운영체제

Assignment #3

Class : A

Professor : 김태석 교수님

Student ID : 2022202065

Name : 박나림

Introduction

이번 과제는 3 가지로 나누어서 진행하게 된다. 첫번째로, 전달받은 pid 를 가진 프로세스의 정보를 출력하는 모듈을 작성한다. 총 8 가지의 정보로, 프로세스 이름, 현재 프로세스의 상태, 프로세스 그룹 정보, 해당 프로세스를 실행하기 위해 수행된 context switch 횟수, fork()를 호출한 횟수, 부모, 형제, 자식 프로세스의 정보를 출력하는 것이 목표이다. 두번째로, Fork 와 Thread 의 수행 시간을 측정하여 비교 분석해본다. 세번째로, CPU scheduling simulator 를 제작한다. 해당 과제들을 통해 프로세스 처리에 대한 이해도를 높이도록 한다.

결과화면

Assignment 3-1

1. 커널 수정

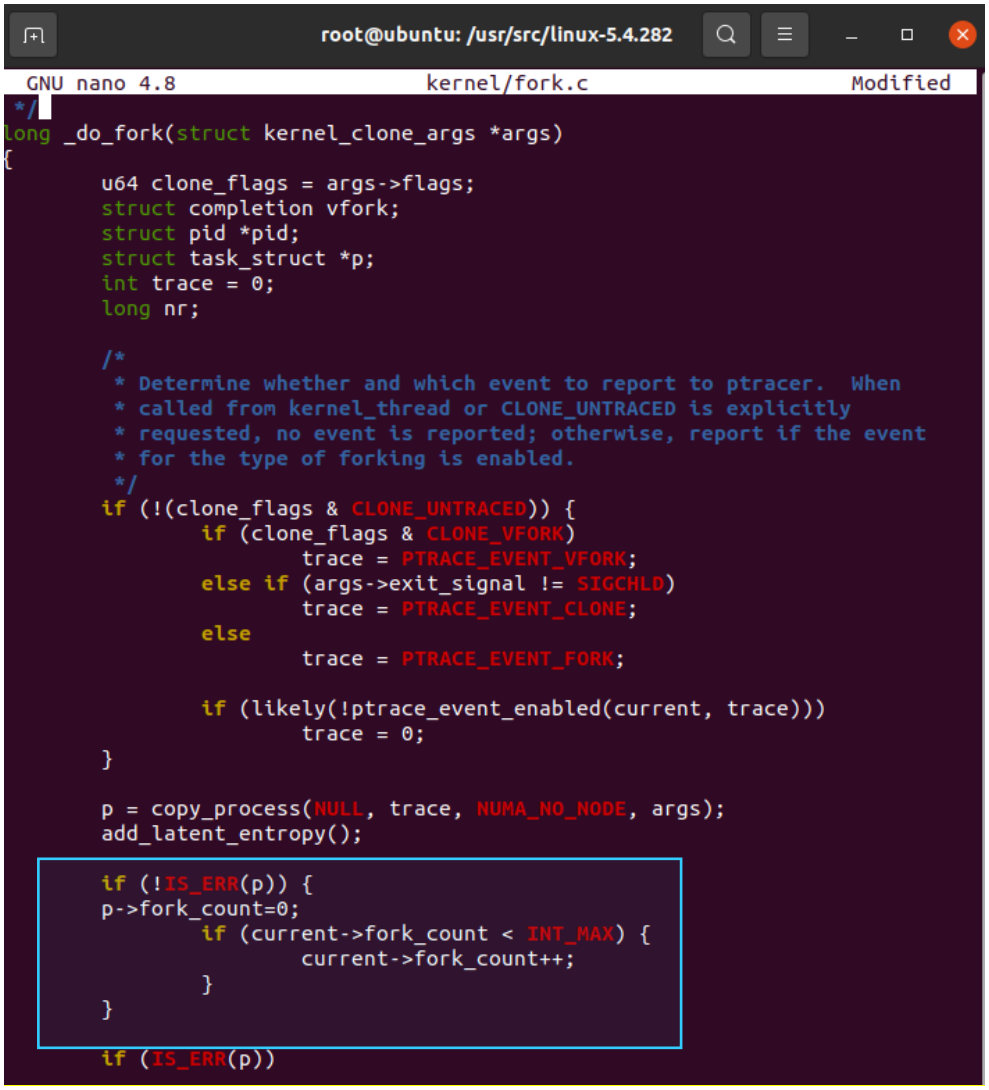
Process_tracer 모듈에서 출력할 fork count 를 계산하기 위해, task_struct 구조체를 수정해야 한다. 아래 경로의 파일 sched.h 에서 struct task_struct 를 검색한다.

```
root@ubuntu:/usr/src/linux-5.4.282# nano include/linux/sched.h
```

