

# Assignment 3

수업 명:	운영체제
담당 교수님:	최상호 교수
학번:	2018202074
이름:	김상우
강의시간:	금 1,2

## Introduction

해당 과제의 목표 3가지 기능에 대해 동작을 하는 프로그램을 만드는 것이다. 첫번째는 다중 process와 thread를 fork와 pthread를 이용해 파일들의 sum을 진행하는 코드를 만드는 것이다. 두번째는 CPU 스케줄링(standard\_RR, FIFO, RR)에 대해 priority와 nice를 수정해가며 각 결과를 비교하는 것이다. 그리고 마지막은 pid를 바탕으로 프로세스의 정보를 출력하는 module을 작성하는 것이다. 이러한 과정을 통해 컴퓨터를 process와 thread를 기준으로 어떻게 동작시킬 수 있는 지 이해하고, CPU scheduling이 어떠한 영향을 미치는지, 또 process의 정보들은 어떤 식으로 사용되는 지 알아보고자 한다.

## Conclusion & Analysis

### Assignment #3-1

#### -numgen.c

```
#include <stdio.h>

#define MAX_PROCESSES 64

int main()
{
    FILE *f_write = fopen("temp.txt", "w"); //open file to write
    for(int i=0;i<MAX_PROCESSES*2;i++)      //loop to print
    {
        fprintf(f_write, "%d\n", i+1); //write in file
    }
    fclose(f_write);                      //close file
    return 0;
}
```

위는 numgen.c를 위해 작성한 코드이다. 위를 보면, temp.txt파일을 fopen을 통해 생성하고 w, 즉 쓰기 모드로 이를 엽니다. 이후 MAX\_PROCESS의 두배만큼 fprintf를 이용해 숫자를 순서대로 적습니다. 이후 fclose를 이용해, 파일을 닫습니다.

#### Fork.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

#include <time.h> //use clock_gettime

#define MAX_PROCESSES 8
```

Process를 이용한, 코드 수행을 위한 header와 MAX\_PROCESSES를 언급합니다.

```

int main()
{
    FILE* f_read = fopen("temp.txt", "r");
    //open txt file to read
    struct timespec start, end;
    //variable to save time
    pid_t pid;

    int result = 1; //variable to save result
    int dep = 1; //variable to save node's depth

    system("rm -rf tmp");
    system("sync");
    system("echo 3 | sudo tee /proc/sys/vm/drop_caches");
    //initialize

```

이후 main함수를 작성합니다. temp에서 숫자를 읽기 위해 fopen을 이용해 temp.txt를 read모드로 open합니다. 이후 시간측정을 위해 timespec을 선언하고, fork를 위한 pid와 tree에 대한 깊이를 나타낼 dep, 그리고 최종 결과를 나타낼 result를 선언합니다.

이후 system함수를 이용해 rm -rf tmp, sync, echo 3 | sudo tee /proc/sys/vm/drop\_caches를 이용해 캐시와 버퍼를 비워 실험에 영향을 주는 요소를 제거합니다.

```

clock_gettime(CLOCK_MONOTONIC, &start); //time to start
pid = fork(); //make child process
if(pid==0){ //in child process
    while(1){ //for loop
        int left, right;
        if(MAX_PROCESSES < dep*2){ //when node in tree
            if(EOF==fscanf(f_read, "%d", &left)){
                left = 0;
            }
            if(EOF==fscanf(f_read, "%d", &right)){
                right = 0;
            }
            //take txt's values
            result = left+right;
            exit(result);
            //add and take result
        }
    }
}

```

이후 clock\_gettime을 이용해 시작 시간을 측정합니다. 이후 tree의 형태로 만들기 위해 fork를 해 줍니다. While을 이용해 무한 루프를 만들어주고, 내부에서 tree의 왼쪽, 오른쪽의 값을 받기 위한 left, right함수를 만들어줍니다. 이후 binary tree는 node의 수가  $dep*2 - 1$ 인 점을 이용  $dep^2$ 보다 작을 때 아래 코드가 진행될 수 있도록 조건문을 걸어줍니다. Fscanf를 이용해 temp의 값을 받아 오며 읽을 것이 없을 경우 0으로 받을 수 있도록 하고 left와 right를 더해 exit로 result를 내보낼 수 있도록 합니다.

```

if(!fork()){ // for left node
    dep*=2;
    continue;
}

wait(&left); //take left result

if(!fork()){ //for right node
    dep*=2;
    continue;
}

wait(&right); //take right result
result = (left>>8)+(right>>8); //return add result
exit(result);
}

waitpid(pid, &result, 0); //wait final result
clock_gettime(CLOCK_MONOTONIC, &end); // time end

```

이후 fork를 이용해 left, right node를 표현하기 위해 fork를 이용해 node수가 2배인 아래 depth에 대해 처리할 수 있게 만들어줍니다. 이후 해당 값들을 wait하여 값을 받아오며 값들을 shift후 더하게 됩니다. 이후 더한 결과를 exit를 통해 result를 보내줍니다. 이후 waitpid를 통해 pid에 대해 결과를 기다립니다. 이후 clock\_gettime을 이용해 끝난 시간을 저장합니다.

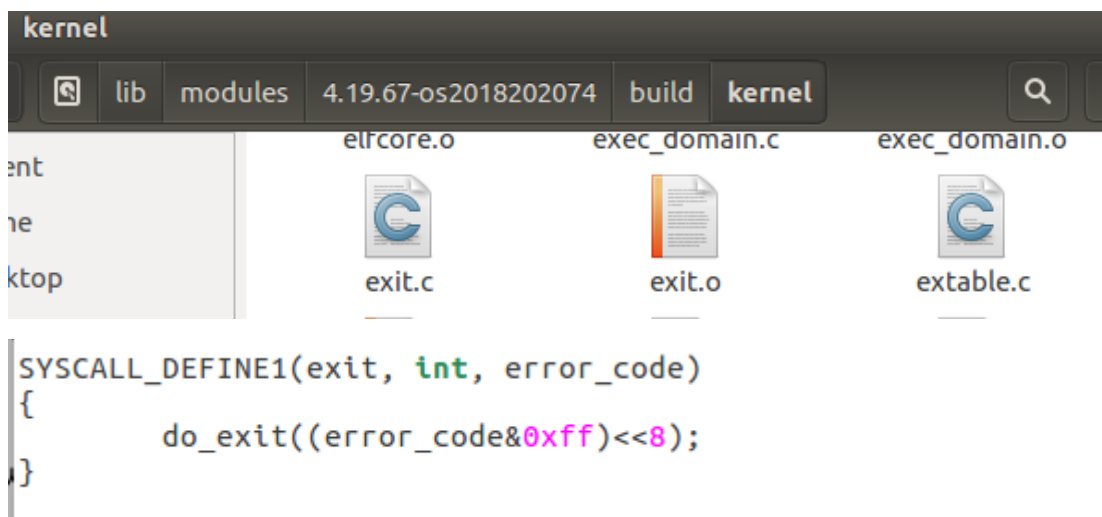
```

