;
132         pq.front->next = NULL;
133         pq.front = temp3;
134     }
135     //printk("MYMOD: task_tick_fifo CALLED\n");
136 }
```

task_tick_fifo() 함수는 time_tick이 될 때마다 실행되는 함수이다. 따라서 RR 정책을 수행하기 위해서 이 함수가 실행될 때마다 queue의 front 노드에 있는 프로세스를 rear 노드로 옮겨주는 작업을 수행하도록 했다. 왜냐하면 pick_next_task_fifo() 함수에서 다음에 실행할 프로세스를 front 노드에서 추출하기 때문이다. 실행한 리눅스 환경에서 task_tick_fifo() 함수가 4ms마다 호출되었기 때문에(cat .config |grep CONFIG_HZ를 통해 확인함) 여기서의 time slice는 4ms라고 할 수 있다.

이 함수를 제외한 나머지 부분은 1)의 FIFO와 같다.

c) Weighted Round Robin

I) enqueue_task_fifo()

```

63 static void
64 enqueue_task_fifo(struct rq *rq, struct task_struct *p, int flags)
65 {
66     Node *newNode = (Node*)kmallocc(sizeof(Node), GFP_KERNEL);
67     newNode->next = NULL;
68     newNode->p = p;
69     if (QIsEmpty(pq)) {
70         pq.front = newNode;
71         pq.rear = newNode;
72     }
73     else {
74         pq.rear->next = newNode;
75         pq.rear = newNode;
76     }
77     weight[(p->pid) % NUM] = ((p->pid) % NUM) + 1;
78     printk("MYMOD: enqueue_task_fifo CALLED\n");
79     add_nr_running(rq, 1);
80 }
81

```

Weighted Round Robin을 구현하기 위해서는 각 프로세스의 우선순위 서열에 따른 가중치가 주어져야 한다. 각 프로세스에 대해 가중치를 부여하려면 전체 프로세스의 개수를 알아야 해서 여기서는 원래 sample 코드에서 주어진 대로 프로세스 개수를 5개로 정했다. 그리고 각 프로세스의 pid는 연속적인 수를 가지기 때문에 pid를 5(전체 프로세스 개수)로 나눈 나머지를 weight 배열의 index로 정하고 나머지에 1씩 더한 값을 가중치를 의미하는 값으로 넣었다. 이를 단순한 식으로 정의한다면, $weight[pid \% 5] = \{1, 2, 3, 4, 5\}$ 와 같이 표현될 것이다.

II) task_tick_fifo()

```

132 static void task_tick_fifo(struct rq *rq, struct task_struct *curr, int queued)
133 {
134     Node *temp3;
135     int k;
136
137     if (pq.front != temp4) {
138         temp4 = pq.front;
139         k = weight[(pq.front->p->pid) % 5];
140         count = k;
141     }
142     if (count == 0) {
143         if (pq.front != pq.rear) {
144             pq.rear->next = pq.front;
145             temp3 = pq.front->next;
146             pq.rear = pq.front;
147             pq.front->next = NULL;
148             pq.front = temp3;
149         }
150     }
151     else
152         count--;
153     //printk("MYMOD: task_tick_fifo CALLED\n");
154 }

```

Round Robin 스케줄링에서처럼 task_tick_fifo() 함수를 고치는 방식을 이용했다. 다음 프로세스가 될 queue의 front 노드에 있는 프로세스의 가중치를 확인한 후, 그 프로세스가 (가중치 * time slice)의 시간만큼 수행될 수 있도록 count라는 변수를 이용했다. 그 시간이 지나면 원래 Round Robin처럼 front 노드를 rear 노드가 되도록 옮김으로써 다음 프로세스를 변경하도록 했다.

3. 스케줄러 실행 결과

a) 실행 순서

I) FIFO