프로세스 생성과 관련된 부분에 int fork_count 변수를 추가한다.

```
#endif
        int                             on_rq;

        int                             prio;
        int                             static_prio;
        int                             normal_prio;
        unsigned int                    rt_priority;

        int                             fork_count; //Assignment 3-1

        const struct sched_class        *sched_class;
        struct sched_entity             se;
        struct sched_rt_entity          rt;
#ifdef CONFIG_CGROUP_SCHED
```

그 다음으로 fork()가 호출될 때마다 fork_count 를 증가시키기 위해 /kernel/fork.c 에 코드를 추가한다. Fork 가 실행될 때 호출되는 함수를 수정해야 되기 때문에 long _do_fork()에 적용한다. 기존 코드에서 copy_process 가 호출되는 시점 아래에 추가한다. 그러면 자식 프로세스를 생성하고 나서 에러가 발생하지 않으면 해당 프로세스의 fork_count 를 0 으로 초기화 한 후, INT_MAX 를 넘지 않는 선에서 fork()가 호출될 때마다 fork_count 가 증가하게 된다.



커널 수정을 마친 뒤, 커널을 컴파일 한다. (make clean -> make -j$(nproc) -> sudo make modules_install -> sudo make install -> sudo update-grub -> sudo reboot)

2. Process_tracer.c & Makefile

Process_tracer.c 에서, 2 차 과제에서 작성한 ftrace(336) 시스템 콜을 wrapping 하여 asmlinkage pid_t process_tracer 로 대체시킨다.

```
static int __init process_tracer_init(void)
{
        // Find system call table
        syscall_table = (void**) kallsyms_lookup_name("sys_call_table");

        // Change permission of the page of system call table(read, write)
        make_rw(syscall_table);
        real_os_ftrace = syscall_table[__NR_os_ftrace];
        syscall_table[__NR_os_ftrace] = process_tracer;

        return 0;
}
```

process_tracer 모듈 함수에서 8 가지 정보들을 printk 로 출력하도록 작성한다. 형제, 자식 프로세스를 출력할 때는 이름이 각각 sibling, child 이고 주소가 list 인 tast_struct 타입 자료구조의 주소를 반환하도록 만든다. 이걸 list_for_each 를 통해 각 주소로 지정된 모든 원소를 순환하여 list 에 반환되도록 만들면 해당하는 프로세스 전체를 출력할 수 있다.

```
    /* Information output from the process */
    printk(KERN_INFO "##### TASK INFORMATION of ''[%d] %s'' #####\n",trace_task, task-
>comm); //(1) process name

    printk(KERN_INFO "- task state : %s\n", stateInfo(task)); //(2) current process state

    printk(KERN_INFO "- Process Group Leader : [%d] %s\n",
           task_pgrp_nr(task), task->group_leader->comm); //(3) process group information

    printk(KERN_INFO "- Number of context switches : %lu\n",
           task->nvcsw + task->nivcsw); //(4) number of context switches

    printk(KERN_INFO "- Number of calling fork() : %d\n", task->fork_count); //(5) Number
of times fork() was called

    //(6) parent process information
    parent_task = task->real_parent;
    printk(KERN_INFO "- it's parent process : [%d] %s\n",
           parent_task->pid, parent_task->comm);

    //(7) sibling processes information
    printk(KERN_INFO "- it's sibling process(es) :\n");
    list_for_each(list, &parent_task->children) {
        struct task_struct *sibling = list_entry(list, struct task_struct, sibling);
        if (sibling->pid != task->pid) {
            printk(KERN_INFO "     > [%d] %s\n", sibling->pid, sibling->comm);
            sibling_count++;
        }
    }
    if (sibling_count == 0) {
        printk(KERN_INFO "     > It has no sibling.\n");
    }
    else {
        printk(KERN_INFO "     > This process has %d sibling process(es)\n", sibling_count);
    }

     //(8) child processes information
     printk(KERN_INFO "- it's child process(es) :\n");
     list_for_each(list, &task->children) {
         struct task_struct *child = list_entry(list, struct task_struct, sibling);
         printk(KERN_INFO "     > [%d] %s\n", child->pid, child->comm);
         child_count++;
     }
     if (child_count == 0) {
         printk(KERN_INFO "     > It has no child.\n");
     }
     else {
         printk(KERN_INFO "     > This process has %d child process(es)\n", child_count);
     }
```