        if(result < 1<<8){
            printf("value of fork : %d\n", result); //print final fork
        }
        else{
            printf("value of fork : %d\n", result>>8);
        }
        double time = (end.tv_sec- start.tv_sec) + ((end.tv_nsec -
start.tv_nsec)/1000000000.0);
        printf("%lf\n", time); //print time
        fclose(f_read); //file close
        return 0;
    }

```

이후 result가 shift가 필요 없을 땐 그대로, shift가 필요할 경우 shift를 위해 조건문으로 이를 구분 해줍니다. 이후 time을 앞에서 구한 값들을 기반으로 처리하고 파일을 닫으며 끝냅니다.

이때 위에서 계속해 shift를 이용해 반환값이 2의 8승이 되지 않도록 유지하는 것을 볼 수 있다. 이는 exit의 system call이 담긴 exit.c파일에서 이유를 찾을 수 있다.(위치는 아래와 같다.)



위를 보면 exit에 대해 argument사용시 0xff와 &연산을 진행하므로 2^8이상의 값에 대해서는 함수로 정확히 전달하지 못하는 것이다. 또한 이후 8bit만큼 left shift를 진행하므로 원래 값을 찾기 위해 8bit만큼 right shift를 진행하는 것이다.

## Thread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>

#include <time.h> //use clock_gettime

#define MAX_PROCESSES 64

FILE * f_read; //to file read

void* thread_sum(void * input) //function to sum
{
    int dep = *((int*)input); //tree's depth save
    int left; //take left node value
    int right; //take right node value
    static int result; //save result
    if(MAX_PROCESSES < dep*2){ //at leaves
        if(EOF==fscanf(f_read, "%d", &left)){
            left =0;
        }
        if(EOF==fscanf(f_read, "%d", &right)){
            right =0;
        }
        //take txt file's numbers
    }
}
```

사용할 헤더와 MAX\_PROCESS, 그리고 파일 open을 위한 변수를 미리 선언합니다. 이후 sum을 위해 함수를 선언합니다. 깊이에 대해 Input을 인자로 받고 왼쪽 노드와 오른쪽 노드 값을 저장할 변수, 최종결과를 저장할 변수도 선언합니다. 이후 유효한 tree내 node수에 대해 leaves에 대해 처리하기 위해 아닌 경우에 대해 조건을 걸고 fscanf로 값과 없을 경우에 대해 값을 받을 수 있게 합니다.

```
    else //nodes are not leaves
    {
        int subdep = dep * 2; //for child node's depth
        pthread_t tid; //threadid to take child's result
        pthread_t tid2;
        int * sub_left; //sub node's left right value
        int * sub_right;

        pthread_create(&tid, NULL, thread_sum, (void*)&subdep);
        //make thread like left child node
        pthread_join(tid, (void**)&sub_left);
        //take left child node's value
        left = (*sub_left);

        pthread_create(&tid2, NULL, thread_sum, (void*)&subdep);
        //make thread like right child node
        pthread_join(tid2, (void**)&sub_right);
        //take right child node's value
        right = (*sub_right);
    }
    result=left+right;
    //add two value
    pthread_exit((void*)&result);
    //return add value
}
```

이후, leaves가 아닌 경우, 파일에서 값을 받는 것이 아닌, 하위 node 값을 받아야 하므로 이에 대해 처리해주어야 합니다. 이를 위해 하위 노드 처리를 위해 dep를 2배 한 후, 하위 node를 만들기 위한 변수들을 선언해 줍니다. Pthread create를 이용해 왼쪽, 오른쪽 노드에 대해 처리를 진행하고, join을 통해 결과를 받아옵니다. 이렇게 받은 값을 left와 right에 저장합니다. 이후 result에 더한 값을 저장하고 exit를 통해 결과를 반환합니다.(상위 노드에 전달)

```
int main()
{
    f_read = fopen("temp.txt", "r");
    //open file to read values
    struct timespec start, end;
    //check time
    pthread_t tid;
    //variable to save thread id
    int *result;
    int dep = 1;    //first depth

    system("rm -rf tmp");
    system("sync");
    system("echo 3 | sudo tee /proc/sys/vm/drop_caches");
}
```

이후 temp.txt파일을 read모드로 열고 시간 측정을 위해 timespec을 선언합니다. 이후 결과와 tree 깊이를 표현하는 등에 대한 변수를 선언하고 캐쉬와 버퍼를 비우기 위해 system함수를 이용해 아래 같이 선언합니다.

```
clock_gettime(CLOCK_MONOTONIC, &start); //time to start

pthread_create(&tid, NULL, thread_sum, (void*)&dep); //call sub-thread
pthread_join(tid, (void**) &result); //block until sub-thread & return

clock_gettime(CLOCK_MONOTONIC, &end); //clock end

printf("value of fork : %d\n", *result); //print fork
double time = (end.tv_sec - start.tv_sec) + ((end.tv_nsec -
start.tv_nsec)/1000000000.0);
printf("%lf\n", time); //print time
fclose(f_read); //close file
return 0;
}
```

Pthread\_create와 join을 이용해 위에서 작성한 function를 실행하고, 이후 clock\_gettime를 앞뒤로 배치해 이를 통해 시간을 측정합니다. 이렇게 측정된 값을 이용해 시간을 측정하고 출력합니다.



```
Makefile (~/.assign_3/3-1) - gedit

CC = gcc

all: hello hello2 hello3

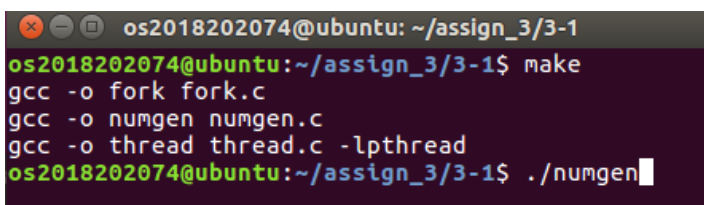
hello: fork.c
    $(CC) -o fork $^

hello2: numgen.c
    $(CC) -o numgen $^

hello3: thread.c
    $(CC) -o thread $^ -lpthread

clean:
    rm hello hello2 hello3
```

Makefile은 위와 같이 작성하여 각 c파일에 대해 실행파일을 생성하도록 하였다.

