

운영체제 실습

[Assignment #3]

Class : 목 3
Professor : 최상호
Student ID : 2020202031
Name : 김재현

Introduction

Fork 와 thread 를 통해 다중 프로세스/쓰레드로 수행하는 프로그램을 작성해봄으로써 다중 프로세스/쓰레드의 작동방식을 알아보고, 두 작동의 결과, 수행시간 등을 비교해봅니다.

FCFS, RR, SJF, SRTF 등의 알고리즘을 사용하여 CPU scheduling simulator 를 구현해봅니다.

결과화면

3-1

```
// finding target
do
{
    if(target_task->pid == pid)
        break;
    target_task = next_task(target_task);
}
while(target_task->pid != init_task.pid);
```

init_task 부터 모든 프로세스를 순환하며 입력받은 pid 와 같은 task_struct 를 찾습니다.

process name

```
printk(KERN_INFO "##### TASK INFORMATION of '['%d] %s' #####\n", target_task->pid, target_task->comm);
```

찾은 target_task 의 pid 와 comm(process name)을 출력합니다.

process status

```
static const char* state_to_str(const struct task_struct* target)
{
    long state = target->state;
    long exit_state = target->exit_state;

    switch(state) {
    case TASK_RUNNING:
        return "Running or ready";
    case TASK_INTERRUPTIBLE:
        return "Wait";
    case TASK_UNINTERRUPTIBLE:
        return "Wait with ignoring all signals";
    case TASK_STOPPED:
        return "Stopped";
    }

    switch(exit_state) {
    case EXIT_ZOMBIE:
        return "Zombie process";
    case EXIT_DEAD:
        return "Dead";
    }

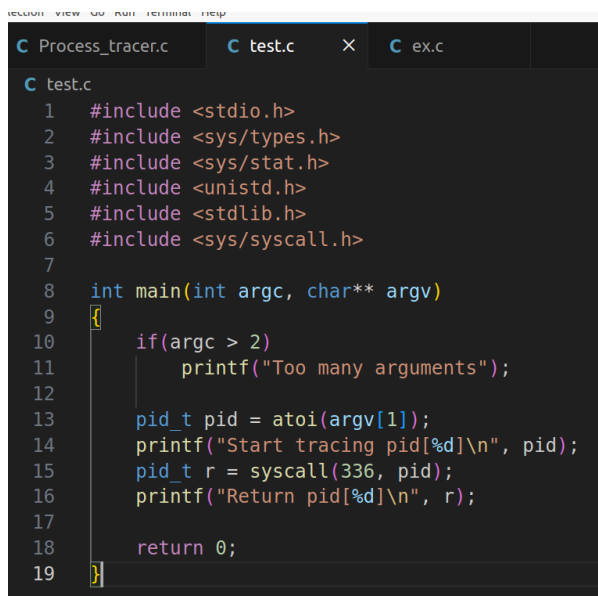
    return "etc.";
}
```

각 task_struct 에는 state 와 exit_state 가 존재합니다. status 와 exit_status 의 값에 따라 현재 프로세스의 상태가 결정됩니다. switch 문을 통해 각 상태에 따라 특정 문자열을 반환하도록 했습니다.

```
printk(KERN_INFO "- task state : %s\n", state_to_str(target_task));
```

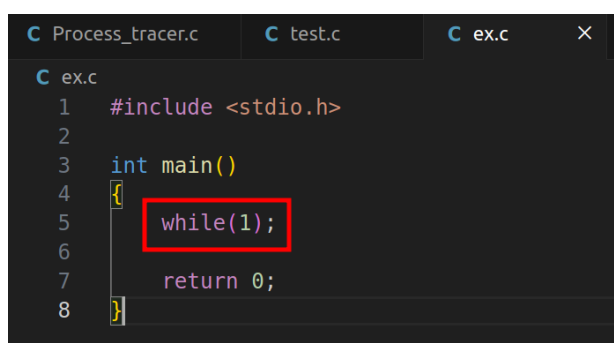
target 의 state 를 출력합니다.

- 다음과 같이 pid 를 입력 받고 해당 pid 의 정보를 출력하는 파일을 만듭니다.



```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <sys/syscall.h>
7
8 int main(int argc, char** argv)
9 {
10     if(argc > 2)
11         printf("Too many arguments");
12
13     pid_t pid = atoi(argv[1]);
14     printf("Start tracing pid[%d]\n", pid);
15     pid_t r = syscall(336, pid);
16     printf("Return pid[%d]\n", r);
17
18     return 0;
19 }
```

- Running or ready



```
1 #include <stdio.h>
2
3 int main()
4 {
5     while(1);
6
7     return 0;
8 }
```

다음과 같이 무한반복문을 도는 파일을 실행시킵니다.

```
os2020202031@ubuntu:~/3-1$ ./ex &
[1] 4024
os2020202031@ubuntu:~/3-1$ ./test 4024
Start tracing pid[4024]
Return pid[4024]
os2020202031@ubuntu:~/3-1$ dmesg | tail -n 20
[ 1054.194654] - its sibling process(es)
[ 1054.194655] > [4018] test
[ 1054.194656] > [0] \xc0U\xb2\x80\x88\xff\xff
[ 1054.194657] > This process has 2 sibling process(es)
[ 1054.194657] - its child process(es)
[ 1054.194658] It has no child.
[ 1054.194659] ##### END OF INFORMATION #####
[ 1089.708863] ##### TASK INFORMATION of '[4024] ex' #####
[ 1089.708865] - task state : Running or ready
[ 1089.708866] - Process Group Leader : [4024] ex
[ 1089.708867] - Number of context switches : 54
[ 1089.708868] - Number of calling forks : 0
[ 1089.708869] - its parent process : [2698] bash
[ 1089.708870] - its sibling process(es)
[ 1089.708871] > [4025] test
[ 1089.708872] > [0] \xc0U\xb2\x80\x88\xff\xff
[ 1089.708872] > This process has 2 sibling process(es)
[ 1089.708873] - its child process(es)
[ 1089.708874] It has no child.
[ 1089.708874] ##### END OF INFORMATION #####
os2020202031@ubuntu:~/3-1$
```

다음과 같이 task state 가 Running or ready 인 것을 알 수 있습니다.