Task 의 state 정보들은 stateInfo 함수로 만들어서 case 문을 통해 문자열을 반환하도록 작성하였다. task->state 의 define 되어있는 매크로 변수에 따라 7 가지의 상태로 구분된다.

```c
/* Information on task status */
static char *stateInfo(struct task_struct *task) {
    switch (task->state) {
        case TASK_RUNNING:
            return "Running or ready";
        case TASK_UNINTERRUPTIBLE:
            return "Wait with ignoring all signals";
        case TASK_INTERRUPTIBLE:
            return "Wait";
        case __TASK_STOPPED:
            return "Stopped";
        case EXIT_ZOMBIE:
            return "Zombie process";
        case EXIT_DEAD:
            return "Dead";
        default:
            return "etc.";
    }
}
```

코드 작성을 완료한 후 해당 모듈에 대한 Makefile 도 작성한다.

```
obj-m += Process_tracer.o

PWD = $(CURDIR)

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

test.c 에서는 wrapping 한 시스템 콜 336 을 호출하여 pid 1 (systemed)를 전달한다.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main()
{
        syscall(336, 1);

        return 0;
}
```

gcc test.c 로 a.out 파일을 만든 뒤, 해당 디렉토리를 make 한다음 모듈을 적재한다.

```
os2022202065@ubuntu:~/Assignment3/Assignment3-1$ sudo insmod Process_tracer.ko
```

./a.out 으로 테스트를 실행하고 dmesg 로 확인해보면 아래와 같이 프로세스 정보들을 출력한 결과를 확인할 수 있다.

```
[ 2093.447630] ##### TASK INFORMATION of ''[1] systemd'' #####
[ 2093.447633] - task state : Wait
[ 2093.447635] - Process Group Leader : [1] systemd
[ 2093.447637] - Number of context switches : 27135
[ 2093.447638] - Number of calling fork() : 190
[ 2093.447640] - it's parent process : [0] swapper/0
[ 2093.447641] - it's sibling process(es) :
[ 2093.447642]     > [2] kthreadd
[ 2093.447643]     > This process has 1 sibling process(es)
```

```
[ 2093.447644] - it's child process(es) :
[ 2093.447646]     > [314] systemd-journal
[ 2093.447648]     > [353] systemd-udevd
[ 2093.447650]     > [366] vmware-vmblock-
[ 2093.447652]     > [649] systemd-resolve
[ 2093.447653]     > [651] systemd-timesyn
[ 2093.447655]     > [663] VGAuthService
[ 2093.447656]     > [665] vmtoolsd
[ 2093.447658]     > [689] accounts-daemon
[ 2093.447659]     > [690] acpid
[ 2093.447660]     > [694] avahi-daemon
[ 2093.447662]     > [695] cron
[ 2093.447664]     > [698] dbus-daemon
[ 2093.447665]     > [700] NetworkManager
[ 2093.447666]     > [714] irqbalance
[ 2093.447668]     > [720] networkd-dispat
[ 2093.447669]     > [723] polkitd
[ 2093.447671]     > [733] rsyslogd
[ 2093.447672]     > [737] snapd
[ 2093.447674]     > [738] switcheroo-cont
[ 2093.447675]     > [740] systemd-logind
[ 2093.447676]     > [752] udisksd
[ 2093.447678]     > [755] wpa_supplicant
[ 2093.447680]     > [781] ModemManager
[ 2093.447682]     > [813] unattended-upgr
[ 2093.447683]     > [824] gdm3
[ 2093.447684]     > [905] whoopsie
[ 2093.447686]     > [911] kerneloops
[ 2093.447688]     > [919] kerneloops
[ 2093.447689]     > [967] rtkit-daemon
[ 2093.447691]     > [1082] upowerd
[ 2093.447693]     > [1270] colord
[ 2093.447694]     > [1325] systemd
[ 2093.447696]     > [1341] gnome-keyring-d
[ 2093.447697]     > [3980] cupsd
[ 2093.447699]     > [3982] cups-browsed
[ 2093.447700]     > This process has 35 child process(es)
[ 2093.447700] ##### END OF INFORMATION #####
os2022202065@ubuntu:~/Assignment3/Assignment3-1$
```