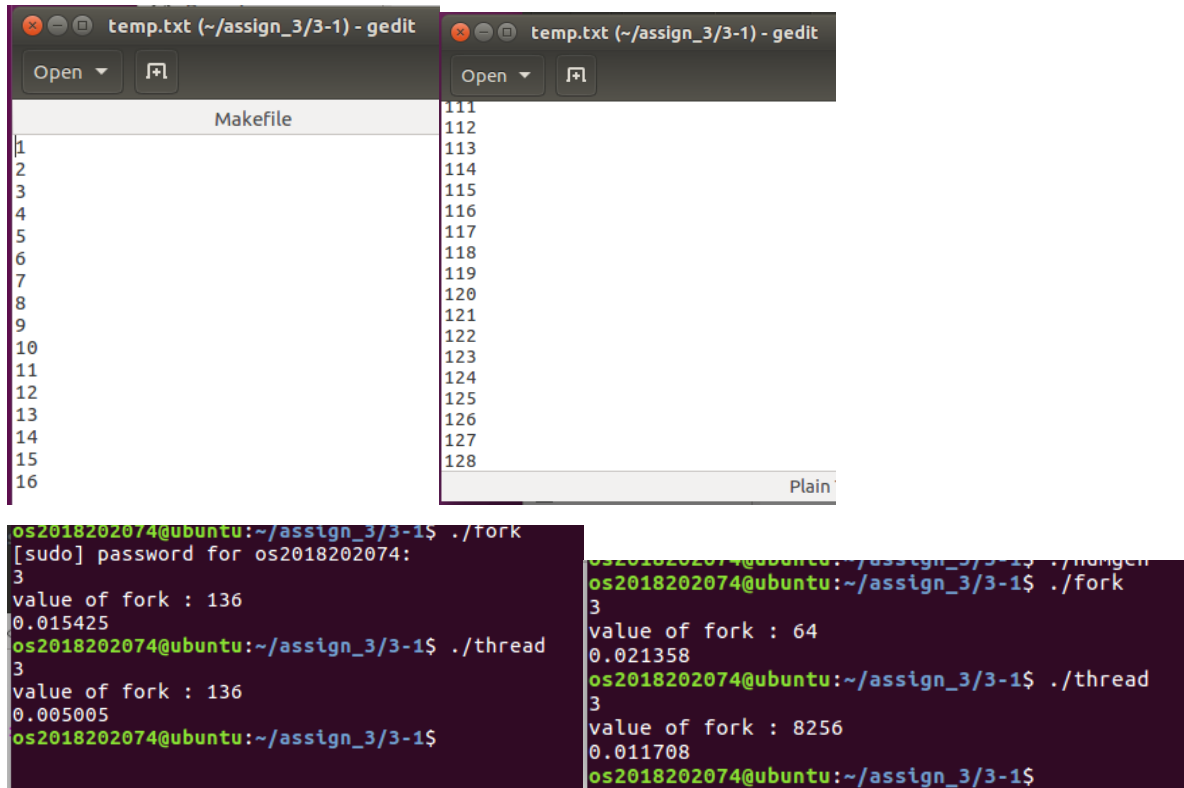


```
os2018202074@ubuntu: ~/assign_3/3-1
os2018202074@ubuntu:~/assign_3/3-1$ make
gcc -o fork fork.c
gcc -o numgen numgen.c
gcc -o thread thread.c -lpthread
os2018202074@ubuntu:~/assign_3/3-1$ ./numgen
```

이후 make를 이용, 모든 c파일에 대해 실행파일을 만들고 ./+실행파일 명 을 이용해 실행한다.

Numgen의 경우 아래와 같이 결과가 나온다.

각각, MAX\_PROCESSES를 8, 64한 결과로 원하는 대로 출력되었다.



```
temp.txt (~/.assign_3/3-1) - gedit
Makefile
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

temp.txt (~/.assign_3/3-1) - gedit
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
Plain

os2018202074@ubuntu:~/assign_3/3-1$ ./fork
[sudo] password for os2018202074:
value of fork : 136
0.015425
os2018202074@ubuntu:~/assign_3/3-1$ ./thread
3
value of fork : 136
0.005005
os2018202074@ubuntu:~/assign_3/3-1$

os2018202074@ubuntu:~/assign_3/3-1$ ./fork
value of fork : 64
0.021358
os2018202074@ubuntu:~/assign_3/3-1$ ./thread
3
value of fork : 8256
0.011708
os2018202074@ubuntu:~/assign_3/3-1$
```

위의 결과들은 각각 fork.c thread.c의 MAX\_PROCESSES가 8, 64일 때의 실행결과이다. ECHO에 의해 3이 공통적으로 출력되며 MAX\_PROCESSES가 8일때는 1부터 16까지의 합인 136이 적절하게 나오며 속도는 thread일때 빠르게 동작함을 확인할 수 있었다. 이는 thread간의 처리가 접근에 있어 속도가 빨라 유리하기 때문이다. 이러한 점은 MAX\_PROCESSES가 64일때도 같다. 단, fork의 경우 64로 처리할 경우 fork가 64가 나옴을 확인할 수 있었다. 이는 thread가 1부터 64까지의 더한 값을 잘 나타낸 것과 대조적이다. 이는 위에서 언급한 exit의 특성 때문이다. 8bit까지 처리하는 exit의 특성상  $2^8$ 이상의 수를 표현할 수 없으며 그 때문에 하위 비트들만 계산이 진행되어 64만이 남게 된 것이다.



하위 8bit만 할 경우  $2^6 = 64$ 만이 남음을 확인했다.