- Wait

```
C Process_tracer.c  C test.c  C ex.c  X
C ex.c
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5
6  int main()
7  {
8      // parent
9      if(fork() != 0)
10         wait(NULL);
11     // child
12     else
13         while(1);
14
15     return 0;
16 }
```

다음과 같이 무한루프를 도는 자식 프로세스를 wait 하는 파일을 만듭니다.

```
os2020202031@ubuntu: ~/3-1
os2020202031@ubuntu:~/3-1$ ./ex &
[1] 4209
os2020202031@ubuntu:~/3-1$ ./test 4209
Start tracing pid[4209]
Return pid[4209]
os2020202031@ubuntu:~/3-1$ dmesg | tail -n 20
[ 1089.708872] > [0] \xc0U|\xb2\x80\x88\xff\xff
[ 1089.708872] > This process has 2 sibling process(es)
[ 1089.708873] - its child process(es)
[ 1089.708874] It has no child.
[ 1089.708874] ##### END OF INFORMATION #####
[ 1712.151765] audit: type=1400 audit(1730358026.033:50): apparmor="DENIED" operation="capable" profile="/usr/sbin/cups-browsed" pid=4095 comm="cups-browsed" capability=23 capname="sys_nice"
[ 2149.136002] ##### TASK INFORMATION of '[4209] ex' #####
[ 2149.136005] - task state : Wait
[ 2149.136006] - Process Group Leader : [4209] ex
[ 2149.136007] - Number of context switches : 1
[ 2149.136008] - Number of calling forks : 1
[ 2149.136010] - its parent process : [2698] bash
[ 2149.136010] - its sibling process(es)
[ 2149.136011] > [4211] test
[ 2149.136012] > [0] \xc0U|\xb2\x80\x88\xff\xff
[ 2149.136013] > This process has 2 sibling process(es)
[ 2149.136014] - its child process(es)
[ 2149.136015] > [4210] ex
[ 2149.136016] > This process has 1 child process(es)
[ 2149.136017] ##### END OF INFORMATION #####
```

다음과 같이 task state 가 Wait 인 것을 알 수 있습니다.

- Stopped

```
Process_tracer.c  test.c  ex.c  X
C ex.c
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5
6  int main()
7  {
8      while(1);
9
10     return 0;
11 }
```

다음과 같이 무한루프를 도는 파일을 만듭니다.

```
os2020202031@ubuntu: ~/3-1
os2020202031@ubuntu:~/3-1$ ./ex
^Z
[1]+  Stopped                  ./ex
os2020202031@ubuntu:~/3-1$ jobs -l
[1]+  4868 Stopped              ./ex
os2020202031@ubuntu:~/3-1$ ./test 4868
Start tracing pid[4868]
Return pid[4868]
os2020202031@ubuntu:~/3-1$ dmesg | tail -n 15
[ 2489.077641] It has no child.
[ 2489.077641] ##### END OF INFORMATION #####
[ 3062.724951] ##### TASK INFORMATION of '[4868] ex' #####
[ 3062.724954] - task state : Stopped
[ 3062.724955] - Process Group Leader : [4868] ex
[ 3062.724956] - Number of context switches : 21
[ 3062.724957] - Number of calling forks : 0
[ 3062.724958] - its parent process : [2698] bash
[ 3062.724959] - its sibling process(es)
[ 3062.724960]   > [4869] test
[ 3062.724961]   > [0] \xc0U\|xb2\x80\x88\xff\xff
[ 3062.724962]   > This process has 2 sibling process(es)
[ 3062.724962] - its child process(es)
[ 3062.724963] It has no child.
[ 3062.724963] ##### END OF INFORMATION #####
os2020202031@ubuntu:~/3-1$
```

ctrl + z 를 통해 ex 프로세스에 SIGSTP 신호를 보내니 task state 가 Stopped 가 되는 것을 확인할 수 있습니다.

- Zombie process

```
C Process_tracer.c  C test.c  C ex.c  X
C ex.c
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main()
8  {
9      if(fork() == 0)
10         exit(1);
11     else
12         while(1);
13
14     return 0;
15 }
```

다음은 자식 프로세스를 종료시킨 후 부모 프로세스가 wait 없이 무한루프를 도는 파일입니다. 자식 프로세스는 Zombie process 가 될 것입니다.

```

os2020202031@ubuntu:~/3-1$ ./ex &
[1] 7913
os2020202031@ubuntu:~/3-1$ ps u
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
os20202+      1473  0.0  0.1 164020  6596 tty2    Ssl+  Oct30   0:00 /usr/lib/gdm3/gdm-x-session --run-sc
os20202+      1479  2.5  3.1 404408 127248 tty2    Sl+   Oct30   2:25 /usr/lib/xorg/Xorg vt2 -displayfd 3
os20202+      1523  0.0  0.3 188220 13708 tty2    Sl+   Oct30   0:00 /usr/libexec/gnome-session-binary --
os20202+      2698  0.0  0.1 11232  5132 pts/0    Ss   Oct30   0:00 bash
os20202+      7913 94.0  0.0   2356   576 pts/0    R    01:07   0:02 ./ex
os20202+      7914  0.0  0.0     0     0 pts/0    Z    01:07   0:00 [ex] <defunct>
os20202+      7915  0.0  0.0  11488  3236 pts/0    R+   01:07   0:00 ps u
os2020202031@ubuntu:~/3-1$ ./test 7914
Start tracing pid[7914]
Return pid[7914]
os2020202031@ubuntu:~/3-1$ dmesg | tail -n 15
[ 5284.078414] - its child process(es)
[ 5284.078415] It has no child.
[ 5284.078416] ##### END OF INFORMATION #####
[ 5747.374403] ##### TASK INFORMATION of '[7914] ex' #####
[ 5747.374406] - task state : Zombie process
[ 5747.374407] - Process Group Leader : [7914] ex
[ 5747.374408] - Number of context switches : 1
[ 5747.374409] - Number of calling forks : 0
[ 5747.374410] - its parent process : [7913] ex
[ 5747.374411] - its sibling process(es)
[ 5747.374412] > [0] @\xb7/\x80\x88\xff\xff
[ 5747.374413] > This process has 1 sibling process(es)
[ 5747.374414] - its child process(es)
[ 5747.374414] It has no child.
[ 5747.374415] ##### END OF INFORMATION #####
os2020202031@ubuntu:~/3-1$