현재 프로세스 pid 를 전달하는 것으로 해보면 형제, 자식 프로세스가 없다는 문구가 출력된다.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main()
{
        syscall(336, getpid());

        return 0;
}
```

```
[ 2361.508736] ##### TASK INFORMATION of ''[4555] a.out'' #####
[ 2361.508747] - task state : Running or ready
[ 2361.508749] - Process Group Leader : [4555] a.out
[ 2361.508763] - Number of context switches : 0
[ 2361.508764] - Number of calling fork() : 0
[ 2361.508766] - it's parent process : [3909] bash
[ 2361.508766] - it's sibling process(es) :
[ 2361.508767]     > It has no sibling.
[ 2361.508768] - it's child process(es) :
[ 2361.508769]     > It has no child.
[ 2361.508770] ##### END OF INFORMATION #####
os2022202065@ubuntu:~/Assignment3/Assignment3-1$
```

Assignment 3-2

Numgen.c 를 작성한다. 실행 시 Temp.txt 를 생성하여 MAX_PROCESS 의 2 배만큼 i 값을 기록한다.

Open | numgen.c ~/Assignment3/Assignment3-2 | Sav

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_PROCESSES 64

int main() {
    FILE *f_write = fopen("./temp.txt", "w");
    if (f_write == NULL) {
        perror("file open error!\n");
        return -1;
    }

    //write i value twice as much as the MAX_PROCESS
    for (int i = 0; i < MAX_PROCESSES * 2; i++) {
        fprintf(f_write, "%d\n", i + 1);
    }

    fclose(f_write);

    return 0;
}
```

fork.c 에서는 생성된 자식 프로세스마다 temp.txt 에서 값을 2 개씩 읽고 더한다. 이때 부모 프로세스에게 exit()를 통해 값을 넘겨주며 8bit 만큼 right shift 해준다.

-반환 값은 2^8 이상이면 안되며, 8bit right shift 를 해주어야 하는 이유

프로세스 종료 정보가 시스템에 저장될 때, 8bit 로 제한되기 때문에 반환 값은 그 이상이면 안된다. wait() 호출 시 정보가 16bit 로 압축되는데, 그 중 종료 상태는 하위 8bit 만 차지하기 때문이다. 아래 코드처럼 부모 프로세스가 WEXITSTATUS(status)로 자식 프로세스의 종료 상태를 확인할 때 위 이유로 8bit 만큼 right shift 한다. 해당 코드에서는 미리 8bit 로 계산하여 반환하기 때문에 부모 프로세스가 자식 프로세스의 반환 값을 데이터의 손실 없이 정상적으로 확인할 수 있다.

```
clock_gettime(CLOCK_MONOTONIC, &start); //start measuring performance time
for (int i = 0; i < MAX_PROCESSES; i++) {
    pid = fork();
    if (pid < 0) {
        perror("fork error!");
        return -1;
    }
    if (pid == 0) {  //child process
        int num1, num2;
        if (read2Num(file, &num1, &num2)) {
            int sum = num1 + num2;
            exit(sum & 0xFF); //esure the exit value is 8 bits //8 bit right shift
        } else {
            exit(0);
        }
    }
}

for (int i = 0; i < MAX_PROCESSES; i++) { //parent process
    wait(&status); //wait for child process to terminate
    if (WIFEXITED(status)) {
        res += WEXITSTATUS(status);
    }
}
clock_gettime(CLOCK_MONOTONIC, &end); //End of performance time measurement
```

시간 측정은 clock_gettime()을 사용하여 fork 전 후로 start, end 로 값을 저장한다.

tread.c 에서도 clock_gettime()으로 시간을 측정하며, 여기서는 pthread 라이브러리를 이용한다. Create 로 자식 thread 를 생성하고 thread 함수를 실행한다. (2 개 값 더하기)

```
clock_gettime(CLOCK_MONOTONIC, &start); //start measuring performance time
for (int i = 0; i < MAX_PROCESSES; i++) {
    if (pthread_create(&threads[i], NULL, threadFunc_Sum, (void *)file) != 0) {
        perror("create thread error!");
        return -1;
    }
}

for (int i = 0; i < MAX_PROCESSES; i++) { //parent thread
    pthread_join(threads[i], NULL);
}
clock_gettime(CLOCK_MONOTONIC, &end); //end of performance time measurement
```

```c
void *threadFunc_Sum(void *arg) {
    FILE *file = (FILE *)arg;
    int num1, num2;

    if (fscanf(file, "%d\n%d\n", &num1, &num2) == 2) {
        //read and add two numbers
        int sum = num1 + num2;
        pthread_mutex_lock(&mutex);
        res += sum;
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}
```

Makefile 에선 3 개의 파일에 대해 작성한다. 이때 thread 에서 pthread 를 추가해야 관련 문법에 대해 오류가 발생하지 않는다.

```
all: numgen forkTest threadTest

numgen: numgen.c
	gcc -o numgen numgen.c
forkTest: fork.c
	gcc -o forkTest fork.c
threadTest: thread.c
	gcc -o threadTest thread.c -pthread
```

Make 한 뒤 numgen 을 실행하면 temp.txt 에 설정한 AX_PROCESS 의 2 배만큼 써진다.