```
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched/mysched
[PID:2887] Original sched_policy: 0
[PID:2887] Changed sched_policy: 7
[PID:2891] Original sched_policy: 0
[PID:2891] Changed sched_policy: 7
[PID:2890] Original sched_policy: 0
[PID:2890] Changed sched_policy: 7
[PID:2889] Original sched_policy: 0
[PID:2889] Changed sched_policy: 7
[PID:2888] Original sched_policy: 0
[PID:2888] Changed sched_policy: 7
[PID:2887] Count = 00
[PID:2887] Count = 01
[PID:2887] Count = 02
[PID:2887] Count = 03
[PID:2887] Count = 04
[PID:2887] Count = 05
[PID:2887] Count = 06
[PID:2887] Count = 07
[PID:2887] Count = 08
[PID:2887] Count = 09
[PID:2887] Time interval: 4.234614709
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched/mysched$ [PID:2891] Count = 00
[PID:2891] Count = 01
[PID:2891] Count = 02
[PID:2891] Count = 03
[PID:2891] Count = 04
[PID:2891] Count = 05
[PID:2891] Count = 06
[PID:2891] Count = 07
[PID:2891] Count = 08
[PID:2891] Count = 09
[PID:2891] Time interval: 8.446554101
```

```
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched
[PID:2890] Count = 00
[PID:2890] Count = 01
[PID:2890] Count = 02
[PID:2890] Count = 03
[PID:2890] Count = 04
[PID:2890] Count = 05
[PID:2890] Count = 06
[PID:2890] Count = 07
[PID:2890] Count = 08
[PID:2890] Count = 09
[PID:2890] Time interval: 12.643138444
[PID:2889] Count = 00
[PID:2889] Count = 01
[PID:2889] Count = 02
[PID:2889] Count = 03
[PID:2889] Count = 04
[PID:2889] Count = 05
[PID:2889] Count = 06
[PID:2889] Count = 07
[PID:2889] Count = 08
[PID:2889] Count = 09
[PID:2889] Time interval: 16.930047413
[PID:2888] Count = 00
[PID:2888] Count = 01
[PID:2888] Count = 02
[PID:2888] Count = 03
[PID:2888] Count = 04
[PID:2888] Count = 05
[PID:2888] Count = 06
[PID:2888] Count = 07
[PID:2888] Count = 08
[PID:2888] Count = 09
[PID:2888] Time interval: 21.155960107
```

FIFO 스케줄링 정책에 따라 프로세스들이 들어간 순서대로 실행되는 것을 확인할 수 있다.

II) RR

```
[PID:4470] Original sched_policy: 0
[PID:4472] Original sched_policy: 0
[PID:4472] Changed sched_policy: 7
[PID:4470] Changed sched_policy: 7
[PID:4473] Original sched_policy: 0
[PID:4473] Changed sched_policy: 7
[PID:4474] Original sched_policy: 0
[PID:4474] Changed sched_policy: 7
[PID:4471] Original sched_policy: 0
[PID:4471] Changed sched_policy: 7
[PID:4471] Count = 00
[PID:4474] Count = 00
[PID:4473] Count = 00
[PID:4472] Count = 00
[PID:4470] Count = 00
[PID:4471] Count = 01
[PID:4473] Count = 01
[PID:4474] Count = 01
[PID:4472] Count = 01
[PID:4470] Count = 01
[PID:4471] Count = 02
[PID:4473] Count = 02
[PID:4474] Count = 02
[PID:4472] Count = 02
[PID:4470] Count = 02
[PID:4471] Count = 03
[PID:4473] Count = 03
[PID:4474] Count = 03
[PID:4472] Count = 03
[PID:4470] Count = 03
[PID:4471] Count = 04
[PID:4473] Count = 04
[PID:4474] Count = 04
[PID:4472] Count = 04
[PID:4470] Count = 04
[PID:4471] Count = 05
[PID:4473] Count = 05
[PID:4474] Count = 05
[PID:4472] Count = 05
[PID:4470] Count = 05
[PID:4471] Count = 06
[PID:4473] Count = 06
[PID:4474] Count = 06
[PID:4472] Count = 06
[PID:4470] Count = 06
[PID:4471] Count = 07
[PID:4473] Count = 07
[PID:4474] Count = 07
[PID:4472] Count = 07
```