```

예상과 동일하게 task state 가 Zombie process 로 표시되는 것을 알 수 있습니다.

process group leader

```
group_leader_task = target_task->group_leader;
```

task_struct 멤버변수 group_leader 를 통해 process group leader 의 task_struct 를 얻었습니다.

context switch 횟수

task_struct 의 nvcsw 멤버변수에는 자발적인 context switch 횟수가 저장되고, nivcsw 멤버변수에는 비자발적인 context switch 횟수가 저장되므로, 이 두 변수의 합이 곧 context switch 횟수입니다.

```
printk(KERN_INFO "- Number of context switches : %ld\n", target_task->nvcsw + target_task->nivcsw);
```

target 의 context switch 횟수를 출력합니다.

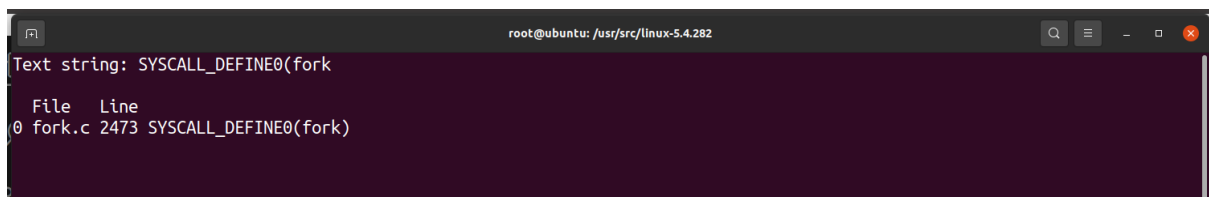
fork

```
int fork_count;
/*
 * New fields for task_struct should be added above here, so that
 * they are included in the randomized portion of task_struct.
 */
randomized_struct_fields_end

/* CPU-specific state of this task: */
struct thread_struct thread;

/*
 * WARNING: on x86, 'thread_struct' contains a variable-sized
 * structure. It *MUST* be at the end of 'task_struct'.
 *
 * Do not put anything below here!
 */
```

struct task_struct 구조체에 fork 횟수를 카운트하는 fork_count 라는 int 형 변수를 만들었습니다.



The screenshot shows a debugger window with the title bar "root@ubuntu: /usr/src/linux-5.4.282". The search bar contains the text "SYSCALL_DEFINE0(fork)". Below the search bar, a table lists the search results:

File	Line
0 fork.c	2473 SYSCALL_DEFINE0(fork)

cscope 를 통해 fork syscall 의 원형을 찾습니다.

```

#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
    struct kernel_clone_args args = {
        .exit_signal = SIGCHLD,
    };

    return _do_fork(&args);
#else
    /* can not support in nommu mode */
    return -EINVAL;
#endif
}
#endif

#ifdef __ARCH_WANT_SYS_VFORK
SYSCALL_DEFINE0(vfork)
{
    struct kernel_clone_args args = {
        .flags          = CLONE_VFORK | CLONE_VM,
        .exit_signal    = SIGCHLD,
    };

    return _do_fork(&args);
}
#endif

```

fork 와 vfork 모두 _do_fork 를 호출하는 것을 확인할 수 있습니다.

```
root@ubuntu: /usr/src/linux-5.4.282

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

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

/*
 * Do this prior waking up the new thread - the thread pointer
 * might get invalid after that point, if the thread exits quickly.
 */
trace_sched_process_fork(current, p);

pid = get_task_pid(p, PIDTYPE_PID);
nr = pid_vnr(pid);

// make fork_count of child process 0
p->fork_count = 0;

// increase fork_count of current process
current->fork_count++;

if (clone_flags & CLONE_PARENT_SETTID)
```

2393,36-43 79%

다음은 _do_fork 함수 정의의 일부분입니다.

copy_process 는 부모 프로세스의 PCB 를 복사하여 새로운 PCB 를 생성하는 함수로, p 는 current 를 복사한 새로운 task_struct 즉, 자식 프로세스의 task_struct 입니다.

copy_process 전후로 다음과 같이 현재 프로세스의 fork_count 를 증가시키는 코드와 자식 프로세스의 fork_count 를 초기화시키는 코드를 작성합니다.

```
printk(KERN_INFO "- Number of calling forks : %d\n", target_task->fork_count);
```

fork_count 를 출력합니다.

parent process

```
real_parent_task = target_task->parent;
```

```
printk(KERN_INFO "- its parent process : [%d] %s\n", real_parent_task->pid, real_parent_task->comm);
```

부모 프로세스의 pid 와 process name 출력하게 했습니다.