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ ./numgen
```

Open ▾ temp.txt ~/Assignment3/Assignm...

```
1
2
3
4
5
6
7
8
```

매 실험 전마다 아래 명령어들로 캐시 및 버퍼를 비워서 실험에 영향을 주는 요소를 제거한다.

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ rm -rf tmp*
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ sync
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2022202065:
3
```

아래는 차례대로 프로세스를 늘려서 측정한 결과이다.

-MAX_PROCESS: 4

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ ./forkTest
value of fork : 36
0.000929
```

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ ./threadTest
value of thread : 36
0.000492
```

-MAX_PROCESS: 8

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ ./forkTest
value of fork : 136
0.001890
```

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ ./threadTest
value of thread : 136
0.000710
```

-MAX_PROCESS: 64

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ ./forkTest
value of fork : 8256
0.014419
```

```
os2022202065@ubuntu:~/Assignment3/Assignment3-2$ ./threadTest
value of thread : 8256
0.003447
```

-결과 분석

처음 4 개의 프로세스로 했을 땐 약간의 차이만 났다가, 마지막에 64 개의 프로세스로 실행한 결과에서는 차이가 크게 났다. 대체적으로 fork 보다 thread 가 빠르게 측정되는데, 프로세스가 많아질수록 thread 가 더 효율적이라는 것을 알 수 있다.

Fork 는 부모 프로세스 메모리 공간의 전체를 copy 하기 때문에 같은 메모리 공간을 공유하는 thread 보다 더 오래 걸린다. 또한 context switching 에서도, fork 는 os 가 메모리 매핑 변경 등 더 많은 작업을 해야 하는 반면 thread 는 메모리 공간과 리소스를 공유하기 때문에, 메모리 맵 전환 없이 최소 레지스터 값 전환만 필요하므로 오버헤드가 적다. Fork 는 프로세스를 생성할 때마다 새 address space 를 할당하고 커널에서 추가 작업을 해야되기 때문에 thread 의 생성, 종료가 fork 보다 더 빠르게 이루어진다.

Assignment 3-3

각 cpu scheduler 알고리즘을 테스트 하기 위한 input.1 파일을 생성한다. 총 6 개의 프로세스 예제이며, 순서대로 PID, Arrival time, Burst time 을 나타낸다.

```
1 0 10
2 0 9
3 3 5
4 7 4
5 10 6
6 10 7
```

입력으로 들어오는 프로세스들의 시간과 flag 를 기록하기 위해 구조체를 만들고, maximum queue 의 길이가 1000 이므로 해당 구조체 배열을 1000 길이로 만들어 준다. 또한 gantt chart 를 출력할 때 pid 와 해당 pid 의 정보를 나타내기 위해서 2 차원 배열로 생성한다. 종류 별 시간을 기록하기 위해 현재 시간과 idle 시간을 double 로 전역변수로 설정해준다.

```
typedef struct prc{
        int pid;
        int arrivalTime;
        int burstTime;

        double waitTime; // turnaround - burst
        double turnTime; // completion - arrival
        double resTime; // start - arrival

        double startTime; //first run
        double remainingTime;
        double completionTime;

        int start; //start flag
        int complete; //end flag
} process;
```

```
int pn = 0; //process number
process proc[1000]; //ready queue
char ganttChart[10000][5];
double currTime = 0.0, totalIdleTime = 0.0;
```

1) First Come First Served (FCFS)

첫번째 알고리즘으로, FCFS 는 큐처럼 도착 순서에 따라 선입선출 구조로 프로세스가 실행된다. 따라서 qsort 를 이용하여 input 파일에서 읽어온 프로세스들을 도착시간에 따라 정렬을 해준 다음, 현재 실행 시간보다 도착시간이 작을 때, 즉 이미 도착해 있는 프로세스들에 대하여 끝날 때까지 반복문을 통해 실행된다. 처음 시작한 프로세스는 start time 과 flag 를 설정해주고, 남은 시간을 1 초씩 감소시킨다. 이때 gantt chart 에도 sprint 로 pid 값을 넣어준다. 실행이 끝났으면 완료 시간과 flag 를 설정하고, context switching time 으로 0.1 을 더해준다. 만약 현재 시간에 도착한 프로세스가 없으면 예외처리를 해준다.