## Assignment #3-2

### Filegen.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

#define MAX_PROCESSES 10000

int main(){
    char name[100];                //to save take file name
    mkdir("./temp", 0777);         //make folder "temp"
    for(int i=0; i < MAX_PROCESSES; i++){ //make MAXPROCESSES number folder
        sprintf(name, "./temp/%d", i); //to make file name
        FILE * f_write = fopen(name, "w"); //make file
        fprintf(f_write, "%d\n", 1+rand()%9); //and put random number
        fclose(f_write);           //close file
    }
    return 0;
}
```

위는 filegen의 코드이다. 위를 보면 우선 temp파일을 0777권한으로 생성하고(read write exec모두 허용) 이후 for문을 이용 MAX\_PROCESSES개의 파일을 해당 코드에서 다룰 수 있도록 하였다. 반복문 내부에서 sprintf를 이용해 파일명과 위치를 처리하고 fopen을 이용해 temp내에 i에 해당하는 파일을 쓰기 권한을 준 상태로 생성한다. 이후 fprintf를 이용해 1+(0-8)의 값을 무작위로 넣어 준 후 파일을 닫는다.

### schedtest.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<sys/syscall.h>
#include<time.h>
#include<math.h>

#include<sched.h>           //SCHED header
#include<sys/wait.h>        //to wait each process
#include<sys/mman.h>        //to use mmap function

#define MAX_PROCESSES 10000

static double *total;      //sum of each processes run time
static int *file_num=0;
```

이후 위 같은 헤더들과 함께, MAX\_PROCESSES의 값. 그리고 전체 실행시간을 저장할 변수와 file의 개수를 측정하기 위한 file\_num을 전역변수로서 만들어준다.



```

void schedtest(int type,int priority,int nice_val)
{
    char f_name[128]; //variable to save file name
    int wait_val;      //variable to wait
    int read_val;      //variable to read value

    //use mmap to match address
    total = mmap(NULL, sizeof(double), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
    *total = 0;

```

이후 측정을 위해 동작을 담당할 schedtest 함수를 선언한다. file이름과 wait, read를 위한 variable을 선언한다. 이후 mmap을 이용, 파일을 프로그램 내 주소 공간 메모리에 대응시키고 초기화를 진행한다.

```

for(int i=0; i<MAX_PROCESSES; i++){ //make process about MAX_PROCESSES
    if(!fork()){ //make child process
        struct sched_param sched_set;

        //if standard_RR, we need to nice value
        if(type == SCHED_OTHER){
            sched_set.sched_priority = 0;
            sched_setscheduler(0, type, &sched_set);
            nice(nice_val); //use nice function
        }

        //if FIFO OR RR, we need to priority
        else{
            sched_set.sched_priority = priority;
            sched_setscheduler(0, type, &sched_set);
        }

        struct timespec start,end; //to calculate run time

        clock_gettime(CLOCK_MONOTONIC, &start); //start time

        //open file and read&print file data, and close file
        FILE *f_read;
        sprintf(f_name, "./temp/%d", i);

        f_read = fopen(f_name, "r");
        fscanf(f_read, "%d", &read_val);
        fclose(f_read);

        clock_gettime(CLOCK_MONOTONIC, &end); //end time

        double time = (end.tv_sec - start.tv_sec) + ((end.tv_nsec -
start.tv_nsec)/1000000000.0); //take run time about this process

        *total += time; //add this time to calculate average turn
        around time
        exit(0); //finish process
    }
}

```

이후 MAX Processes만큼, 즉 모든 파일에 대응하는 process에 scheduling을 적용하기 위해 반복문을 걸고 이에 대한 정보를 받아줄 sched\_param을 선언한다. 이때 non real time을 사용하므로 이후 nice value를 사용해 standard\_RR이 사용될 경우, sched\_setscheduler를 이용해 현재의 cpu Scheduling방식과 더불어 priority를 0으로 적용시키고 nice함수를 이용해 nice val을 우선 처리하도록 하였다.

만약 FIFO나 Round Robin을 이용하는 경우, Real time을 기준으로 계산을 진행한다. 이를 설정해 주기 위해 sched\_set.sched\_priority를 이용, 우선도를 설정해준다.

이후 시작과 끝 time을 기록할 timespec함수를 2개를 선언한다. Clock\_gettime을 이용해 측정 구역 앞뒤로 시간을 측정한다. 내부는 sprintf를 이용해 파일 위치를 찾고 이를 read 모드로 fopen을 통해 파일을 열고, fscanf를 통해 값을 받아온다. 이후, 파일을 fclose로 닫아줍니다. 이후 시간을 계산 하여 총 시간에 더해주었습니다.

```

//wait all process end
for(int i = 0; i < MAX_PROCESSES; i++){
    wait(&wait_val);
}

double average = (*total)/MAX_PROCESSES; //make average turn around time
munmap(total,sizeof(double)); //free mmap

//print average turn around time
printf("Average turn around time about files : %lf\n", average);
}

```

이후 모든 프로세스들이 끝남을 기다리고, 끝난 후 평균 turnaround time을 구하기 위해 이를 Process수만큼 나눠줍니다. 이후 mmap으로 할당된 total을 빼준 후, 시간을 출력해줍니다.

```

int main()
{
    system("rm -rf tmp");
    system("sync");
    system("echo 3 | sudo tee /proc/sys/vm/drop_caches");
    //initialize

    int type=0; //variable to save cpu scheduling
    int prior=0; //variable to save priority
    int nice=0; //variable to save nice value
    struct timespec start, end; //variable to calculate time

    printf("CPU scheduling(1=standard_RR 2=FIFO 3=RR) : "); //set cpu scheduling
    scanf("%d",&type);
}

```

system함수를 이용해 버퍼를 비워주고 값을 입력받기 위한 변수들과 시간을 저장할 timespec 변수를 지정합니다. 이후 scanf를 통해 CPU scheduling방식을 입력받습니다.

```

if(type==1){ //standard_RR case
    type=SCHED_OTHER;
    printf("NICE(-20=high 0=def 19=low) : "); //take nice value
    scanf("%d",&nice);
}
else if(type==2){ //FIFO case
    type=SCHED_FIFO;
    printf("Priority(1=high 2=def 3=low) : ");
    scanf("%d",&prior);
    if(prior==1){ //set priority value
        prior=sched_get_priority_min(type);
    }
    else if(prior==2){
        prior=(sched_get_priority_min(type)+sched_get_priority_max
(type))/2;
    }
    else if(prior==3){
        prior=sched_get_priority_max(type);
    }
}
else if(type==3){ //RR case
    type=SCHED_RR;
    printf("Priority(1=high 2=def 3=low) : ");
    scanf("%d",&prior);
    if(prior==1){ //set priority value
        prior=sched_get_priority_min(type);
    }
    else if(prior==2){
        prior=(sched_get_priority_min(type)+sched_get_priority_max
(type))/2;
    }
    else if(prior==3){
        prior=sched_get_priority_max(type);
    }
}
}

```

만약 standard\_RR으로 정해졌다면, type을 SCHED\_OTHER로 바꿔주고 nice값으로 쓰기 위한 값을 입력받습니다. 만약 realtime을 사용하는 fifo나 RR이 입력될 경우 type에 이에 맞는 값을 넣고 priority를 입력받습니다. 그리고 이에 맞춰 sched\_get\_priority\_min/max함수를 이용해 high, default, low의 경우에 대한 priority값을 받아줍니다.

```

        clock_gettime(CLOCK_MONOTONIC, &start); //start time of full processes
        schedtest(type, prior, nice);          //run schedtest function
        clock_gettime(CLOCK_MONOTONIC, &end);   //end time of full processes

        double time = (end.tv_sec- start.tv_sec) + ((end.tv_nsec -
start.tv_nsec)/1000000000.0);
        printf("Full time of process : %lf\n", time); //print full processes time
        return 0;
}

