[운영체제 2차 과제]

프로세스 및 리눅스 스케줄링의 이해

이름: 김보민

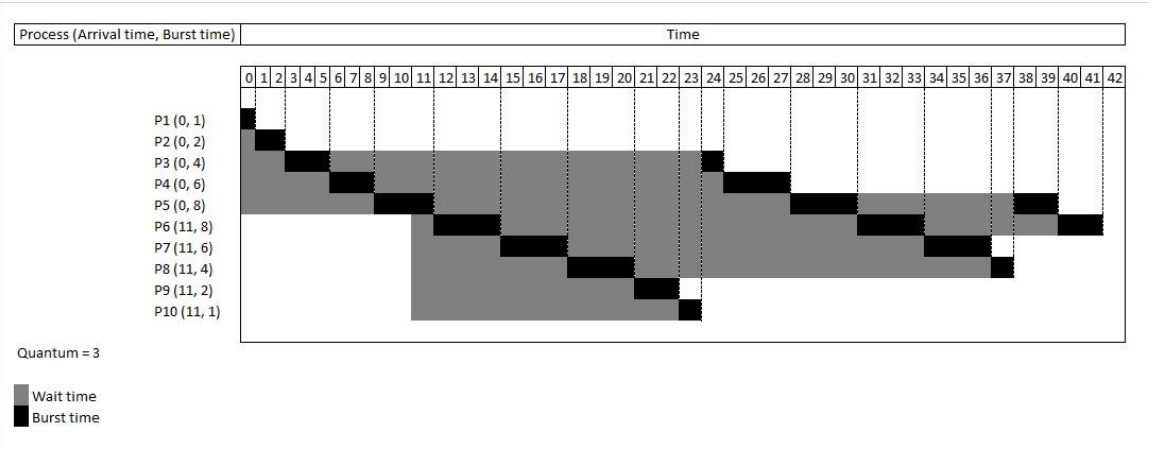
학과: 산업경영공학부

학번: 2021170810

제출일자: 2023년 6월 4일

free day: 3일 사용

0. Round Robin 스케줄링에서 Time Slice의 영향



위 그림에서도 볼 수 있듯이 Round Robin 스케줄링은, 도착한 순서대로 프로세스를 배치하지만 정해진 time quantum (time slice)에 의해 실행을 제한하는 스케줄링 기법이다. 즉 동일한 시간 할당량을 주고 그 시간 안에 완료되지 않은 프로세스는 큐의 맨 뒤로 배치한다. 각 프로세스는 time slice 동안 CPU를 사용할 수 있으며, time slice가 끝나면 다른 프로세스에게 CPU를 양보한다. 이렇게 함으로써 각 프로세스는 동등한 기회를 가지게 되고, 우선순위가 낮은 프로세스도 실행될 수 있다.

Time slice는 프로세스의 성능 자체에도 영향을 미치는데, time slice에 따라 프로세스를 전환할 때마다 context switching이 발생하고, 이로 인해 오버헤드가 발생하기 때문이다. time slice가 짧을수록, 프로세스 전환이 빈번하게 발생하며, 이는 오버헤드를 증가시킨다. Context switching이 생길 때마다 추가적인 시간과 자원을 소모시켜 성능을 저하시킬 수 있다. 반면에 time slice가 길다면, 프로세스가 대기 중인 이벤트나 입출력 작업 등에 대한 응답이 느려질 수 있다. 빠른 응답 시간(response time)을 요구하는 Interactive 프로세스를 만족시키지 못할 수 있는 것이다.

1. 과제 목적

앞서 살펴본 것과 같이, Round Robin 스케줄링은 프로세스를 전환할 때마다 context switching이 발생하고, 이에 따라 오버헤드가 달라진다. 따라서 time slice를 어떻게 설정하는 지에 따라 프로세스의 전체적인 성능에도 영향을 미친다. 본 과제의 목표는 이러한 요인들의 영향을 확인하기 위하여, timeslice를 변화시키며 성능을 확인하는 실험을 설계 및 구현한 뒤 직접 결과를 확인하는 데 있다.