```
48 void FCFS() {
49         int slot = 0; //slot of gantt chart
50         int completeProc = 0; //completed process
51         int currProc = 0; //Currently selected process
52
53         //Sort by arrival order
54         qsort(proc, pn, sizeof(process), compareProc);
55
56         while (completeProc < pn) {
57                 if (proc[currProc].arrivalTime <= currTime && currProc < pn) { //already
   arrived
58                         if(!proc[currProc].start) {
59                                 proc[currProc].startTime = currTime;
60                                 proc[currProc].start = 1; //start flag
61                         }
62                         sprintf(ganttChart[slot], "|P%d|", proc[currProc].pid);
63                         currTime += 1;
64                         proc[currProc].remainingTime -= 1;
65                         slot += 1;
66
67                         //End of process execution
68                         if (proc[currProc].remainingTime <= 0) {
69                                 proc[currProc].complete = 1;
70                                 proc[currProc].completionTime = currTime;
71                                 completeProc += 1;
72                                 currProc += 1; //next process
73                                 currTime += 0.1; //add context switch time
74                         }
75                 }
76                 else { //Nothing has arrived yet
77                         sprintf(ganttChart[slot], "|x|");
78                         currTime += 1;
79                         slot += 1;
80                         totalIdleTime +=1;
```

2) Shortest Job First (SJF)

두번째로 SJF 는 도착한 것들 중에서 제일 적은 burst time 을 가진 프로세스를 실행하는 알고리즘이다. 따라서 반복할 때 추가로 burst time 을 검사하여 더 짧은 것을 현재 실행할 프로세스로 선택하여 끝날 때까지 실행한다. (비선점 방식)

```
while (completeProc < pn) {
        int shortBurst = 999;
        currProc = -1;

         for (int i=0; i<pn; i++) {
                 //Select the shortest time in the current time
                 if (!proc[i].complete) { //not done
                         if(proc[i].arrivalTime <= currTime) {//already arrived
                                 if (proc[i].burstTime < shortBurst) { //Update to the
t one
                                         shortBurst = proc[i].burstTime;
                                         currProc = i;
                                 }
                         }
                 }
         }
```

3) Shortest Remaining Time First (SRTF)

위를 선점 방식으로 만든 SRTF 알고리즘은 단순히 burst time 만 보는 것이 아니라, 그때마다 남은 시간을 비교하여 적은 시간을 가진 프로세스를 실행하도록 바꾼다. 위 알고리즘에서 busrt time 이 아니라 remaining time 으로 비교하도록 수정한다.

```
while (completeProc < pn) {
        int currProc = -1; //shortestJob
        int shortRemaining = 999;

        //Select the shortest time in the current time
        for (int i = 0; i < pn; i++) {
                if (!proc[i].complete) { //not done
                        if(proc[i].arrivalTime <= currTime) {
                                if (proc[i].remainingTime < shortRemaining) {
                                        shortRemaining = proc[i].remainingTime;
                                        currProc = i;
                                }
                        }
                }
        }
```

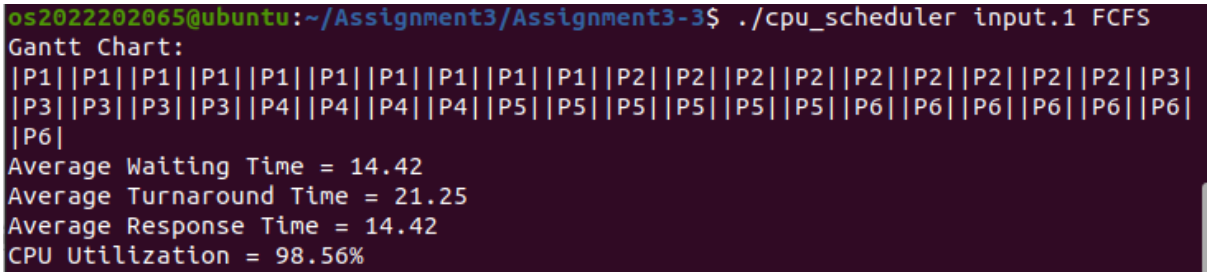
4) Round Robin (RR)

마지막으로 Round Robin 알고리즘은 time quantum 을 추가로 입력 받아서 해당 시간 만큼만 실행하는 방식이다. Ready queue 를 1000 길이로 만들어서 완료되지 않은 프로세스들을 push 한 뒤, front 로 하나씩 꺼내서 quantum 만큼만 실행했다가 끝나지 않으면 다시 queue 의 맨 뒤로 push 한다. 이러한 방식으로 끝날 때까지 돌아가면서 수행하도록 만든다.

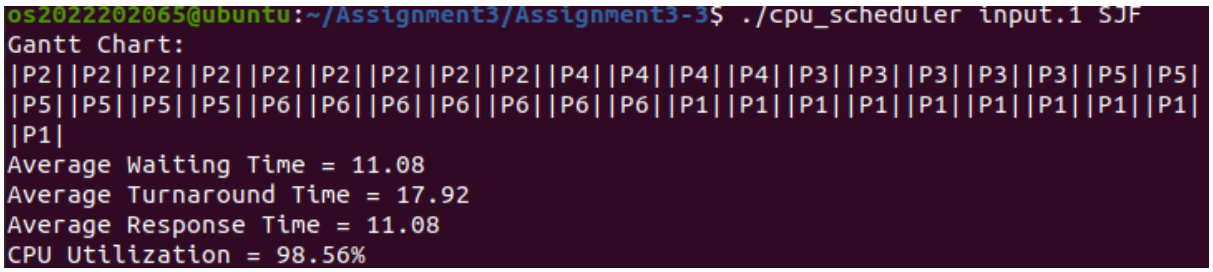
```
while (completeProc < pn) {
        while (currProc < pn && proc[currProc].arrivalTime <= currTime) { //already