```

Clock\_gettime을 이용해 위에서 선언한 함수가 돌아가는 시간을 측정할 수 있도록 하고, 위에서 입력을 통해 받아진 값들을 이용해 함수를 실행시킵니다. 이후 해당 시간을 계산 및 출력합니다

```

all: hello hello2
|
hello: schedtest.c
      $(CC) -o schedtest $^

hello2: filegen.c
      $(CC) -o filegen $^

clean:
      rm hello | rm hello2

```

이후 make를 통해 각각 c파일에 대응하는 schedtest와 filegen 실행파일을 만들수 있도록 Makefile을 생성합니다.

```

os2018202074@ubuntu:~$ cd assign_3/3-2
os2018202074@ubuntu:~/assign_3/3-2$ make
cc -o schedtest schedtest.c
cc -o filegen filegen.c
filegen.c: In function 'main':
filegen.c:11:2: warning: implicit declaration of function 'mkdir' [-Wimplicit-fu
nction-declaration]
   mkdir("./temp", 0777); //make folder "temp"
   ^
os2018202074@ubuntu:~/assign_3/3-2$ ./filegen

```

Filegen.c가 잘 작동되는 지 확인하기 위해 MAX\_PROCESSES를 10000으로 하고 위를 실행하였다.



10000개의 파일이 1부터 9사이의 랜덤한 값을 가진 상태로 생성되었다.

이후 각 경우에 대해 출력을 해보았다.

## 1.Standard\_RR

-high

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 1
NICE(-20=high 0=def 19=low) : -20
Average turn around time about files : 0.001464
Full time of process : 0.822257
```

-def

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 1
NICE(-20=high 0=def 19=low) : 0
Average turn around time about files : 0.002915
Full time of process : 0.697018
```

-low

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 1
NICE(-20=high 0=def 19=low) : 19
Average turn around time about files : 0.008284
Full time of process : 0.746081
```

Turn around time

회차	1	2	3	4	5
High	0.001428	0.001831	0.001464	0.002134	0.001540
Def	0.003571	0.003834	0.002252	0.003044	0.002746
Low	0.006315	0.006232	0.008284	0.009167	0.008883

결과는 위와 같다. Fulltime의 경우 통상적으로 nice값을 low로 줄수록 오래 걸렸으며(nice값이 high에 가까울 수록 들어오자마자 많은 cpu자원이 할당되기 때문에 추측) turnaround time을 측정한 결과, high-def-low 순으로 turnaround time이 증가, 즉 high일수록 더 빨리 진행됨을 알 수 있었습니다. 이는 non real time으로 nice값이 작을수록 cpu로부터 많은 자원을 할당받는 standard\_RR의 특징 때문이라는 것을 알 수 있습니다.

## 2.FIFO

-high

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 2
Priority(1=high 2=def 3=low) : 1
Average turn around time about files : 0.000302
Full time of process : 0.994729
```

-def

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 2
Priority(1=high 2=def 3=low) : 2
Average turn around time about files : 0.001499
Full time of process : 0.830961
```

-low

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 2
Priority(1=high 2=def 3=low) : 3
Average turn around time about files : 0.001221
Full time of process : 0.912048
```

Turn around time

회차	1	2	3	4	5
High	0.000302	0.000334	0.000510	0.000887	0.000371
Def	0.001289	0.001111	0.001110	0.001106	0.000550
Low	0.001373	0.001221	0.001242	0.001195	0.001108

결과는 위와 같다. Fulltime에 큰 변화는 없었습니다. turn around time의 경우, 위 경우 high로 갈수록 turn around time이 짧아지는 모습을 보이지만 극단적이지만 결과값에 변동이 심하며 더 많은 회차를 돌릴시 결과적으로 통상적으로 비슷하게 나오는 것을 확인할 수 있었습니다.

이는 process의 동작이 모두 같으며, realtime을 기반으로 진행하기에 이러한 결과가 나오는 것으로 추측된다.

### 3.Round Robin

-high

-def

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
[sudo] password for os2018202074:
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 3
Priority(1=high 2=def 3=low) : 1
Average turn around time about files : 0.001111
Full time of process : 0.833073

os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 3
Priority(1=high 2=def 3=low) : 2
Average turn around time about files : 0.001088
Full time of process : 1.035247
```

-low

```
os2018202074@ubuntu:~/assign_3/3-2$ sudo ./schedtest
3
CPU scheduling(1=standard_RR 2=FIFO 3=RR) : 3
Priority(1=high 2=def 3=low) : 3
Average turn around time about files : 0.001015
Full time of process : 0.861393
```

Turn around time

회차	1	2	3	4	5
High	0.001111	0.001421	0.001613	0.000923	0.001344
Def	0.001088	0.001384	0.001020	0.001101	0.001162
Low	0.001015	0.001461	0.001192	0.001081	0.001132

Round Robin은 위의 경우 process들이 다 같은 기능을 수행하고, Round Robin 자체가 특정 시간 간격을 기준으로 실행하는 만큼, priority에 관계없이 turnaround들이 비슷한 평균을 보이는 것을 확인할 수 있었다.

추가적으로 위에서 turn around 시간에 대해서는 standard\_RR이 가장 길게 걸리는 모습을 보여주었다. 이는 Real time 스케줄링이 process에 대해 우선적으로 시간을 할당하기 때문이다.

총시간의 경우 nice함수를 비롯해 FIFO, RR과 Standard\_RR의 함수내 동작 function이 조건문에 의해 달라지기 때문이다.

### Assignment #3-3

코드 작성에 앞서 아래와 같이 assignment 2에서 작성한 ftrace를 확인한다.

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi arch/x86/entry/syscalls/syscall_64.tbl
```

```
331      common  pkey_free          __x64_sys_pkey_free
332      common  statx              __x64_sys_statx
333      common  io_pgetevents      __x64_sys_io_pgetevents
334      common  rseq               __x64_sys_rseq
336      common  ftrace             __x64_sys_ftrace
#
```

위를 통해 syscall 함수에 인자로 336을 넣어 해당 함수를 불러올 수 있음을 확인했다.

