운영체제 실습

Assignment #3

Class : A

Professor : 김태석 교수님

Student ID : 2020202090

Name : 최민석

Introduction

Assignment3-1 에서는 시스템 콜을 래핑하여 task_struct 의 정보를 출력하도록 하는 모듈을 구현한다. 이 모듈을 통해 336 번 시스템 콜에서 입력 받은 pid 의 프로세스의 이름, 현재 상태, 그룹 정보, 컨텍스트 스위칭 횟수, fork 호출 횟수, 상위 프로세스 정보, 시빌링 프로세스 정보, 하위 프로세스 정보 등을 표시한다.

Assignment3-2 에서는 temp.txt 에 적힌, 4 의 거듭 제곱 개수의 숫자들을 트리 형태로 더하는 프로그램을 작성한다. numgen.c 에서는 temp.txt 에 한 줄 당 숫자를 순차적으로 작성해 파일을 생성하고 fork.c, thread.c 에서는 각각 하위 프로세스 생성, 스레드 생성을 통해 작업을 분산하여 수행한다.

Assignment3-3 에서는 CPU 스케쥴링 시뮬레이터를 제작해본다. 다양한 스케쥴링 알고리즘을 사용하여 레디 큐에서 작업을 선택해 처리하는 것을 시뮬레이션할 수 있도록 구현한다. FCFS, RR, SJF, SRTF 알고리즘에 대해서 시뮬레이션을 진행한다.

FCFS 는 First Come, First Service 의 약어로, 먼저 들어온 작업 순서대로 처리하는 스케쥴링 알고리즘이다. FIFO 와 유사한 방식으로 동작한다. 대기 시간이 가장 짧지만 실행 시간이 긴 프로세스가 먼저 들어오면 이후 프로세스의 처리 시간이 계속해서 느려지는 문제점이 있다.

SJF 는 Shortest Job First 의 약어로, 가장 빨리 끝나는 작업을 처리하는 스케쥴링 알고리즘이다. 최소한의 컨텍스트 스위칭을 통해 작업량을 줄일 수 있지만, 계속해서 짧은 작업이 들어올 경우 긴 작업은 시간을 할당 받지 못하는 문제점이 있다.

SRTF 는 Shortest Remaining Time First 의 약어로, 작업이 들어오는 순간 남은 프로세스들의 남은 처리 시간을 비교해보고 가장 빨리 끝나는 순서대로 작업을 처리하는 스케쥴링 알고리즘이다. 이 방식도 SJF 와 마찬가지로 특정 프로세스가 시간을 할당 받지 못할 가능성이 있다.

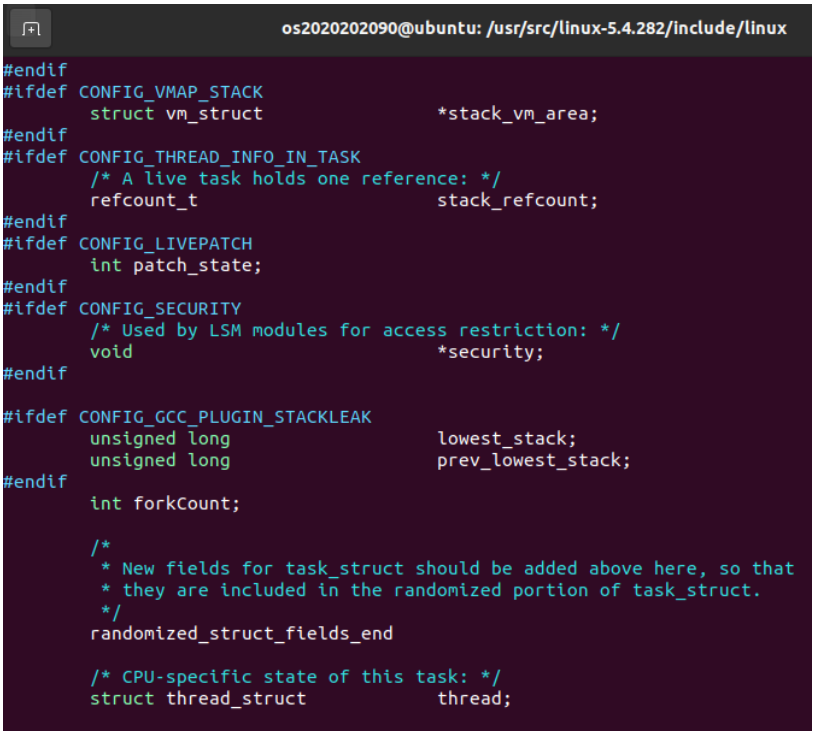
RR 은 라운드-로빈 스케쥴링 알고리즘의 약어로, 우선 순위 큐를 사용하여 우선 순위가 높은 작업을 먼저 큐에서 뽑아 타임 슬라이스만큼 실행하고 우선 순위를 하락시키는 방식의 스케쥴링 알고리즘이다. 모든 프로세스가 공평하게 시간을 할당 받을 수 있다는 장점이 있고, 응답 시간을 개선할 수 있다는 장점이 있다. 하지만 타임 슬라이스가 너무 짧을 경우 컨텍스트 스위칭의 비중이 너무 커지고, 타임 슬라이스가 너무 길면 장점인 응답 시간이 다시 늘어나는 트레이드 오프가 발생한다.