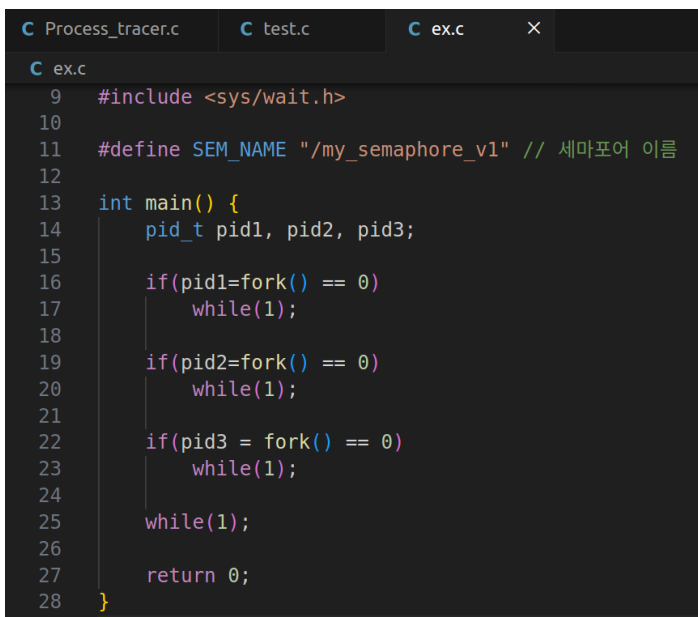
sibling, children process 개수 확인

```
printk(KERN_INFO "- its sibling process(es)\n");
// finding all siblings of target
list_for_each_entry(sibling_task, &target_task->sibling, sibling)
{
    if(sibling_task->pid == 0)
        continue;

    /* task now points to one of current's siblings */
    printk(KERN_INFO "\t> [%d] %s\n", sibling_task->pid, sibling_task->comm);
    sibling_cnt++;
}
if(sibling_cnt > 0)
    printk(KERN_INFO "\t> This process has %d sibling process(es)\n", sibling_cnt);
else
    printk(KERN_INFO "It has no sibling.\n", sibling_cnt);

printk(KERN_INFO "- its child process(es)\n");
// finding all children of target
list_for_each_entry(children_task, &target_task->children, sibling)
{
    /* task now points to one of current's children */
    printk(KERN_INFO "\t> [%d] %s\n", children_task->pid, children_task->comm);
    child_cnt++;
}
if(child_cnt > 0)
    printk(KERN_INFO "\t> This process has %d child process(es)\n", child_cnt);
else
    printk(KERN_INFO "It has no child.\n", child_cnt);
```

target_task 의 sibling, children 을 통해 형제자매, 자식 프로세스들을 순환하며 모든 프로세스를 출력하고 개수를 카운트합니다.



```
1  #include <sys/wait.h>
2
3  #define SEM_NAME "/my_semaphore_v1" // 세마포어 이름
4
5  int main() {
6      pid_t pid1, pid2, pid3;
7
8      if(pid1=fork() == 0)
9          while(1);
10
11      if(pid2=fork() == 0)
12          while(1);
13
14      if(pid3 = fork() == 0)
15          while(1);
16
17      while(1);
18
19      return 0;
20 }
```

다음은 세 개의 자식프로세스가 무한루프를 돌도록 하는 코드입니다.

```
os2020202031@ubuntu:~/3-1$ ./ex &
[1] 10214
os2020202031@ubuntu:~/3-1$ ps u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
os20202+  1473  0.0  0.1 164020 6028 tty2    Ssl+  Oct30   0:00 /usr/lib/gdm3/gdm-x-session --run-sc
os20202+  1479  2.1  2.5 408964 99908 tty2    Sl+   Oct30   3:55 /usr/lib/xorg/Xorg vt2 -displayfd 3
os20202+  1523  0.0  0.3 188220 12636 tty2    Sl+   Oct30   0:00 /usr/libexec/gnome-session-binary --
os20202+  2698  0.0  0.1 11232 5152 pts/0    Ss   Oct30   0:01 bash
os20202+  7980  0.0  0.1 10740 5004 pts/1    Ss   01:14   0:00 bash
os20202+  8054  1.2  0.2 12944 11296 pts/1    S+   01:17   0:58 cscope -R
os20202+  8073  0.0  0.2 24672 10060 pts/1    S+   01:18   0:00 vi +96 include/linux/sched.h
os20202+  10214  0.0  0.0 2356 576 pts/0    R    02:34   0:06 ./ex
os20202+  10215  0.0  0.0 2356 72 pts/0    R    02:34   0:00 ./ex
os20202+  10216  0.0  0.0 2356 72 pts/0    R    02:34   0:06 ./ex
os20202+  10217  0.0  0.0 2356 72 pts/0    R    02:34   0:00 ./ex
os20202+  10219  0.0  0.0 11488 3248 pts/0    R+   02:34   0:00 ps u
os2020202031@ubuntu:~/3-1$ ./test 10214
Start tracing pid[10214]
Return pid[10214]
os2020202031@ubuntu:~/3-1$ ./test 10216
Start tracing pid[10216]
Return pid[10216]
```

부모프로세스의 pid 는 10214

자식프로세스의 pid 는 10215~10217 입니다.

```
[10954.842072] ##### TASK INFORMATION of '[10214] ex' #####
[10954.842074] - task state : Running or ready
[10954.842076] - Process Group Leader : [10214] ex
[10954.842077] - Number of context switches : 761
[10954.842078] - Number of calling forks : 3
[10954.842080] - its parent process : [2698] bash
[10954.842080] - its sibling process(es)
[10954.842081] > [10220] test
[10954.842082] > This process has 1 sibling process(es)
[10954.842083] - its child process(es)
[10954.842084] > [10215] ex
[10954.842085] > [10216] ex
[10954.842087] > [10217] ex
[10954.842088] > This process has 3 child process(es)
[10954.842088] ##### END OF INFORMATION #####
[10960.424337] ##### TASK INFORMATION of '[10216] ex' #####
[10960.424341] - task state : Running or ready
[10960.424342] - Process Group Leader : [10216] ex
[10960.424343] - Number of context switches : 892
[10960.424344] - Number of calling forks : 0
[10960.424345] - its parent process : [10214] ex
[10960.424346] - its sibling process(es)
[10960.424347] > [10217] ex
[10960.424348] > [10215] ex
[10960.424349] > This process has 2 sibling process(es)
[10960.424350] - its child process(es)
[10960.424351] It has no child.
[10960.424352] ##### END OF INFORMATION #####
os2020202031@ubuntu:~/3-1$
```

부모프로세스는 test 가 sibling 이고, 자식은 3 개입니다.