2. 소스코드 및 작업 과정

A. 성능분석을 위한 유저 프로그램(cpu.c) 구현

```c
int calc() {
        int i,j,k;
        int matrixA[ROW][COL];
        int matrixB[ROW][COL];
        int matrixC[ROW][COL];

                for (i = 0; i < ROW; i++) {
                        for (j = 0; j < COL; j++) {
                                for (k = 0; k < COL; k++) {
                                        matrixC[i][j] += matrixA[i][j] * matrixB[i][j];
                                }
                        }
                }
}
```

Calc()는 행렬곱셈을 수행하는 함수로, 과제 튜토리얼에 첨부되어있는 그대로 가져왔다.

```c
int main(int argc, char* argv[]) {
        int i;
        int num_of_processes, time_to_execute;
        pid_t pids[100];

        // input argument error
        if (argc != 3){
                fprintf(stderr, "Usage: %s <num of processes> <time to execute>\n", argv[0]);
                exit(1);
        }

        num_of_processes = atoi(argv[1]);
        time_to_execute = atoi(argv[2]);
```

main() 함수는 프로세스의 개수와 각 프로세스를 수행할 시간 2개의 인자를 받아야한다. main(int argc, char* argv[]) 형태에서 argc는 함수에 전달되는 데이터의 개수, argv[]는 함수에 전달되는 실제적인 데이터이다. 우리는 2개의 인자를 받지만 argc는 자신의 주소정보까지 포함하여 1을 더한 3이 되어야한다. 3이 아닐 시에는, 사용자에게 올바른 입력 형태를 알려주도록 출력하였다. 그리고 프로세스의 개수는 변수 num_of_processes에, 프로세스를 수행할 시간은time_to_execute에 저장했다.

```c
for (i = 0; i < num_of_processes; i++) {
        printf("Creating Process: #%d\n", i);
        pid=i;
        pids[i] = fork(); //creating process
        if (pids[i] < -1) return -1;
        else if (pids[i] == 0) {
                clock_gettime(CLOCK_MONOTONIC, &begin);//시작시각
                pid=i;
```

다음은 num_of_processes 개수만큼의 프로세스를 생성하는 코드이다. 반복문을 통해 i가 0부터 (num_of_processes -1) 까지 증가하게끔 하면서, pid (자신의 프로세스 번호) 또한 i와 같은 값을 가지게끔 설정했다. 부모 프로세스 마다 **fork()**를 통해 프로세스를 생성했으며, 배열 pids[i]에 생성된 자식 프로세스의 ID가 저장되도록 했다. **fork()** 함수는 현재 프로세스를 복제하고, 복제된 프로세스에서는 0을 반환하고, 부모 프로세스에서는 자식 프로세스의 프로세스 ID를 반환한다. pids[i] == -1 이면, fork() 호출에 실패한 경우이다. 새로운 프로세스를 생성할 수 없으므로 종료한다. pids[i] == 0이면 현재 실행 중인 프로세스가 자식 프로세스인 경우로, 자식 프로세스는 이후의 코드 블록을 실행한다.

```c
clock_gettime(CLOCK_MONOTONIC, &begin);//시작시각
pid=i;
timeslice=1;

while (1) {
        calc();
        signal(SIGINT, sighandler_child);
        clock_gettime(CLOCK_MONOTONIC, &end);//종료시각
        count++;

        total_time = ((end.tv_sec - begin.tv_sec) * 1000000000 + (end.tv_nsec - begin.tv_nsec)); //ns
```

Syscall함수 **clock_gettime()**을 이용하여 프로세스가 생성되어 연산을 시작한 시점의 시각을 begin에 저장한다. 이후 while 문에서는 실제로 calc() 함수를 통해 행렬 연산을 수행한 후, end에 종료된 시각도 저장한다. 변수 count는 100ms마다 각 epoch에서 실행한 연산 횟수이며, calc()함수가 끝날때마다, count 값은 1씩 증가시킨다.