```
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched/mysched
[PID:4470] Count = 00
[PID:4471] Count = 01
[PID:4473] Count = 01
[PID:4474] Count = 01
[PID:4472] Count = 01
[PID:4470] Count = 01
[PID:4471] Count = 02
[PID:4473] Count = 02
[PID:4474] Count = 02
[PID:4472] Count = 02
[PID:4470] Count = 02
[PID:4471] Count = 03
[PID:4473] Count = 03
[PID:4474] Count = 03
[PID:4472] Count = 03
[PID:4470] Count = 03
[PID:4471] Count = 04
[PID:4473] Count = 04
[PID:4474] Count = 04
[PID:4472] Count = 04
[PID:4470] Count = 04
[PID:4471] Count = 05
[PID:4473] Count = 05
[PID:4474] Count = 05
[PID:4472] Count = 05
[PID:4470] Count = 05
[PID:4471] Count = 06
[PID:4473] Count = 06
[PID:4474] Count = 06
[PID:4472] Count = 06
[PID:4470] Count = 06
[PID:4471] Count = 07
[PID:4473] Count = 07
[PID:4474] Count = 07
[PID:4472] Count = 07
[PID:4470] Count = 07
[PID:4471] Count = 08
[PID:4473] Count = 08
[PID:4474] Count = 08
[PID:4472] Count = 08
[PID:4470] Count = 08
[PID:4471] Count = 09
[PID:4471] Time interval: 20.339873869
[PID:4473] Count = 09
[PID:4473] Time interval: 20.341953667
[PID:4472] Count = 09
[PID:4472] Time interval: 20.427188666
[PID:4474] Count = 09
[PID:4474] Time interval: 20.510645266
[PID:4470] Count = 09
[PID:4470] Time interval: 20.523681760
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched/mysched$
```

RR 스케줄링 정책에 따라 프로세스 5개가 번갈아가며 실행됨을 확인할 수 있다.

III) Weighted Round Robin

```
[PID:10559] Original sched_policy: 0
[PID:10559] Changed sched_policy: 7
[PID:10563] Original sched_policy: 0
[PID:10563] Changed sched_policy: 7
[PID:10562] Original sched_policy: 0
[PID:10562] Changed sched_policy: 7
[PID:10561] Original sched_policy: 0
[PID:10561] Changed sched_policy: 7
[PID:10560] Original sched_policy: 0
[PID:10560] Changed sched_policy: 7
[PID:10559] Count = 00
[PID:10563] Count = 00
[PID:10562] Count = 00
[PID:10561] Count = 00
[PID:10559] Count = 01
[PID:10563] Count = 01
[PID:10560] Count = 00
[PID:10559] Count = 02
[PID:10562] Count = 01
[PID:10563] Count = 02
[PID:10559] Count = 03
[PID:10561] Count = 01
[PID:10562] Count = 02
[PID:10563] Count = 03
[PID:10559] Count = 04
[PID:10563] Count = 04
[PID:10559] Count = 05
[PID:10562] Count = 03
[PID:10561] Count = 02
[PID:10560] Count = 01
[PID:10559] Count = 06
[PID:10563] Count = 05
[PID:10562] Count = 04
[PID:10559] Count = 07
[PID:10561] Count = 03
[PID:10563] Count = 06
[PID:10559] Count = 08
[PID:10562] Count = 05
[PID:10560] Count = 02
[PID:10563] Count = 07
[PID:10561] Count = 04
[PID:10559] Count = 09
[PID:10559] Time interval: 14.5696203
```