자식프로세스는 ex 가 sibling 이고, 자식은 없습니다.

3-2

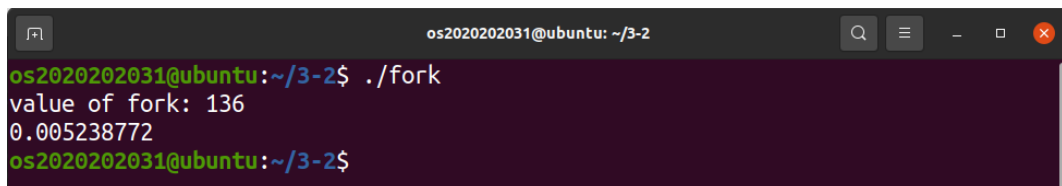
자식프로세스에서 부모프로세스로 값을 넘겨줄 때 `exit()`을 사용합니다. 반환 값은 2^8 이상이면 안되며, 8-bit 만큼 right shift 해주어야 child process 가 반환된 값을 정상적으로 확인할 수 있습니다.

반환 값이 0~255 까지로 제한되는 이유는, 자식 프로세스가 부모 프로세스에 종료상태를 8 비트만으로 표현하기 때문입니다.

또한 자식 프로세스가 `exit` 으로 부모 프로세스에 종료 상태 값을 넘겨줄 때, 값을 8-bit left shift 하여 넘겨주기 때문에, 8-bit 만큼 right shift 해주어야 child process 가 반환된 값을 정상적으로 확인할 수 있는 것입니다.

1) `MAX_PROCESSES = 8`

$$\sum_{i=1}^{16} i = 136$$

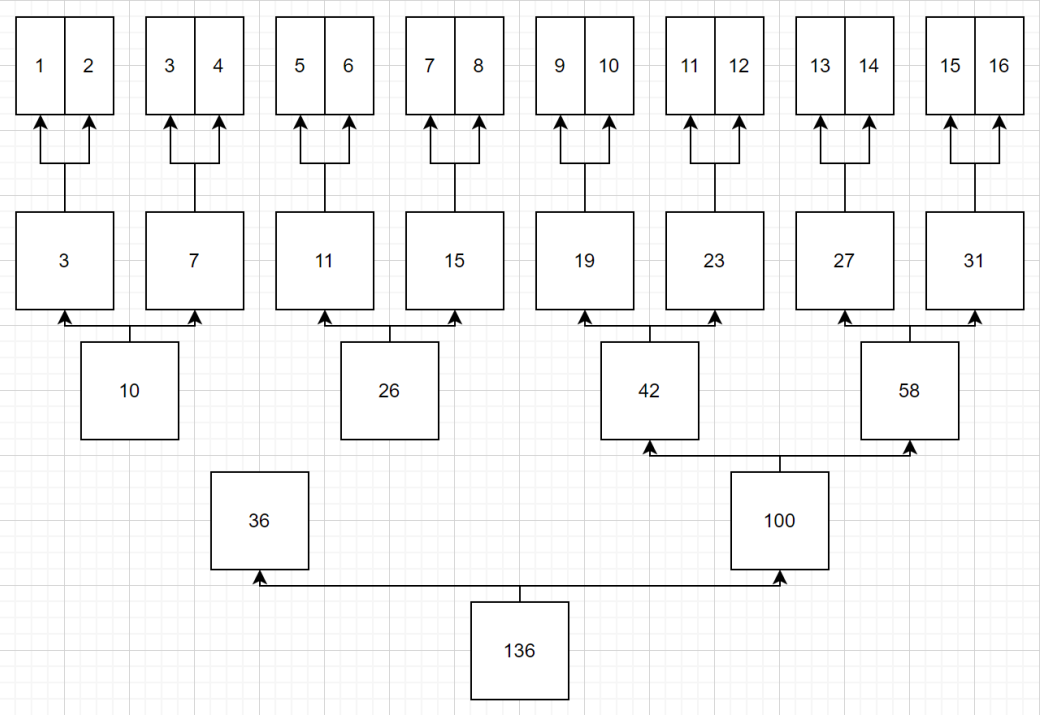


```
os2020202031@ubuntu: ~/3-2
os2020202031@ubuntu:~/3-2$ ./fork
value of fork: 136
0.005238772
os2020202031@ubuntu:~/3-2$
```

```
temp.txt
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 3
18 7
19 11
20 15
21 19
22 23
23 27
24 31
25 10
26 26
27 42
28 58
29 36
30 100
31 136
```

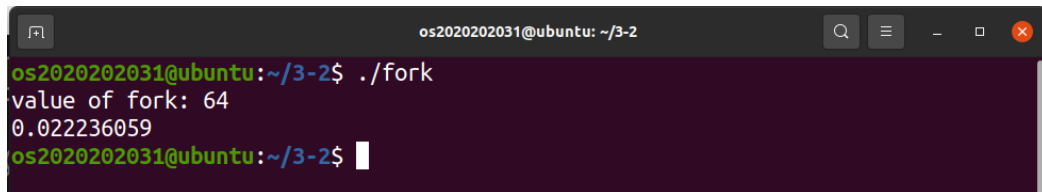
```
os2020202031@ubuntu:~/3-2$ ./thread
value of thread: 136
0.003899762
os2020202031@ubuntu:~/3-2$
```

```
temp.txt
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 3
18 7
19 11
20 15
21 19
22 23
23 27
24 31
25 10
26 26
27 42
28 58
29 36
30 100
31 136
```