결과화면

1. Assignment3-1

336 번 시스템 콜 os_ftrace 를 후킹하여 입력 받은 pid 에 대한 프로세스 정보를 출력하는 모듈을 구현한다. 입력 받은 pid 의 task_struct 구조체에 있는 프로세스 이름, 프로세스 그룹 리더 정보, 컨텍스트 스위칭 횟수, 상위 프로세스 정보, 하위 프로세스 정보 등을 받아서 출력한다.

과제 구현을 위한 조건 중 fork()한 횟수를 계산하기 위해 커널 내부의 sched.h 에서 task_struct 를 수정해 forkCount 변수를 추가한다.



Int 형 forkCount 변수를 선언한다.

이제 실제 fork()가 수행될 때 forkCount 를 증가시키도록 커널을 수정해야 한다. fork.c 에 선언된 do_fork 함수에 forkCount 를 증가시키는 코드를 추가한다.

os2020202090@ubuntu: /usr/src/linux-5.4.282

```
 * it and waits for it to finish using the VM if required.
 *
 * args->exit_signal is expected to be checked for sanity by the caller.
 */
long _do_fork(struct kernel_clone_args *args)
{
        u64 clone_flags = args->flags;
        struct completion vfork;
        struct pid *pid;
        struct task_struct *p;
        struct task_struct *me = current;
        int trace = 0;
        long nr;

        /*
         * Determine whether and which event to report to ptracer.  When
         * called from kernel_thread or CLONE_UNTRACED is explicitly
         * requested, no event is reported; otherwise, report if the even
         * for the type of forking is enabled.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if (args->exit_signal != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
        }

        p = copy_process(NULL, trace, NUMA_NO_NODE, args);
        add_latent_entropy();

        if (IS_ERR(p))
                return PTR_ERR(p);

        me->forkCount++;
```

이제 fork 를 할 때 현재 task_struct 구조체의 forkCount 가 1 씩 증가할 것이다. 커널을 컴파일한 다음 재부팅한다.

```
os2020202090@ubuntu:~$ cd /usr/src/linux-5.4.282
os2020202090@ubuntu:/usr/src/linux-5.4.282$ sudo make -j12
[sudo] password for os2020202090:
  HOSTCC  arch/x86/tools/relocs_32.o
  HOSTCC  arch/x86/tools/relocs_64.o
  HOSTCC  arch/x86/tools/relocs_common.o
  HOSTCC  scripts/kallsyms
  DESCEND  objtool
  HOSTCC  scripts/genksyms/lex.lex.o
  HOSTCC  scripts/selinux/genheaders/genheaders
  HOSTCC  scripts/selinux/mdp/mdp
  HOSTCC  scripts/sign-file
```

sudo make -j12

sudo make modules_install

sudo make install

sudo reboot

커널을 컴파일 한 후 process_tracer.c 를 컴파일해 커널 오브젝트 파일을 메모리에 적재하고 테스트 코드를 실행해본다.