```
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched/mysched$ [PID:10562] Count = 06
[PID:10563] Count = 08
[PID:10563] Count = 09
[PID:10563] Time interval: 16.515110411
[PID:10562] Count = 07
[PID:10561] Count = 05
[PID:10560] Count = 03
[PID:10562] Count = 08
[PID:10561] Count = 06
[PID:10562] Count = 09
[PID:10562] Time interval: 18.502673821
[PID:10560] Count = 04
[PID:10561] Count = 07
[PID:10561] Count = 08
[PID:10560] Count = 05
[PID:10561] Count = 09
[PID:10561] Time interval: 20.312761034
[PID:10560] Count = 06
[PID:10560] Count = 07
[PID:10560] Count = 08
[PID:10560] Count = 09
[PID:10560] Time interval: 21.764438949
```

각 프로세스의 pid를 전체 프로세스 개수로 나눈 나머지를 기준으로 가중치를 잡았기 때문에 나머지가 큰 pid를 가진 프로세스가 가장 큰 가중치를 가졌음을 알 수 있다. 실행 순서를 보면, pid가 10559, 10563, 10562, 10561, 10560 순으로 실행되었기 때문에 가중치가 큰 프로세스가 먼저 실행되어 먼저 끝났음을 알 수 있다.

b) 각 스케줄러 별 TAT, Response time

	TAT(s)	Response time(s)
FIFO	$(4.2+8.4+12.6+16.9+21.2) / 5 = 12.7$	$(0+4.2+8.4+12.6+16.9) / 5 = 8.42$
RR	$(20.3+20.3+20.4+20.5+20.5) / 5 = 20.4$	$(0+0.01+0.02+0.03+0.04) / 5 = 0.02$
WRR	$(14.6+16.5+18.5+20.3+21.8) / 5 = 18.3$	$(0+0.01+0.02+0.03+0.06) / 5 = 0.024$

여기서 TAT, Response time을 계산하기 위해서는 몇 가지 가정이 필요하다. 일단, 프로세스 5개가 동시에 생성될 수는 없지만, 그 시간의 차이가 거의 없기 때문에 동시에 생성되었다고 생각하고 계산을 했다. 그리고 프로세스의 각 count 하나만큼의 시간 길이는 한 프로세스가 총 걸린 시간을 나타내는 time interval들 사이의 count 개수를 통해서 구했다. 여기서 구한 것을 Response time을 계산할 때 각 프로세스가 실행 시작하는 데 걸린 시간을 구할 때 사용하였다.

4. 실행 결과 분석 및 장단점

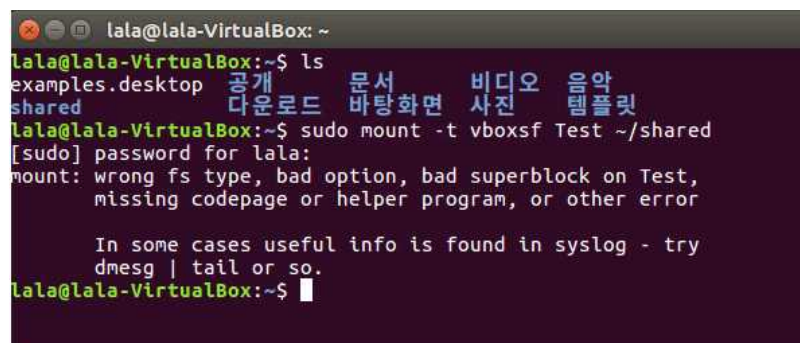
스케줄링 정책을 Weighted Round Robin으로 정한 것은 기존의 FIFO, RR에서 고려할 수 없었던 우선순위를 고려하기 위함이다. FIFO, RR에서는 우선순위를 전혀 고려하지 않기 때문에 우선순위에 맞게 프로세스를 실행하는 것이 불가능하다. 그러나 Weighted Round Robin에서는 각 프로세스마다 1~5의 가중치를 부여함으로써 가중치가 높은 프로세스가 더 많은 비율로 실행되도록 하였다. 결과적으로 3-a)에서 확인할 수 있듯이, 가중치가 큰 프로세스가 먼저 실행 완료되었다.

구현한 Weighted Round Robin에 맞게 완벽하게 1~5의 비율로 프로세스들이 실행 완료되지 않은 것은 time slice 한 번마다 프로세스가 실행되는 것이 아니기 때문인 것으로 보인다. 즉, 구현한 것이 정확하게 들어맞기 위해서는 time slice마다(task_tick_fifo())가 호출될 때마다 pick_next_task_fifo()가 호출되어야 하는데 그렇지 않기 때문에 가중치만큼 프로세스가 실행되지 않는 것이다. 그렇지만 부여한 우선순위에 따라서 프로세스가 순서대로 끝나는 것은 task_tick_fifo() 함수에 의해 queue의 front 노드에 가중치가 높은 프로세스가 들어 있을 확률이 높아졌기 때문이다.

Weighted Round Robin은 Round Robin의 장점인 fairness와 response time 최소화 중 response time 최소화 또한 달성할 수 있다. 3-b)의 표를 확인하면, RR과 WRR의 response time은 각각 0.02s, 0.024s임을 알 수 있다. 의도한 것보다 WRR의 response time이 작게 나오긴 했지만 원래 Weighted Round Robin에서도 비율에 따른 차이가 있을 뿐, response time은 FIFO 등의 다른 스케줄링 정책보다 작게 나올 수 있다. 왜냐하면 우선순위에 따라 프로세스를 실행할 때 time slice를 활용하기 때문이다.

5. 숙제 구현 과정 중 발생한 문제점과 해결방법

숙제 구현 과정에 발생한 문제점은 크게 두 가지였다. 두 문제 모두 해결하는 데 생각보다 오랜 시간이 걸렸다.