total_time은 프로세스 생성부터 행렬연산을 마친 현재 시점까지 걸린 시간으로, 행렬 연산이 끝날때마다 clock_gettime(CLOCK_MONOTONIC)을 통해 저장한 시각과 프로세스 생성 시각의 차로 계산한다. 주의

할 점은 초와 나노초를 구분하여 계산해야하며, 단위를 나노초로 맞추기 위해 초 단위에는 1000000000을 곱하여 총 total_time(ns단위)을 산출한다.

```
if (total_time/100000000 == timeslice){
        printf("PROCESS #%02d count = %04d %dms\n", pid, count, 100);
        timeslice++;
}
```

다음은 100ms마다 각 epoch에서 실행한 연산 횟수와 각 epoch의 실제 실행 시간을 출력하는 코드로, timeslice라는 변수를 도입했다. total_time (ns)를 100000000ns(100ms)으로 나누었을 때 timeslice와 같은지 확인하여 같으면 출력하도록 했다. 여기서 timeslice는 초기값을 1로 가지고, 조건문을 통과할 때마다 1을 증가시켰다. 결과적으로 total_time이 100ms의 배수의 근사값을 가질 때마다 출력이 된다.

```
//총시간이 time_to_execute 초과할 시
if (total_time > (time_to_execute*1000000000LL)){
        printf("Done!! PROCESS #%02d : %06d %lldms\n", pid, count, total_time/1000000
        break;
}
```

행렬연산은 초기에 input으로 받은 프로세스 수행시간만큼 진행해야 했기에, 연산 시간(total_time)이 프로세스를 수행할 시간(time_to_execute)을 넘어갈 경우 최종 결과 메시지를 출력하고 break을 통해 행렬연산 반복문을 빠져나오도록 했다

```
                exit(0);
        }

}

//wait until child ends
for(i = 0; i < num_of_processes; i++) {
        pid_t terminatedChild = waitpid(pids[i], NULL, 0);

 }

return 0;
```

반복문을 빠져나온 자식 프로세스는 exit(0)을 통해 종료했고, 부모 프로세스의 경우 반복문을 통해 모든 자식 프로세스에 대해 waitpid() 함수를 호출하여 각 자식 프로세스의 종료를 기다린다.

***추가과제- signal handler 구현: ctrl+c 입력시 종료**

```
long long total_time;
int pid;
int count;

//extra task
void sighandler_child(int signo) {
        printf("DONE!! Process %02d : %d  %lldms \n", pid, count, total_time/1000000 );
        exit(0);
}
void sighandler_parent(int signo){
    pid_t terminatedChild = waitpid(pid, NULL, 0);
    exit(0);
}
```

sighandler_child() 함수는 자식 프로세스에서 SIGINT(인터럽트) 시그널을 처리하기 위한 핸들러이다. 해당 프로세스가 완료되었음을 출력하고 pid (자신의 프로세스 번호), count (연산횟수), total_time(누적 연산시간)을 출력하도록 했다. sighandler_parent() 함수는 부모 프로세스에서 SIGINT(인터럽트) 시그널을 처리하기 위한 핸들러이다. 자식 프로세스의 종료를 기다린 뒤 부모 프로세스도 종료하도록 구현했다. 이후 main() 함수에서 각각을 signal 함수에 넣어 interrupt를 구현했다.

B. Time Slice에 따른 행렬연산 성능변화 분석

1) 스케줄러 방식 변경 -> Round Robin

이 단계에서는 생성한 프로그램에 대한 CPU 스케줄링 방식을 Round Robin으로 변경한다.

```
struct sched_attr {
   uint32_t size;              /* Size of this structure */
   uint32_t sched_policy;      /* Policy (SCHED_*) */
   uint64_t sched_flags;       /* Flags */
   int32_t  sched_nice;        /* Nice value (SCHED_OTHER,
                                  SCHED_BATCH) */
   uint32_t sched_priority;    /* Static priority (SCHED_FIFO,
                                  SCHED_RR) */
   /* Remaining fields are for SCHED_DEADLINE */
   uint64_t sched_runtime;
   uint64_t sched_deadline;
   uint64_t sched_period;
};

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