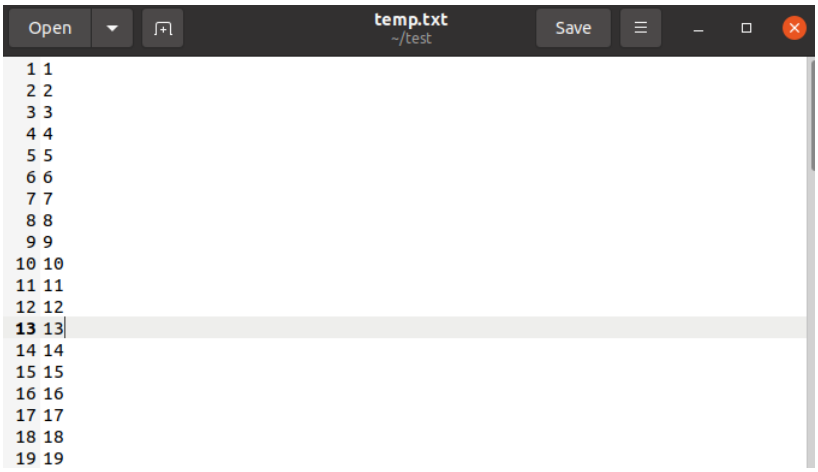
```
os2020202090@ubuntu:~/test$ sudo insmod Process_tracer.ko
[sudo] password for os2020202090:
os2020202090@ubuntu:~/test$ sudo lsmod
Module                  Size  Used by
Process_tracer         16384  0
```

pid=1 에 대한 모듈 실행 결과는 다음과 같다

```
os2020202090@ubuntu:~/test$ dmesg | tail -n 48
[ 3302.705871] ##### TASK INFOMATION of ''[1] systemd'' #####
[ 3302.705875] - task state : Wait
[ 3302.705879] - Process Group Leader : [1] systemd
[ 3302.705882] - Number of context switches : 2406
[ 3302.705886] - Number of calling fork() : 202
[ 3302.705890] - it's parent process : [0] systemd
[ 3302.705893] - it's sibiling process(es) :
[ 3302.705897]   > [2] kthreadd
[ 3302.705900]   > This process has 1 sibling process(es)
[ 3302.705904] - it's child process(es) :
[ 3302.705908]   > [328] systemd-journal
[ 3302.705912]   > [366] systemd-udevd
[ 3302.705916]   > [377] vmware-vmblock-
[ 3302.705919]   > [665] systemd-resolve
[ 3302.705923]   > [666] systemd-timesyn
[ 3302.705927]   > [678] VGAuthService
[ 3302.705931]   > [680] vmtoolsd
[ 3302.705934]   > [758] accounts-daemon
[ 3302.705938]   > [759] acpid
[ 3302.705942]   > [762] avahi-daemon
[ 3302.705945]   > [763] cron
[ 3302.705949]   > [766] dbus-daemon
[ 3302.705953]   > [769] NetworkManager
[ 3302.705957]   > [775] irqbalance
[ 3302.705961]   > [779] networkd-dispat
[ 3302.705964]   > [780] polkitd
[ 3302.705968]   > [784] rsyslogd
[ 3302.705972]   > [796] snapd
[ 3302.705976]   > [797] switcheroo-cont
[ 3302.705980]   > [799] systemd-logind
[ 3302.705984]   > [806] udisksd
[ 3302.705987]   > [807] wpa_supplicant
[ 3302.705992]   > [845] ModemManager
[ 3302.705996]   > [873] unattended-upgr
[ 3302.706000]   > [880] gdm3
[ 3302.706003]   > [930] whoopsie
[ 3302.706007]   > [931] kerneloops
[ 3302.706011]   > [937] kerneloops
[ 3302.706015]   > [978] rtkit-daemon
[ 3302.706019]   > [1086] upowerd
[ 3302.706023]   > [1300] colord
[ 3302.706027]   > [1378] systemd
[ 3302.706031]   > [1415] gnome-keyring-d
[ 3302.706035]   > [1985] fwupd
[ 3302.706039]   > [8143] cupsd
[ 3302.706042]   > [8144] cups-browsed
[ 3302.706046]   > This process has 36 child process(es)
[ 3302.706050] ##### END OF INFORMATION #####
```

2. Assignment3-2

numgen.c 을 통해 temp.txt 에 한 줄에 1 씩 증가하는 정수 값을 입력하여 입력 파일을 준비한다.