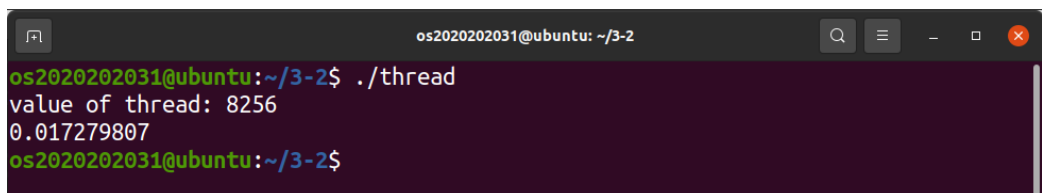
2) MAX_PROCESSES = 64c

$$\sum_{i=1}^{128} i = 8256$$



```
os2020202031@ubuntu: ~/3-2
os2020202031@ubuntu:~/3-2$ ./fork
value of fork: 64
0.022236059
os2020202031@ubuntu:~/3-2$
```

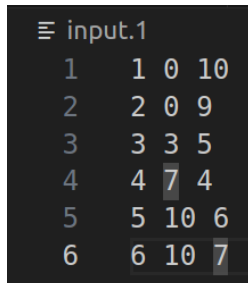
Fork 의 경우 자식 프로세스가 `exit()`으로 부모 프로세스에게 256 이상의 값을 전달할 때 8 비트 이상의 상위 비트는 잘리게 되므로, 결과 값이 $8256 \% 256 = 64$ 로 일정하게 나옵니다. 그 이유는 임의의 두 수 a, b 의 나머지연산의 합과 a, b 의 합의 나머지연산은 같기 때문입니다.



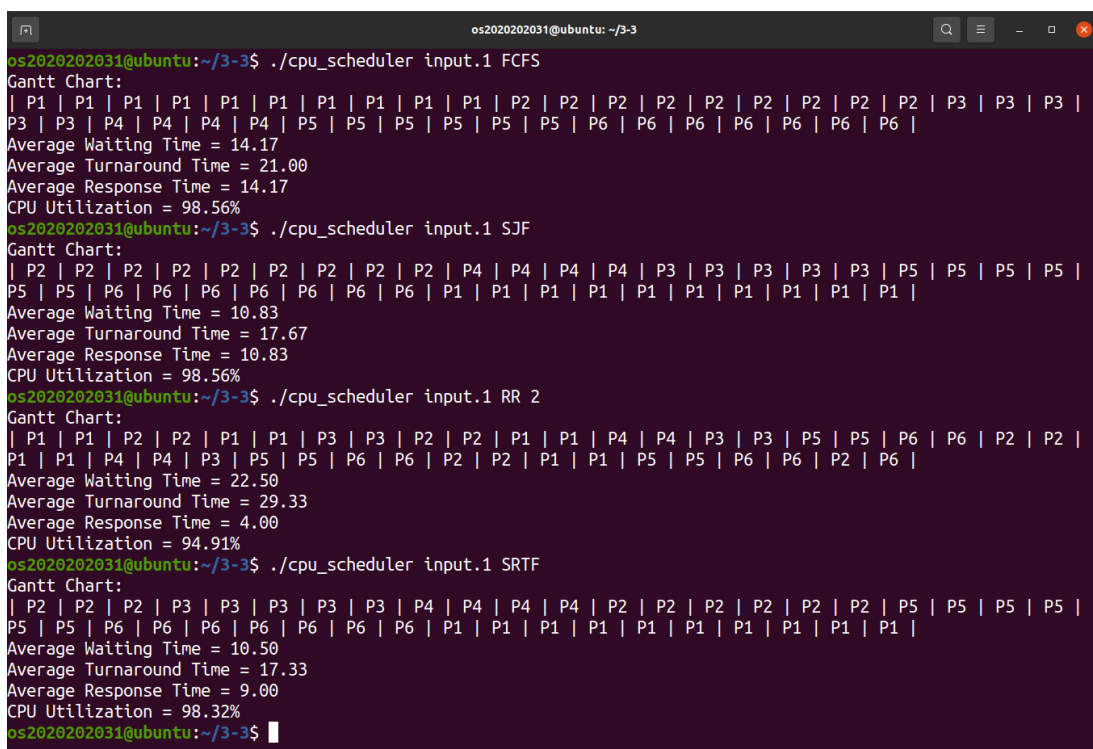
```
os2020202031@ubuntu: ~/3-2
os2020202031@ubuntu:~/3-2$ ./thread
value of thread: 8256
0.017279807
os2020202031@ubuntu:~/3-2$
```

3-3

1. Input.1



≡ input.1
1 1 0 10
2 2 0 9
3 3 3 5
4 4 7 4
5 5 10 6
6 6 10 7



```
os2020202031@ubuntu: ~/3-3
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler input.1 FCFS
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | 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%
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler input.1 SJF
Gantt Chart:
| P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P4 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P3 | P5 | P5 | P5 |
P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | 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 = 98.56%
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler input.1 RR 2
Gantt Chart:
| P1 | P1 | P2 | P2 | P1 | P1 | P3 | P3 | P2 | P2 | P1 | P1 | P4 | P4 | P3 | P3 | P5 | P5 | P6 | P6 | P2 | P2 |
P1 | P1 | P4 | P4 | P3 | P5 | P5 | P6 | P6 | P2 | P2 | P1 | P1 | P5 | P5 | P6 | P6 | P2 | P6 |
Average Waiting Time = 22.50
Average Turnaround Time = 29.33
Average Response Time = 4.00
CPU Utilization = 94.91%
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler 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 |
Average Waiting Time = 10.50
Average Turnaround Time = 17.33
Average Response Time = 9.00
CPU Utilization = 98.32%
os2020202031@ubuntu:~/3-3$
```

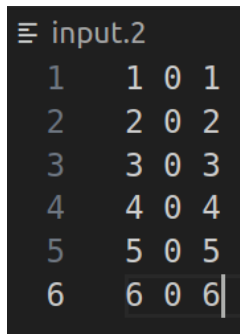
FCFS 와 SJF 는 non-preemptive 이므로 average waiting time = average response time 입니다. 또한, 한 번 프로세스가 시작하면 끝날 때까지 context switching 이 발생하지 않으므로 두 알고리즘의 CPU Utilization 이 같습니다.

Average Waiting Time, Average Turnaround Time, Average Response Time 모든 측면에서 SJF 가 성능이 좋습니다.