```
struct sched_attr attr;
memset(&attr, 0, sizeof(attr));
attr.size = sizeof(struct sched_attr);

attr.sched_priority = 10;       // set process's priority to 10.
attr.sched_policy = SCHED_RR; // set scheduler to RoundRobin.
int result = sched_setattr(getpid(), &attr, 0);
if (result == -1) printf("Error calling sched_setattr.\n");
```

seched_setattr()함수를 만들었고, 이 함수는 SYS_sched_setattr이라는 syscall을 호출하여 프로세스의 스케줄링 정책, 우선순위 등을 변경할 수 있게 하였다. 이 함수가 인자로 받을 구조체 sched_attr도 전에 선언하여 구성요소를 담은 구조체를 설정하였다.

main()함수 내에서 attr.sched_policy = SCHED_RR; 을 이용해 Round-Robin으로 프로세스의 스케줄링 정책을 지정한다. Fork() 함수 이전에 위치시켜 child 프로세스가 모두 RT-RR로 스케줄링 되도록 하였다.

2) CPU 코어 설정

```
root@bomin-VirtualBox:~# cd /sys/fs/cgroup/cpuset
root@bomin-VirtualBox:/sys/fs/cgroup/cpuset# mkdir mycpu
root@bomin-VirtualBox:/sys/fs/cgroup/cpuset# cd mycpu
root@bomin-VirtualBox:/sys/fs/cgroup/cpuset/mycpu# echo 0 > cpuset.cpus
root@bomin-VirtualBox:/sys/fs/cgroup/cpuset/mycpu# echo 0 > cpuset.mems
root@bomin-VirtualBox:/sys/fs/cgroup/cpuset/mycpu# echo $$ > tasks
root@bomin-VirtualBox:/sys/fs/cgroup/cpuset/mycpu# cat tasks
2245
2325
root@bomin-VirtualBox:/sys/fs/cgroup/cpuset/mycpu# cd
root@bomin-VirtualBox:~# ./cpu 2 30
Creating Process: #0
```

성능을 명확하게 관찰하기 위해 core의 개수를 다음과 같이 0번 코어만 하나만을 사용하도록 했다.

이후 실제로 프로세스가 코어 하나만을 사용하는지 확인하기 위해 5개의 프로세스를 5초 동안 실행했고, 아래와 같이 코어를 한 개로 설정하기 전에 비해 설정 후 연산량이 줄어든 것을 확인할 수 있었다.

```
Done!! PROCESS #01 : 000594 5002ms
Done!! PROCESS #03 : 000662 5002ms
PROCESS #00 count = 0609 100ms
Done!! PROCESS #00 : 000609 5003ms
PROCESS #04 count = 0635 100ms
Done!! PROCESS #04 : 000635 5002ms
PROCESS #02 count = 0594 100ms
Done!! PROCESS #02 : 000594 5002ms
```

<설정 전>

```
Done!! PROCESS #02 : 000490 5005ms
PROCESS #00 count = 0588 100ms
Done!! PROCESS #00 : 000588 5009ms
PROCESS #01 count = 0459 100ms
Done!! PROCESS #01 : 000459 5009ms
PROCESS #03 count = 0496 100ms
Done!! PROCESS #03 : 000496 5001ms
PROCESS #04 count = 0546 100ms
Done!! PROCESS #04 : 000546 5002ms
```

<설정 후>

3) Time Slice 변화에 따른 성능 차이 분석

```
bomin@bomin-VirtualBox:~$ ./cpu 1 1
Creating Process: #0
PROCESS #00 count = 0025 100ms
PROCESS #00 count = 0056 100ms
PROCESS #00 count = 0086 100ms
PROCESS #00 count = 0114 100ms
PROCESS #00 count = 0144 100ms
PROCESS #00 count = 0175 100ms
PROCESS #00 count = 0204 100ms
PROCESS #00 count = 0234 100ms
PROCESS #00 count = 0265 100ms
PROCESS #00 count = 0294 100ms
Done!! PROCESS #00 : 000294 1002ms
```