(numgen.c 로 생성한 temp.txt)

그리고 나서 이 파일을 입력으로 fork.c, thread.c 를 실행해 트리 형식으로 값들을 더하여 합을 계산하고, 이를 temp.txt 에 기록하는 프로그램을 작성한다. (프로그램 실행 전 버퍼를 비워 정확한 결과를 얻을 수 있도록 한다)

```
370 1552
371 2576
372 3600
373 4624
374 5648
375 6672
376 7696
377 2080
378 6176
379 10272
380 14368
381 8256
382 24640
383 32896
```

MAX_PROCESS=128 을 실행한 결과이다. 각 중간 값과 전체 합이 기록된 것을 관찰할 수 있다.

- fork.c 실행 결과 (MAX_PROCESSES=4)

```
os2020202090@ubuntu:~/test$ rm -rf tmp*
os2020202090@ubuntu:~/test$ sync
os2020202090@ubuntu:~/test$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2020202090:
3
os2020202090@ubuntu:~/test$ ./fork.o
Final Result: 36
Execution Time: 0.0005 seconds
```

실행 시간: 0.0005 초

- thread.c 실행 결과 (MAX_PROCESSES=4)

```
os2020202090@ubuntu:~/test$ rm -rf tmp*
os2020202090@ubuntu:~/test$ sync
os2020202090@ubuntu:~/test$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2020202090:
3
os2020202090@ubuntu:~/test$ ./thread.o
Final Result: 36
Execution Time: 0.0018 seconds
```

실행 시간: 0.0018 초

fork.o 에서 사용하는 exit 함수는 상태 값을 최대 8 비트만 반환할 수 있기 때문에 255 가 넘는 값을 반환하면 오버플로우가 발생한다.

```
num1: 117, num2:118, sum: 235
num1: 119, num2:120, sum: 239
num1: 121, num2:122, sum: 243
num1: 123, num2:124, sum: 247
num1: 125, num2:126, sum: 251
num1: 127, num2:128, sum: 255
num1: 129, num2:130, sum: 259
```

64 번째에서 129+130=259 가 출력되어 값이 255 를 넘기 때문에 exit(sum)을 통해 실제로는 4 가 반환된다. 입력은 등차가 1 인 등차 수열의 형태로 들어오기 때문에, 64 번째마다 값을 보정해줘서 해결할 수 있다.

- 보정을 하지 않았을 경우 (MAX_PROCESSES=128)

```
os2020202090@ubuntu:~/test$ rm -rf tmp*
os2020202090@ubuntu:~/test$ sync
os2020202090@ubuntu:~/test$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2020202090@ubuntu:~/test$ ./fork.o
Final Result: 16512
Execution Time: 0.0145 seconds
```

프로세스 개수가 일정 이상으로 늘어나면 오버플로우로 인해 잘못된 값을 출력하고, 전체 연산 결과가 올바르지 않게 된다.

- 보정을 수행한 경우 (MAX_PROCESSES=128)

```
os2020202090@ubuntu:~/test$ rm -rf tmp*
os2020202090@ubuntu:~/test$ sync
os2020202090@ubuntu:~/test$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2020202090:
3
os2020202090@ubuntu:~/test$ ./fork.o
Final Result: 32896
Execution Time: 0.0149 seconds
```

정확한 연산 결과가 출력된다.

Thread 와 fork 를 사용한 방식을 각각 비교하면, MAX_PROCESSES 가 많아질수록 병렬 프로세스가 더욱 많아지는 fork.c 의 실행 속도가 더 빠른 것을 관찰할 수 있다.