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi include/linux/syscalls.h
```

```
}
asmlinkage long sys_hello(void);
asmlinkage long sys_add(int, int);
asmlinkage int ftrace(pid_t pid);
#endif
```

위같이 기존의 함수는 pid\_t pid를 인자로 받으며 ftrace라는 이름을 가진 것을 확인할 수 있었다.

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ cd ftrace
os2018202074@ubuntu:~/Downloads/linux-4.19.67/ftrace$ vi ftracehooking.c
```

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <asm/unistd.h>

SYSCALL_DEFINE1(ftrace, pid_t, pid)
{
    printk("No hijack %ld\n", pid);
    return 0;
}
~
```

또한 해당 system call은 printk를 통해 pid와 함께 위 같은 문자열을 출력하고 0을 return하는 것을 확인했다.

위와 같이 wrapping을 진행할 함수를 확인하였다.

해당 코드에서 fork한 수를 확인하여야 하지만 fork 수를 세어주는 변수는 없기에 이를 직접 만들어야 한다. 이를 받을 변수를 sched.h(linux-4.19.67/include/linux)에 선언하였다. 해당 변수는 struct task\_struct내에 선언되었다.

```

struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK

    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */

```

(생략)

```

#ifdef CONFIG_SECURITY
    /* Used by LSM modules for access restriction: */
    void *security;
#endif

    //2018202074 function plus
    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.
     */

```

해당 변수는 fork가 실행될 때마다 값이 상승해야 한다. 이를 위해 fork.c(linux-4.19.67/kernel/fork.c)에 아래와 같이 작성하여 생성시에 0으로 초기화되고 자신의 parent process의 fork\_count는 1 상승시킬 수 있도록 하였다.

```

long _do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr,
              unsigned long tls)
{
    struct completion vfork;
    struct pid *pid;
    struct task_struct *p;
    int trace = 0;
    long nr;

```

(생략)

```

    /* Do this prior waking up the new thread - the thread pointer
     * might get invalid after that point, if the thread exits quickly.
     */
    //2018202074 fork.c add line
    p->fork_count = 0;
    p->parent->fork_count++;

    trace_sched_process_fork(current, p);

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

```

이후 컴파일을 통해 다시금 위에서 진행된 처리들에 대해 업데이트 했다.

(make -j 8, sudo make modules\_install, sudo make install, reboot)

## -process\_tracer.c

```
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>

#include <asm/syscall_wrapper.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/init_task.h>

void ** syscall_table; //save system call table
void *real_ftrace; //save original ftrace function
```

코드를 작성하기 위한 헤더들과 함께 syscall table을 담을 변수, 그리고 원래의 ftrace를 저장할 변수를 선언한다.

```
void make_rw(void *addr) //function to read and write
{
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);
    if(pte->pte &~ _PAGE_RW)
        pte->pte |= _PAGE_RW;
}

void make_ro(void *addr)//function to cancel (read and write)
{
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);
    pte->pte = pte->pte &~ _PAGE_RW;
}
```

addr이 속해 있는 페이지에 읽기 및 쓰기 권한을 부여하는 함수 make\_rw를 위와 같이 선언하고 이를 통해 system call table에 대해 쓰기 권한을 취득하여 해당 model이 hijack할 수 있게 한다. Make\_ro를 통해서는 이를 원래대로 돌린다.

```
pid_t process_tracer(const struct pt_regs *regs){//hijack function
    int pid0 = regs->di;
    struct task_struct *task1; //to search input process
    struct task_struct *task; //take input process

    struct task_struct *sibling_task; //sibling process take
    struct task_struct *child_task; //children process take
    struct task_struct *parent_task; //parent process take
    struct list_head *list; //save list of sibling & children
    struct list_head *list2;

    for_each_process(task1){ //find input process
        if((int)task1->pid == pid0){ //same pid search
            task = task1; //take process
        }
    }
    int count=0; //process count(sibling, children)
    if(task == NULL){ //if no correct information
        return -1;
    }
}
```

이후 ftrace를 wrapping할시 작동하는 함수를 선언하였다. 인자로는 pt\_regs \*regs를 받으며 해당 인자에서 regs->di를 통해 인자의 pid를 알아내었다. 이후 주석에 적힌 기능을 하는 변수들을 선언하고, 받아온 pid를 바탕으로 process를 for\_each\_process를 이용해 찾는다. 이후 sibling과 children수를 세는 변수 count를 0으로 초기화 한다. 만약 인자의 pid로 동작하는 process를 찾을 수 없는 경우 오류라는 의미로 return -1을 진행하였다.



```

printk("[os2018202074] ##### TASK INFORMATION of '['[d] %s ' ' #####\n", pid0, task->comm); //process id and name

int statecheck = task->state; //state check
if(statecheck == 0){ //and print current state
    printk("[os2018202074] - task state : Running or ready\n");
}
else if(statecheck == 2){
    printk("[os2018202074] - task state : Wait with ignoring all signals\n");
}
else if(statecheck == 1){
    printk("[os2018202074] - task state : Wait\n");
}
else if(statecheck == 4){
    printk("[os2018202074] - task state : Stopped\n");
}
else if(statecheck == 32){
    printk("[os2018202074] - task state : Zombie process\n");
}
else if(statecheck == 16){
    printk("[os2018202074] - task state : Dead\n");
}
else{
    printk("[os2018202074] - task state : etc\n");
}
}

```

이후 양식에 맞추어 process id와 process의 이름을 pid0와 task->comm을 통해 출력한다. 이후 task->state를 이용해 process의 상태를 확인한다. 이를 바탕으로 받아온 값들을 조건문을 이용해 확인하고 양식에 맞게 출력해주었다.

```

//print group leader
printk("[os2018202074] - Process Group Leader : [d] %s\n", task->group_leader->pid, task->group_leader->comm);
//print context switch number
printk("[os2018202074] - Number of context switches : %ld\n", task->nivcsw);
//print fork number(use sched & fork file)
printk("[os2018202074] - Number of calling fork() : %d\n", task->fork_count);
//print about parent process
printk("[os2018202074] - it's parent process(es) : [d] %s\n", task->parent->pid, task->parent->comm);

```

이후 task\_struct의 group\_leader pid, comm을 이용해 group leader의 정보를 출력합니다. 또한 context switch의 수를 nivcsw로, fork의 수는 fork\_count로, parent의 내용은 task\_struct의 parent의 pid, comm을 이용해 parent process의 정보를 출력하였다.