```
bomin@bomin-VirtualBox:~$ ./cpu 3 3
Creating Process: #0
Creating Process: #1
Creating Process: #2
PROCESS #02 count = 0017 100ms
PROCESS #00 count = 0013 100ms
PROCESS #01 count = 0025 100ms
PROCESS #00 count = 0034 100ms
PROCESS #01 count = 0044 100ms
PROCESS #02 count = 0046 100ms
PROCESS #01 count = 0481 100ms
PROCESS #00 count = 0544 100ms
PROCESS #02 count = 0581 100ms
PROCESS #01 count = 0495 100ms
Done!! PROCESS #01 : 000495 3001ms
PROCESS #00 count = 0558 100ms
Done!! PROCESS #00 : 000558 3003ms
PROCESS #02 count = 0600 100ms
Done!! PROCESS #02 : 000600 3003ms
```

```
bomin@bomin-VirtualBox:~$ ./cpu 5 5
Creating Process: #0
Creating Process: #1
Creating Process: #2
Creating Process: #3
Creating Process: #4
PROCESS #01 count = 0009 100ms
PROCESS #04 count = 0536 100ms
PROCESS #03 count = 0485 100ms
PROCESS #02 count = 0490 100ms
Done!! PROCESS #02 : 000490 5005ms
PROCESS #00 count = 0588 100ms
Done!! PROCESS #00 : 000588 5009ms
PROCESS #01 count = 0459 100ms
Done!! PROCESS #01 : 000459 5009ms
PROCESS #03 count = 0496 100ms
Done!! PROCESS #03 : 000496 5001ms
PROCESS #04 count = 0546 100ms
Done!! PROCESS #04 : 000546 5002ms
```

cpu 프로그램을 실행시켰을 때 다음과 같은 결과가 나오며, 실제 실험에서는 2개의 프로세스를 30초 동안 실행했다. Timeslice에 따른 contexts switching 오버헤드 측정이 실험의 목표이기에

```
root@bomin-VirtualBox:~# echo 1 > /proc/sys/kernel/sched_rr_timeslice_ms
```

다음과 같이 proc을 통해 1ms, 10ms, 100ms로 timeslice를 설정하며 프로세스가 모두 종료됐을 때 총 연산횟수를 확인하였다. 이상치 오류를 피하기 위해 각 timeslice마다 5번씩 실험을 진행하였으며, 5개의 결과값 중 중앙값으로 연산값을 지정하여 정리한 결과 다음 figure1, figure2 가 도출되었다.

	1ms	10ms	100ms
RR time-slice	# of calc	# of calc	# of calc
Process #0	3305	3313	3548
Process #1	3377	3488	3489
Total calc.	6682	6801	7037

Figure1. 프로세스 별 Time Slice에 따른 행렬 연산 횟수

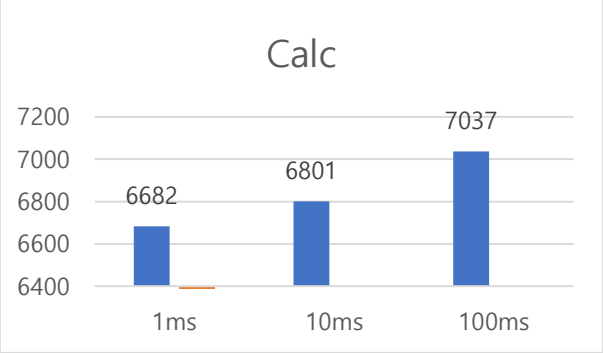


Figure 2. 프로세스 별 Time Slice에 따른 행렬 연산 횟수 그래프

C. CPU Burst Time을 활용한 성능 변화 분석

1) 분석 과정

```
bomin@bomin-VirtualBox:~$ cd /usr/src/linux-4.20.11/kernel/sched
bomin@bomin-VirtualBox:/usr/src/linux-4.20.11/kernel/sched$ sudo gedit stats.h
```