- fork.c (MAX_PROCESSES=128)

```
os2020202090@ubuntu:~/test$ rm -rf tmp*
os2020202090@ubuntu:~/test$ sync
os2020202090@ubuntu:~/test$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2020202090:
3
os2020202090@ubuntu:~/test$ ./fork.o
Final Result: 32896
Execution Time: 0.0149 seconds
```

실행 시간: 0.0149 초

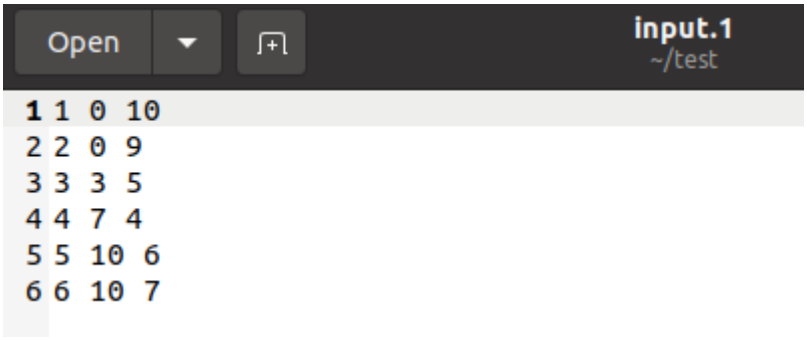
- thread.c (MAX_PROCESSES=128)

```
os2020202090@ubuntu:~/test$ rm -rf tmp*
os2020202090@ubuntu:~/test$ sync
os2020202090@ubuntu:~/test$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2020202090:
3
os2020202090@ubuntu:~/test$ ./thread.o
Final Result: 32896
Execution Time: 0.0183 seconds
```

실행 시간: 0.0183 초

3. Assignment3-3

cpu_scheduler.c 에서는 input file 을 읽어와서 task 를 배열로 저장한다. 입력 파일에는 pid, arrival time, burst time 이 빈 칸으로 구분되어 기록되어 있는데, 이것을 모두 읽어서 프로세스 구조체에 저장한다.



프로그램을 실행할 때 인수로 입력 데이터가 든 파일의 이름과 스케쥴링 알고리즘을 입력한다. 라운드 로빈 알고리즘으로 실행할 경우 타임 슬라이스 값을 추가로 입력해줘야 한다. 만약 RR 알고리즘인데 타임 슬라이스 값이 입력되지 않을 경우 에러 메시지를 출력한다.

알고리즘 실행 결과로 간트 차트로 프로세스 실행 시퀀스를 표시하고, 평균 대기 시간, 평균 턴어라운드 시간, 평균 응답 시간, CPU 점유율을 계산하여 출력한다.

- FCFS 알고리즘 실행 결과

```
os2020202090@ubuntu:~/test$ ./cpu_scheduler.o input.1 FCFS
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 | P2 | P2 | P2
| P2 | P2 | P2 | P3 | P3 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5
| P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting Time = 14.17
Average Turnaround Time = 21.00
Average Response Time = 14.17
CPU Utilization = 98.56%
```

FCFS 는 컨텍스트 스위칭 횟수가 가장 적기 때문에 전체 실행 시간은 가장 적게 나온다. 하지만 컨베이 효과 때문에 앞의 작업 실행 시간만큼 뒤의 작업들의 대기 시간이 증가하게 되어 평균 응답 시간은 가장 길다.

- RR 알고리즘 실행 결과 (timeQuantum=2)

```
os2020202090@ubuntu:~/test$ ./cpu_scheduler.o input.1 RR 2
Gantt Chart:
| P1 | P1 | P2 | P2 | P3 | P3 | P1 | P1 | P2 | P2 | P3 | P3 | P4 | P4 | P5 | P5
| P6 | P6 | P1 | P1 | P2 | P2 | P3 | P4 | P4 | P5 | P5 | P6 | P6 | P1 | P1 | P2
| P2 | P5 | P5 | P6 | P6 | P1 | P1 | P2 | P6 |
Average Waiting Time = 23.00
Average Turnaround Time = 29.83
Average Response Time = 3.00
CPU Utilization = 96.16%
```

RR 은 우선 순위 큐에서 고른 작업을 타임 슬라이스만큼 돌아가면서 실행한다. 응답 시간은 크게 개선되지만 평균 대기 시간과 평균 턴어라운드 시간이 높게 측정된다.