```

//make list for sibling
list = &(task->sibling);
//check all sibling nodes
list_for_each(list, &task->sibling){
    sibling_task = list_entry(list, struct task_struct, sibling);
    if(sibling_task->pid != 0){ //delete self case
        printk("[os2018202074] > [d] %s\n", sibling_task->pid, sibling_task->comm);
        count++;
    }
}

if(count == 0){ //if no sibling
    printk("[os2018202074] > It has no sibling.\n");
}
else if(count > 0){ //if have sibling, print them
    printk("[os2018202074] > This process has %d sibling process(es)\n", count);
}
printk("[os2018202074] - it's child process(es) :\n");
// initialize count to check number of children
count = 0;

```

이후 task의 sibling에 대해 list에 담고, list\_for\_each와 list\_entry를 이용해 list의 sibling들에 대해 순회하고 이중 본인에 대한 것을 빼기 위해 pid가 0인지 확인합니다. Pid가 0이 아니면 pid와 comm을 이용해 출력합니다.

만약 count가 0이라면 sibling이 없는 것이므로 이에 맞는 결과를, count가 0보다 크다면 개수를 출력합니다. 이후 children에서 다시금 사용하기 위해 count를 0으로 초기화해줍니다.

```

//make list for children
list2 = &(task->children);
//check all children nodes
list_for_each(list2, &task->children){
    child_task = list_entry(list2, struct task_struct, sibling);
    if(child_task->pid != 0){//delete self case
        printk("[os2018202074] > [%d] %s\n", child_task->pid, child_task->comm);
        count++;
    }
}

if(count == 0){//if no children
    printk("[os2018202074] > It has no child.\n");
}
else if(count > 0){//if have children, print them
    printk("[os2018202074] > This process has %d child process(es)\n", count);
}

printk("[os2018202074] ##### END OF INFORMATION #####\n");
//return pid
return pid0;
}

```

이후 task의 children에 대해 list에 담고, list\_for\_each와 list\_entry를 이용해 list의 children들에 대해 순회하고 이중 본인에 대한 것을 빼기 위해 pid가 0인지 확인합니다. pid가 0이 아니면 pid와 comm을 이용해 출력합니다.(children이기에 걸리지 않을 것이지만, 재확인합니다.)

만약 count가 0이라면 children이 없는 것이므로 이에 맞는 결과를, count가 0보다 크다면 개수를 출력합니다. 이후 출력의 종료를 알리고 정상적인 종료임을 알리기 위해 return으로 인자의 pid를 출력해줍니다.

```

static int __init hooking_init(void)    //take and save original system call
{
    syscall_table = (void **) kallsyms_lookup_name("sys_call_table");
    //save system call table
    make_rw(syscall_table);
    real_ftrace = syscall_table[__NR_ftrace]; //save original function
    syscall_table[__NR_ftrace] = process_tracer;
    return 0;
}
static void __exit hooking_exit(void)    //return to original function
{
    syscall_table[__NR_ftrace] = real_ftrace; //return original function
    make_ro(syscall_table);
}
module_init(hooking_init);
module_exit(hooking_exit);
MODULE_LICENSE("GPL");

```

이후 해당 함수들에 대한 대체가 이뤄지기 전 처리를 위해 \_\_init hooking\_init를 사용, syscall table 내에 kallsyms\_lookup\_name을 통해 기존에 사용하던 sys\_call\_table, 즉 system call에 대한 표를 받아온다. 이후 real\_ftrace 내에 원래 위치에 있던 함수를 syscall\_table[system call number]를 통해 저장하고, syscall\_table[\_\_NR\_ftrace]의 위치에 대체할 함수 process\_tracer를 넣는 것을 볼 수 있다.

이는 \_\_exit hooking\_exit를 통해 원래대로 돌릴 필요가 있다. 이를 위해 syscall\_table[\_\_NR\_ftrace] 위치에 이전에 real\_ftrace에 저장했던 원래 함수를 다시 넣어준다.

이후 module\_init, module\_exit를 통해 위 코드 기반의 모듈의 각 생성/종료때 각 함수를 실행한다. Module\_license("GPL")을 통해 모듈의 라이선스는 GPL이 된다.

```

obj-m := ftracehooking.o iotracehooking.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
    gcc -o app app.c
clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean

```

Make file의 경우 ftracehooking.ko 파일과 iotracehooking.ko 파일이 동시에 생성될 필요가 있다. 이를 위해 위와 같은 코드를 작성, ftracehooking.o와 iotracehooking.o에 대해 obj-m, 즉 모듈로서 선언을 하고 KDIR을 이용, module로서의 동작이 가능하도록 kernel을 uname -r로서 확인하고 kernel내 build 위치에 Make 명령어를 쓸수 있도록 해두었으며 내부에 접근하는 데 있어 pwd를 이용해 가능하도록 하였다.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    if(!fork()){
        pid = syscall(336, getpid());    //call system calls
        return 0;
    }
    if(!fork()){
        pid = syscall(336, getpid());    //call system calls
        return 0;
    }
    sleep(10);
    printf("pid : %d\n", getpid());
    pid = syscall(336, getpid());    //call system calls
    return 0;
}

```

테스트를 위해 하나의 process가 2개의 child process를 만들고 각 process들이 한번씩 syscall을 하도록 진행하는 코드를 작성하였다.

```

obj-m := process_tracer.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
    gcc -o test app.c
clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean

```

이후 module 실행을 위해 위같이 선언하였다.

```
os2018202074@ubuntu:~/assign_3/3-3$ make
make -C /lib/modules/4.19.67-os2018202074/build SUBDIRS=/home/os2018202074/assign_3/3-3 modules
make[1]: Entering directory '/home/os2018202074/Downloads/linux-4.19.67'
  CC [M] /home/os2018202074/assign_3/3-3/process_tracer.o
/home/os2018202074/assign_3/3-3/process_tracer.c: In function 'process_tracer':
/home/os2018202074/assign_3/3-3/process_tracer.c:45:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
    int count=0;    //process count(sibling, children)
    ^
/home/os2018202074/assign_3/3-3/process_tracer.c:52:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
    int statecheck = task->state; //state check
    ^
/home/os2018202074/assign_3/3-3/process_tracer.c:36:22: warning: unused variable 'parent_task' [-Wunused-variable]
    struct task_struct *parent_task; //parent process take
                        ^
/home/os2018202074/assign_3/3-3/process_tracer.c:46:4: warning: 'task' may be used uninitialized in this function [-Wmaybe-uninitialized]
    if(task == NULL){    //if no correct information
    ^
Building modules, stage 2.
MODPOST 1 modules
CC      /home/os2018202074/assign_3/3-3/process_tracer.mod.o
LD [M] /home/os2018202074/assign_3/3-3/process_tracer.ko
make[1]: Leaving directory '/home/os2018202074/Downloads/linux-4.19.67'
gcc -o test app.c
os2018202074@ubuntu:~/assign_3/3-3$
```