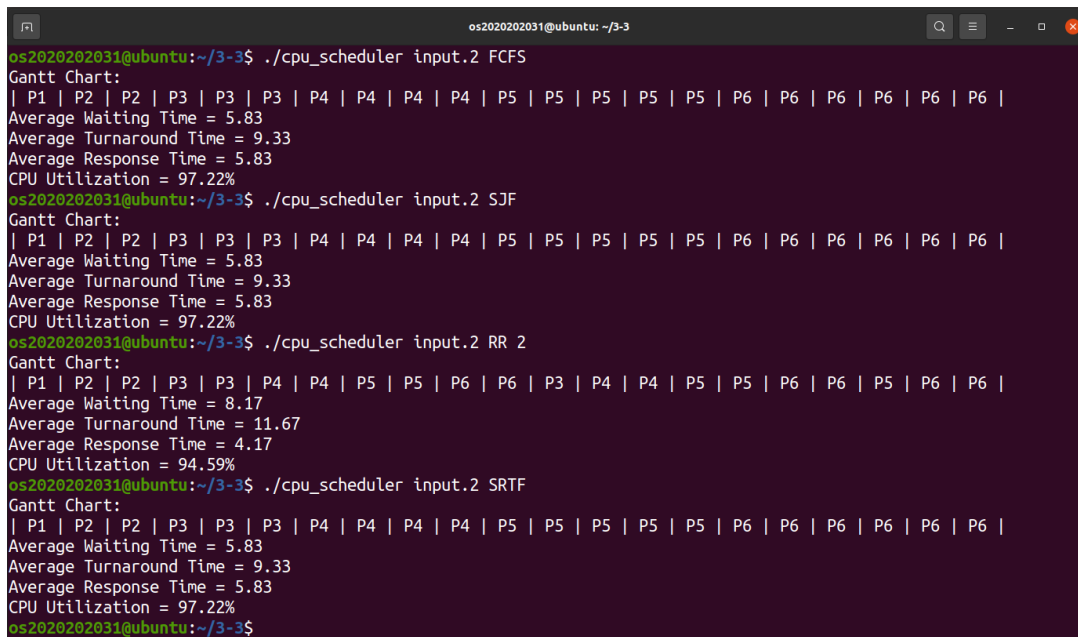
RR 과 SRTF 는 preemptive 이므로 입력이 어떻게 들어오느냐에 따라 average waiting time = average response time 이 아닐 수 있습니다. RR 은 time quantum 마다 한 번씩 계속해서 context switching 을 진행하기 때문에 CPU Utilization 이

떨어지는 것을 확인할 수 있습니다. 그에 비해 SRTF 는 CPU Utilization 도 높고, Average Waiting Time, Average Turnaround Time, Average Response Time 측면에서도 SJF 와 비슷합니다. 따라서 input.1 에서는 SJF 와 SRTF 의 성능이 우수하다고 할 수 있습니다.

2. Input.2



1	1	0	1
2	2	0	2
3	3	0	3
4	4	0	4
5	5	0	5
6	6	0	6



```
os2020202031@ubuntu: ~/3-3
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler input.2 FCFS
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting Time = 5.83
Average Turnaround Time = 9.33
Average Response Time = 5.83
CPU Utilization = 97.22%
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler input.2 SJF
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting Time = 5.83
Average Turnaround Time = 9.33
Average Response Time = 5.83
CPU Utilization = 97.22%
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler input.2 RR 2
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P4 | P4 | P5 | P5 | P6 | P6 | P3 | P4 | P4 | P5 | P5 | P6 | P6 | P5 | P6 | P6 |
Average Waiting Time = 8.17
Average Turnaround Time = 11.67
Average Response Time = 4.17
CPU Utilization = 94.59%
os2020202031@ubuntu:~/3-3$ ./cpu_scheduler input.2 SRTF
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting Time = 5.83
Average Turnaround Time = 9.33
Average Response Time = 5.83
CPU Utilization = 97.22%
os2020202031@ubuntu:~/3-3$
```

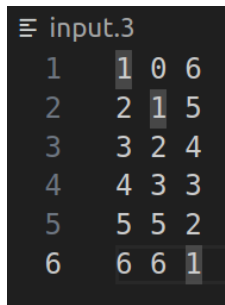
FCFS 와 SJF 는 non-preemptive 이므로 average waiting time = average response time 입니다. 또한, 한 번 프로세스가 시작하면 끝날 때까지 context switching 이 발생하지 않으므로 두 알고리즘의 CPU Utilization 이 같습니다.

Input.2 의 경우 모든 프로세스가 동시에 들어오므로, SJF 의 경우 프로세스 간의 priority 가 그 즉시 정해집니다. 이번 예시에서는 FCFS 도 SJF 와 똑같은 priority 가 부여되어, Average Waiting Time, Average Turnaround Time, Average Response Time 모든 측면에서 FCFS, SJF 가 성능이 같습니다.