                if (!proc[currProc].complete) { //not done
                        rq.push(currProc); //push to ready queue
                }
                currProc++; //next process
        }

        if (rq.empty()) { //ready queue is empty and not complete
                if (completeProc < pn) {
                        sprintf(ganttChart[slot], "|x|"); //Nothing has arrived yet
                        currTime += 1;
                        slot += 1;
                        totalIdleTime += 1;
                        continue;
                }
        }

        if (!rq.empty()) { //process in ready queue
                int curr = rq.front(); //select current process(front of ready queue)
                rq.pop();

                //when running for the first time
                if (!proc[curr].start) {
                        proc[curr].startTime = currTime;
                        proc[curr].start = 1; //start flag
                }

                //choose the shorter one between quantum and remaining time
                int executeTime = std::min(quantum, (int)proc[curr].remainingTime);
                for (int i = 0; i < executeTime; i++) { //execute process
                        sprintf(ganttChart[slot], "|P%d|", proc[curr].pid);
                        currTime += 1;
                        proc[curr].remainingTime -= 1;
                        slot += 1;
                }
```

각 알고리즘을 순서대로 실행한 결과이다.

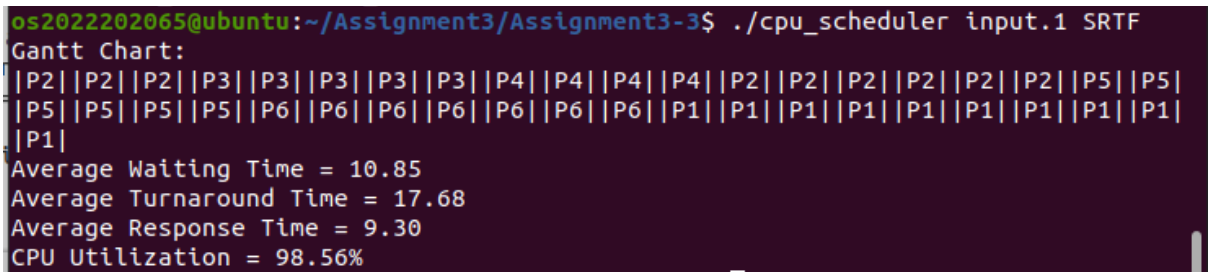
FCFS 는 도착 순서대로 실행되는 것을 알 수 있다.



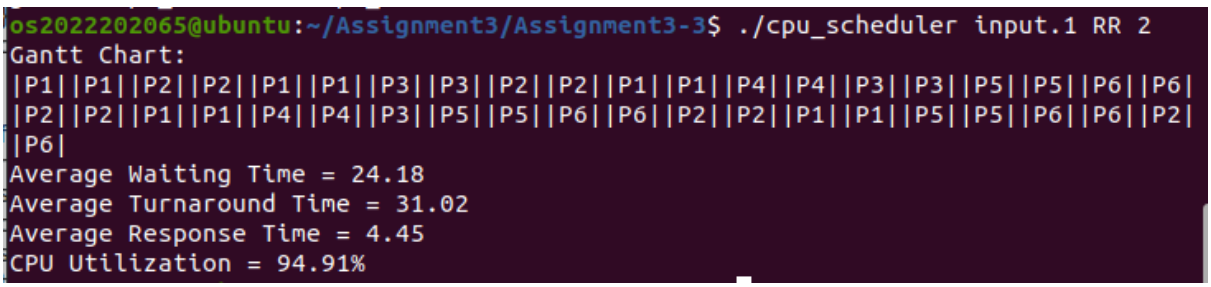
SJF 는 도착한 순서 중에서 burst time 이 적은 것을 먼저 실행하는 것을 볼 수 있다.



SRTF 는 도착한 순서 중에서 remaining time 이 적은 것으로 바꿔가며 실행하는 것을 볼 수 있다.



RR 은 time quantum 을 2 로 했을 때, 2 번씩만 실행하면서 queue 에 있는 순서대로 실행하는 것을 알 수 있다.