- SJF 알고리즘 실행 결과

```
os2020202090@ubuntu:~/test$ ./cpu_scheduler.o input.1 SJF
Gantt Chart:
| P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P4 | P4 | P4 | P4 | P3 | P3 | P3
| P3 | P3 | P5 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P1
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 |
Average Waiting Time = 10.83
Average Turnaround Time = 17.67
Average Response Time = 10.83
CPU Utilization = 98.57%
```

SJF 는 작업 스위칭 간에 리스트에서 가장 버스트 시간이 짧은 작업을 선택해 끝까지 실행한다. 응답 시간은 개선되지만 첫번째에 들어온 P1 프로세스의 버스트 시간이 길어 거의 마지막에 와서야 시간을 할당을 받게 되었다. (starvation 발생)

- SRTF 알고리즘 실행 결과

```
os2020202090@ubuntu:~/test$ ./cpu_scheduler.o input.1 SRTF
Gantt Chart:
| P2 | P2 | P2 | P3 | P3 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P2 | P2 | P2 | P2
| P2 | P2 | P5 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P1
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 |
Average Waiting Time = 10.50
Average Turnaround Time = 17.33
Average Response Time = 9.00
CPU Utilization = 98.35%
```

SRTF 는 작업이 들어올 때 큐에서 가장 버스트 시간이 짧은 작업을 선택해 끝까지 실행한다. SJF 에 비해 응답 시간이 개선되었지만 P1 에 starvation 이 발생하는 문제는 동일하게 관찰된다.

고찰

- Assignment3-1

  처음 구현했을 때 하위 프로세스를 표시하기 위해 list_for_each 의 내부에서 리스트 엔트리의 하위 프로세스를 받아와서 표시를 해보니 pid 의 값이 알 수 없는 값으로 출력됐었다.

  하위 프로세스의 pid 가 일관적으로 876773376 으로 출력되어 처음엔 변수에 문제가 있는 줄 알았지만 나중에 원인을 알 수 있었다.

  list_for_each 에서 이미 pid task 의 children 리스트에 진입했었기 때문에 이 내부에서 다시 children 을 호출하면 하위 프로세스의 하위 프로세스에 접근하는 것이기 때문에 에러가 발생했었다. 하위 프로세스를 선택한 뒤 그 프로세스의 시빌링에 대해 접근하는 것을 통해서 해결하였다.

- Assignment3-2

  fork.c 에서 하위 프로세스가 exit 로 반환하는 값은 8 비트라서 255 이상의 값을 전달할 수 없으니 시프트를 통해 원본 값을 얻어야 한다고 써 있었는데, 256 이상의 값은 9 비트로 10000000x 처럼 MSB 와 LSB 간 길이가 8 비트이기 때문에 시프트를 해줘서 정확한 값을 복원하기는 어려워 보였었다. 나는 시프트를 통해 해결할 수 없다고 판단하고, 내부 반복 루프의 인덱스를 통해 값이 64 를 넘어가는 순간부터 오버플로우로 초기화되는 값인 256 을 곱해서 넣어 줌으로써 값을 보정해주었다.

- Assignment3-3

  처음엔 간트 차트를 밀리 세컨드 단위로 출력하는 것을 모르고 FCFS 를 구현했다가. 이후에 이 사실을 알게 되어 수정했다.

  FCFS, SJF, SRTF 는 작업 수행 중 컨텍스트 스위칭이 없기 때문에 간단하게 해당 프로세스의 버스트 타임만큼 반복하여 출력하면 됐다.

  RR 에선 타임 슬라이스만큼 실행할 때, 타임 슬라이스 이후 버스트 시간이 남은 경우, 버스트 시간이 끝난 경우로 나누었고, 실제 실행되는 만큼 간트 차트를 출력할 수 있도록 했다.

Reference

- 24-2_OSLab_Assignment-3_v5.pdf
- 24-2_OSLab_06_Task_Management.pdf
- 24-2_OSLab_07_Thread.pdf
- 24-2_OSLab_09_CPU_Scheduling.pdf