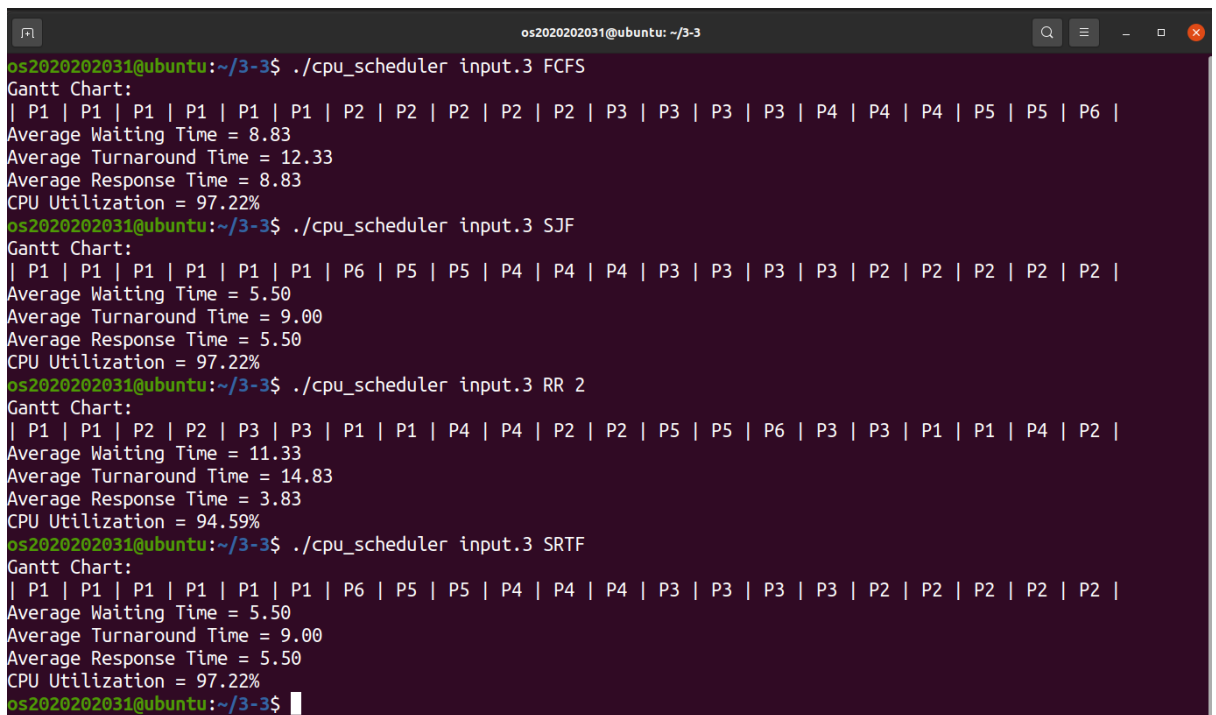
Input.2 는 프로세스 1, 2, 3, 4, 5, 6 순서로 burst_time 이 길어지므로, SRTF 또한 FCFS, SJF 와 모든 측면에서 성능이 같습니다.

RR 과 SRTF 는 preemptive 이므로 입력이 어떻게 들어오느냐에 따라 average waiting time = average response time 이 아닐 수 있습니다. RR 은 time quantum 마다 한 번씩 계속해서 context switching 을 진행하기 때문에 CPU Utilization 이 떨어지는 것을 확인할 수 있습니다.

3. Input.3



1	1	0	6
2	2	1	5
3	3	2	4
4	4	3	3
5	5	5	2
6	6	6	1



```
os2020202031@ubuntu: ~/3-3$ ./cpu_scheduler input.3 FCFS
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 | P2 | P2 | P3 | P3 | P3 | P3 | P4 | P4 | P4 | P5 | P5 | P6 |
Average Waiting Time = 8.83
Average Turnaround Time = 12.33
Average Response Time = 8.83
CPU Utilization = 97.22%
os2020202031@ubuntu: ~/3-3$ ./cpu_scheduler input.3 SJF
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P6 | P5 | P5 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P2 | P2 | P2 | P2 | P2 |
Average Waiting Time = 5.50
Average Turnaround Time = 9.00
Average Response Time = 5.50
CPU Utilization = 97.22%
os2020202031@ubuntu: ~/3-3$ ./cpu_scheduler input.3 RR 2
Gantt Chart:
| P1 | P1 | P2 | P2 | P3 | P3 | P1 | P1 | P4 | P4 | P2 | P2 | P5 | P5 | P6 | P3 | P3 | P1 | P1 | P4 | P2 |
Average Waiting Time = 11.33
Average Turnaround Time = 14.83
Average Response Time = 3.83
CPU Utilization = 94.59%
os2020202031@ubuntu: ~/3-3$ ./cpu_scheduler input.3 SRTF
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P6 | P5 | P5 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P2 | P2 | P2 | P2 | P2 |
Average Waiting Time = 5.50
Average Turnaround Time = 9.00
Average Response Time = 5.50
CPU Utilization = 97.22%
os2020202031@ubuntu: ~/3-3$
```

SJF 의 경우 처음에는 P1 을 실행하지만, P1 의 burst time 인 6ms 가 지나면, 모든 프로세스가 대기 중이기 때문에 burst time 이 작은 순서인 P6, 5, 4, 3, 2 순으로 스케줄링 됩니다.

SRTF 도 마찬가지로 처음에는 P1 을 실행합니다. 그러나 매 ms 가 지날 때마다 들어오는 프로세스들은 P1 과 remaining time 이 같아지므로 P1 을 끝까지 진행합니다. 따라서 이후에는 SJF 와 똑같이 동작하게 되므로, 모든 성능이 같게 나옵니다.

RR 은 response time 을 줄일 수 있지만 다른 측면에서 성능이 뒤쳐지기 때문에, input.3 에서는 SJF 와 SRTF 의 성능이 가장 좋다고 할 수 있습니다.

고찰

fork_count 변수를 task_struct 구조체의 상단에 위치시켰는데 커널패닉이 발생했습니다. 커널컴파일은 주소 하나하나가 민감하게 작동하므로, 변수를 상단에 선언하므로써 구조체 레이아웃이 변경되고, 기존에 task_struct 를 참조하던 컴포넌트들의 오프셋이 변경되어 문제를 발생한다고 생각했습니다. 따라서 task_struct 구조체의 하단에 위치하는 randomized_struct_fields_end 바로 위에 fork_count 를 선언했습니다.

C 언어는 struct 구조체 선언과 동시에 멤버변수를 초기화할 수 없습니다. fork_count 를 초기화하는 방법에 대해 고민했습니다. 모든 프로세스는 결국 다른 프로세스의 자식이라는 점을 떠올려 _do_fork 가 호출될 때 자식 프로세스의 fork_count 를 0 으로 초기화함으로써 모든 프로세스는 fork_count 가 0 인 채로 프로세스를 시작할 수 있게됩니다.

Reference