```
static inline void sched_info_depart(struct rq *rq, struct task_struct *t)
{
        unsigned long long delta = rq_clock(rq) - t->sched_info.last_arrival;

        rq_sched_info_depart(rq, delta);

        if (t->state == TASK_RUNNING)
                sched_info_queued(rq, t);

        //#2021170810 kim bomin
        if (t->rt_priority == 10){
                printk("[Pid: %d], CPUburst: %lld, rt_priority:%u\n", t->pid, delta, t->rt_priority);
        }
}
```

다음과 같이 stat.h 파일에 접근하여 코드를 추가하였다. 이후 커널 컴파일을 진행하였고, 재부팅 후 dmesg를 출력하였다.

2) 성능변화 분석 결과

	1ms		10ms		100ms	
RR time-slice	# of calc	Time(s)	# of calc	Time(s)	# of calc	Time(s)
Process #0	3305	8.338143909	3313	14.863347	3548	14.97749029
Process #1	3377	8.270546968	3488	15.138096	3489	15.1297142
Total calc. and Time	**6682**	**16.60869088**	**6801**	**30.00144268**	**7037**	**30.10720449**

RR time-slice	1ms	10ms	100ms
Calculation per second	402.3194874	226.6890986	233.7314314
Baseline = 1ms	100.00%	56.35%	58.10%
Baseline = 10ms	177.48%	100.00%	103.11%

Figure 3. Time Slice에 따른 성능 변화

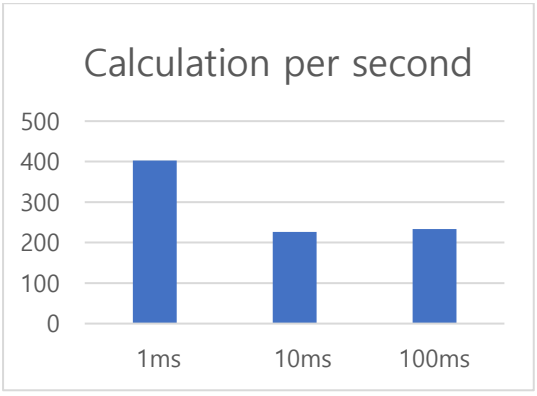
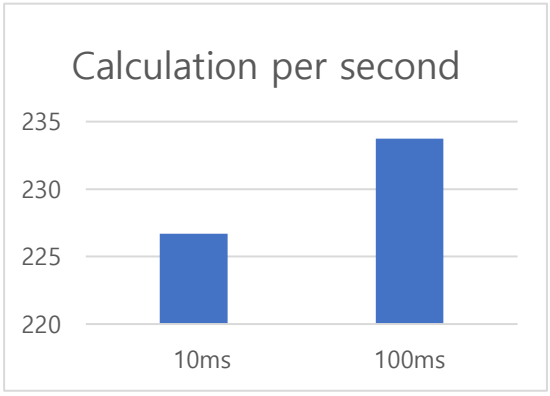


Figure 4. time Slice에 따른 성능 변화 그래프

실험 결과 10ms의 경우, 1ms와 비교했을 때 43.65%의 성능 하락이 발생했고, 100ms 경우 10ms와 비교 시 3.11%의 성능이 향상되었다. 다만 이 실험의 한계점은, time slice가 1ms일 때 cpu burst time의 측정에 오류가 있었다는 점이다. (오류의 원인은 결론 파트에서 분석하였다.) 따라서 1ms일때의 cpu 점유시간이 실제보다 현저히 낮게 측정되었으며, 이는 성능지표인 calculation per second의 값을 현저히 크게 만들었다.