이후 make를 통해 생성을 진행하였다.

```
os2018202074@ubuntu:~/assign_3/3-3$ ls
app.c  modules.order  process_tracer.c  process_tracer.mod.c  process_tracer.o
Makefile  Module.symvers  process_tracer.ko  process_tracer.mod.o  test
os2018202074@ubuntu:~/assign_3/3-3$
```

이후 아래와 같이 wrapping을 하지 않았을 때와 할 경우를 확인하고 dmesg를 통해 출력을 확인하였다.

```
os2018202074@ubuntu:~/assign_3/3-3$ dmesg
[ 8943.524010] No hijack 62823
[ 8943.524025] No hijack 62824
[ 8943.524035] No hijack 62822
```

Wrapping이 진행되지 않을 경우는 기존의 함수가 출력되었다.

```
os2018202074@ubuntu:~/assign_3/3-3$ sudo dmesg -C
os2018202074@ubuntu:~/assign_3/3-3$ ./test
pid : 62822
os2018202074@ubuntu:~/assign_3/3-3$ sudo insmod process_tracer
insmod: ERROR: could not load module process_tracer: No such file or directory
os2018202074@ubuntu:~/assign_3/3-3$ sudo insmod process_tracer.ko
os2018202074@ubuntu:~/assign_3/3-3$ ./test
pid : 62842
os2018202074@ubuntu:~/assign_3/3-3$ sudo rmmod process_tracer.ko
os2018202074@ubuntu:~/assign_3/3-3$
```

위와 같은 파일들이 생성되었다.

```

[ 9205.987681] [os2018202074] ##### TASK INFORMATION of '[63323] test ' #####
[ 9205.987682] [os2018202074] - task state : Running or ready
[ 9205.987682] [os2018202074] - Process Group Leader : [63323] test
[ 9205.987683] [os2018202074] - Number of context switches : 0
[ 9205.987683] [os2018202074] - Number of calling fork() : 0
[ 9205.987683] [os2018202074] - it's parent process(es) : [63322] test
[ 9205.987683] [os2018202074] - it's sibling process(es) :
[ 9205.987684] [os2018202074]   > [63324] test
[ 9205.987684] [os2018202074]   > This process has 1 sibling process(es)
[ 9205.987684] [os2018202074] - it's child process(es) :
[ 9205.987684] [os2018202074]   > It has no child.
[ 9205.987685] [os2018202074] ##### END OF INFORMATION #####
[ 9205.987710] [os2018202074] ##### TASK INFORMATION of '[63324] test ' #####
[ 9205.987711] [os2018202074] - task state : Running or ready
[ 9205.987711] [os2018202074] - Process Group Leader : [63324] test
[ 9205.987711] [os2018202074] - Number of context switches : 0
[ 9205.987712] [os2018202074] - Number of calling fork() : 0
[ 9205.987712] [os2018202074] - it's parent process(es) : [63322] test
[ 9205.987712] [os2018202074] - it's sibling process(es) :
[ 9205.987712] [os2018202074]   > [63323] test
[ 9205.987713] [os2018202074]   > This process has 1 sibling process(es)
[ 9205.987713] [os2018202074] - it's child process(es) :
[ 9205.987713] [os2018202074]   > It has no child.
[ 9205.987713] [os2018202074] ##### END OF INFORMATION #####

```

위는 child에 대한 출력이다. 각각의 child는 자기자신을 groupleader로 가지며, context switch나 fork가 일어나지 않았으며, parent로 pid가 63323인 test process를 지닌다. 또한 서로를 sibling이라 탐지하는 것을 확인하였으며 child가 없는 것도 확인하였다.

```

[ 9215.988291] [os2018202074] ##### TASK INFORMATION of '[63322] test ' #####
[ 9215.988292] [os2018202074] - task state : Running or ready
[ 9215.988293] [os2018202074] - Process Group Leader : [63322] test
[ 9215.988294] [os2018202074] - Number of context switches : 0
[ 9215.988294] [os2018202074] - Number of calling fork() : 2
[ 9215.988294] [os2018202074] - it's parent process(es) : [2947] bash
[ 9215.988294] [os2018202074] - it's sibling process(es) :
[ 9215.988295] [os2018202074]   > It has no sibling.
[ 9215.988295] [os2018202074] - it's child process(es) :
[ 9215.988295] [os2018202074]   > [63323] test
[ 9215.988295] [os2018202074]   > [63324] test
[ 9215.988296] [os2018202074]   > This process has 2 child process(es)
[ 9215.988296] [os2018202074] ##### END OF INFORMATION #####

```

위는 parent process의 출력이다. 위의 두 process가 parent process로 두고 있는 process와 pid와 이름이 동일하며 fork를 두번 실행한 것을 잘 인지하고 있다. 또한 child로 생성된 두 process또한 잘 인식하는 모습이다.

## 고찰

이번 과제를 통해 이전과제를 하며 배웠던 wrapping을 복습할 수 있었고, 추가적으로 system call 이 어떻게 동작하는지 다시금 확인할 수 있었습니다. 또한 해당 과제들을 진행하며 process 작업시 exit의 출력 한계 등을 보며 프로그래머가 process를 다룰 시 해당 함수들이 어떻게 동작 되는지 알아야함을 다시하면 상기할 수 있었습니다. 그 외에도 CPU scheduling에 따라 달라지는 결과를 통해 스케줄링이 작업 속도에 있어 어떠한 영향을 미치는 지 알 수 있었으며, 마지막 3-3을 통해 process 상태를 파악하기 위해 어떤식으로 코드를 짜야하는 지 알 수 있었습니다.

## Reference

강의자료 참고

22-2\_OSLab\_04\_Systemcall.pdf

22-2\_OSLab\_05\_Module\_Programming\_\_Wrapping.pdf

22-2\_OSLab\_06\_Task\_Management.pdf

22-2\_OSLab\_07\_Thread.pdf

22-2\_OSLab\_09\_CPU\_Scheduling.pdf

2022-2\_OSLab\_Assignment\_3.pdf

외부자료 참고

<https://hackereyes.tistory.com/entry/%EB%B0%94%EB%9E%8C%EC%9D%B4-linux-26123-%EC%A4%91%EC%97%90-schedh%EC%95%88%EC%97%90-taskstruct-%EB%B6%84%EC%84%9D>

<https://aidencom.tistory.com/272>

[https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=alice\\_k106&logNo=221170316769](https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=alice_k106&logNo=221170316769)