```
lala@lala-VirtualBox: ~  
lala@lala-VirtualBox:~$ ls  
examples.desktop 공개 문서 비디오 음악  
shared 다운로드 바탕화면 사진 템플릿  
lala@lala-VirtualBox:~$ sudo mount -t vboxsf Test ~/shared  
[sudo] password for lala:  
mount: wrong fs type, bad option, bad superblock on Test,  
missing codepage or helper program, or other error  
  
In some cases useful info is found in syslog - try  
dmesg | tail or so.  
lala@lala-VirtualBox:~$
```

첫 번째 문제는 위에서 볼 수 있듯이 공유폴더를 만들면서 발생했다. 기존의 virtual box를 사용할 수 없는 상태였기 때문에 다시 다운로드 받아서 했는데, 공유폴더를 생성한 후 mount하는 과정에서 문제가 발생했다. 1차 과제에서는 문제없이 실행되던 부분이었기 때문에 당황했는데, '게스트 확장 CD 이미지 삽입'을 하지 않아서 생긴 문제였다. 이를 실행하자 공유폴더를 mount해서 리눅스 환경과 Windows 환경을 연동하는 폴더를 만들 수 있게 되었다.

```
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched/mysched$ sudo insmod mysched_list.ko
죽었음
lala@lala-VirtualBox: /usr/src/linux-4.15.10/kernel/sched/mysched$ lsmod
Module              Size  Used by
mysched_list        16384  1
```

두 번째 문제는 FIFO를 구현한 후, 모듈에 적재할 때 발생했습니다. 모듈 적재에 해당하는 명령어를 입력하면 위에서 볼 수 있듯이 ‘죽었음’이라는 결과가 나왔습니다. 이를 해결하기 위해 처음에는 c 코드에서만 고치려고 했고, 어디가 잘못되었는지 찾기 어려웠습니다. 그래서 dmesg를 통해 확인해보니 다음과 같은 경고 메시지가 있었습니다.

```
251.899086] INIT_MOD: class = 00000000ed55bf7f
251.899087] INIT_MOD: class = 0000000004eb4ede
251.899088] INIT_MOD: class = 00000000c5ba4ef7
251.899088] INIT_MOD: class = 00000000ef41f31b
251.899094] BUG: unable to handle kernel NULL pointer dereference at 00000000
00000000
251.899179] IP: init_mysched+0x59/0x1000 [mysched_queue]
251.899233] PGD 0 P4D 0
```

즉, init_mysched() 함수에서 class를 할당하는 과정에서 문제가 발생한 것입니다. 이 문제의 원인을 생각해보니 이 함수에는 프로세스를 넣는 queue의 front와 rear를 null로 초기화하는 작업이 있었는데 queue를 전역변수에서 잘못 선언해둔 상태로 진행해서 생긴 문제였습니다. 이 문제 해결 과정을 통해서 이후의 문제가 생겼을 때는 dmesg를 통해 진행 상황을 확인함으로써 쉽게 찾고 해결할 수 있었습니다.