각 알고리즘의 waiting time, turnaround time, response time, cpu utilization 을 보면, 단순한 구조인 FCFS 는 모든 시간들이 다 오래 걸린다. SJF 는 빠른 순서대로 실행하기 때문에 모든 시간들이 조금 더 개선되었으며, 이를 선점형으로 바꾼 SRTF 는 더 개선된

것을 알 수 있다. 다만 3 방식 모두 response time 은 느린데, 이는 RR 에서 높게 개선된 것을 볼 수 있다. Time quantum 마다 실행하기 때문에 이를 줄일수록 더 빨라질 것이다. 하지만 그만큼 완료되기까지의 시간이 걸리기 때문에 waiting time 과 turnaround time 이 훨씬 크게 측정된다. 또한 cpu utilization 도 RR 이 낮게 나타난다. 따라서 turnaround time 등을 우선으로 고려할 때는 SRTF 알고리즘, response time 을 우선으로 고려할 때는 RR 알고리즘이 낫다는 것을 확인할 수 있다.

고찰

3-1

Process_tracer 모듈을 적재한 후 실행하여 dmesg 를 확인해 보니, "Process with PID 15744856 not found"로 종료되었다. Pid 를 1 로 전달했음에도 불구하고 알 수 없는 값이 전달된 것이다. 디버그 문구가 출력되었으니 기존 os_ftrace 시스템콜이 wrapping 된 것은 알 수 있었으나, pid 값이 다르게 전달되어서 커널을 재컴파일 했을 때 문제가 생긴 것이라 추측하였다. 하지만 모듈을 삭제하고 336 번을 호출해보니 설정한 pid 값대로 잘 전달되었다. Process_tracer 에서만 pid 값을 제대로 받지 못하는 것 같아 인자 형태를 바꾸었다. 원래 pid_t trace_task 로 받았었다가, 저번 과제처럼 const struct pt_resg *regs 로 받은 뒤에 pid 만 추출하여 확인해보니 정상적으로 값이 전달되는 것을 볼 수 있었다. 그냥 pid 로 받을 땐 왜 전달이 안되는지 아직 더 공부해봐야 할 것 같다.

3-2

파일을 컴파일할 때, "undefined reference to pthread_create", "undefined reference to `pthread_join" 오류가 발생하였다. 스레드를 사용한 코드를 컴파일할 때 POSIX 라이브러리가 필요한데, 이를 위해서 Makefile 에 -pthread 플래그를 써주어야 하는 것을 빠뜨렸기 때문이다. 플래그를 추가하니 정상적으로 컴파일 될 수 있었다.

3-3

Input.1 파일을 읽는 과정에서, 파일의 끝까지 읽기 위해 while(!feof(fp))를 사용했더니 알고리즘 적용 후 gantt chart 를 출력할 때 입력되지 않은 p0 까지 출력되고 시간 계산도 잘못 출력되는 상황이 나타났다. 파일을 읽을때부터 문제가 생긴 것 같아 read count 를 통해 3 가지 변수를 읽을 때만 반복되도록 수정하였더니 정상적으로 변수들이 저장되어 출력될 수 있었다.

Round Robin 알고리즘을 구현할 때, ready queue 가 따로 필요하다 생각하여 #include 로 <queue>헤더를 추가하여 구현하였다. 그러고 나서 make 하니 queue 를 사용한 라인들이 사용할 수 없다는 에러가 발생하였다. 이는 queue 헤더는 c++로 컴파일 되어야 하기 때문임을 깨닫고 Makefile 에서 gcc 를 g++로 수정하니 컴파일 되어 실행될 수 있었다. .c 파일이라도 헤더 사용에 따라 다르게 컴파일 해야된다는 것을 깨달을 수 있었다.

Reference

강의자료 참고