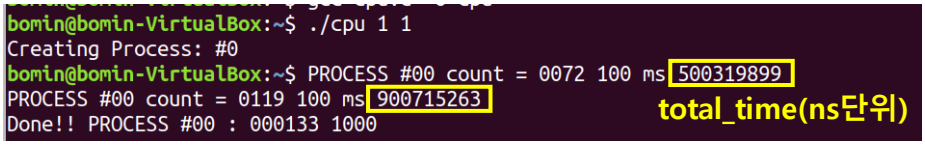
따라서 10ms와 100ms의 관계를 중심으로 분석해보자면, time slice가 100ms일 때 10ms일때보다 성능이 좋은 것을 관찰할 수 있다. time slice에 따라 프로세스를 전환할 때마다 context switching이 발생하고, 이로 인해 오버헤드가 발생하기 때문에, 10ms에서 오버헤드가 더 잦았을 것이다. time slice가 짧을수록, 응답성은 개선되지만 프로세스 전환이 빈번하게 발생하며, 이는 오버헤드를 증가시킴을 확인하였다.

3. 결론

문제점 및 해결과정

cpu.c구현에서의 timeslice 설정의 어려움

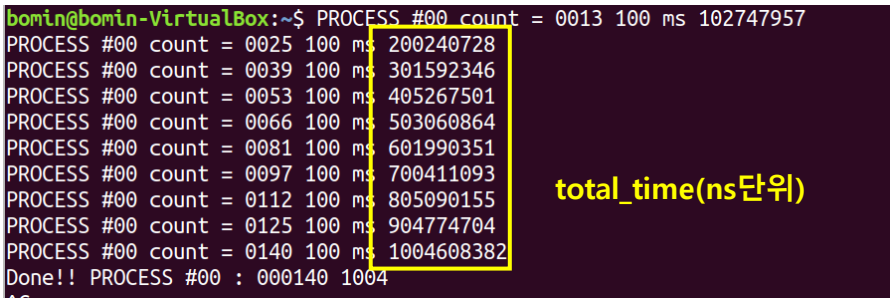
처음에는 100ms마다 각 epoch에서 실행한 연산 횟수를 출력하기 위하여, total_time이 100ms로 나누어 떨어질 때마다 출력하면, 100ms, 200ms, 300ms ... 일때마다 출력이 될 것이라 생각했다. 그래서 if ((total_time/1000000) % 100)==0) 의 조건문을 만족시킬 때마다 출력하게끔 했으나, 다음과 같은 결과가 나왔다.



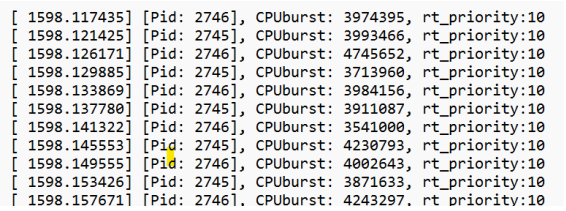
문제는 연산 직후 total time이 100ms로 완전히 나누어 떨어지는 경우가 없을 수도 있다는 점이었다. 따라서 결과값을 보면, total_time을 ms로 환산했을 때 500, 900인 경우만 출력된 것을 볼 수 있는데, 나머지 epoch는 100으로 완전히 나눠떨어지지 않기 때문이다. 따라서, 100의 배수의 근사값을 갖는다면, 출력되도록 다음과 같이 수정하였다.

```
if (total_time/100000000 == timeslice){
        printf("PROCESS #%02d count = %04d %dms\n", pid, count, 100);
        timeslice++;
}
```

Total_time을 100ms로 나눴을 때의 몫이 1일때부터, 증가하여 2, 3, 4 ... 일때 연산횟수를 출력하게끔 하는 코드이다. 이렇게 수정하면 100ms당 하나씩 출력이 되며 꼭 100ms의 배수가 아니더라도 그 근사값일 때 출력이 되는 것을 확인할 수 있다.



실험결과의 한계에 대한 분석



앞서 1ms일때의 cpu burst time의 측정에 오류가 생김을 언급했는데, 이 원인을 분석하자면 프로세스 실행 후 dmesg결과를 보면, cpu burst가 3-4ms대로 지속적으로 출력됨을 발견하였다. 이는 timeslice를 2ms, 3ms로 설정해도 같았으며 4ms까지는 timeslice가 제대로 반영되지 않음을 알게 되었다. 이는 cpu burst time의 합 계산에도 영향을 미쳤고 이러한 이유로 성능 변화 그래프에서 경향성이 모호하게 도출되었다.