

운영체제 Assignment 3

이름 : 이준휘

학번 : 2018202046

교수 : 최상호 교수님

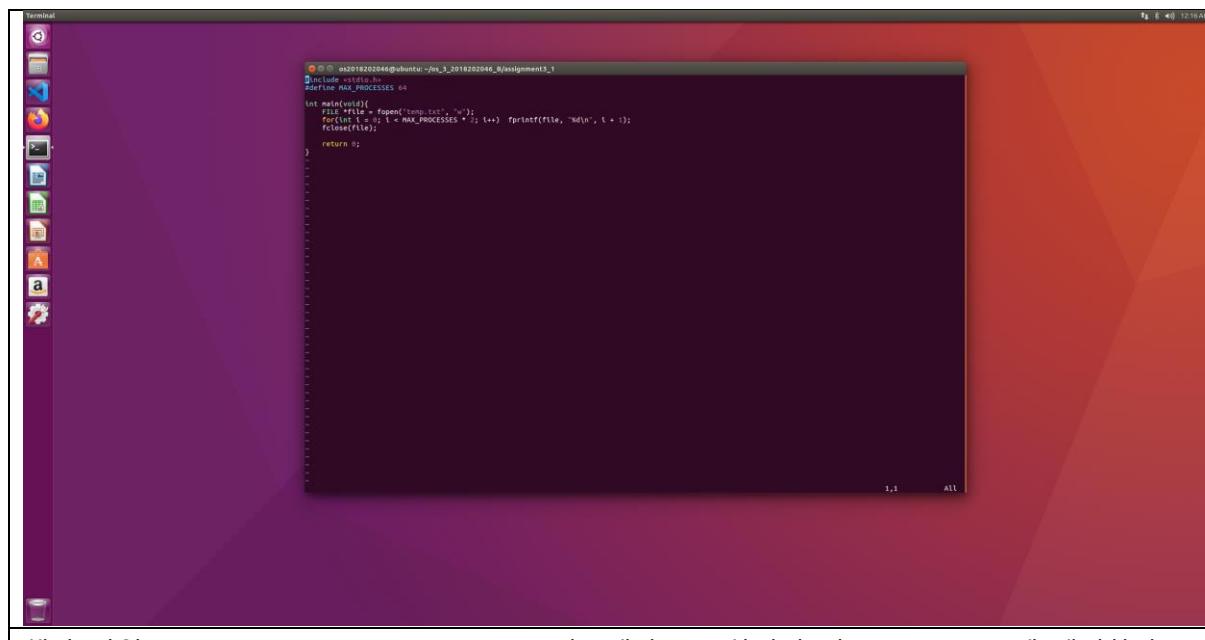
강의 시간 : 금 1, 2

1. Introduction

해당 과제는 3 가지의 소과제로 나누어 진행된다. 첫 번째 과제에서는 자식 프로세스 또는 Thread에서 파일을 읽어 더한 값을 부모에게 전달하는 작업을 수행한다. 해당 과정은 fork와 thread를 통해 각각 구현하며 결과값(value)이 MAX_PROCESSES가 8일 때와 64일 때 어떻게 달라지는지 확인한다. 두 번째 과제에서는 SCHED_RR, SCHED_FIFO, SCHED_OTHER 스케줄러를 통해 fork를 사용하는 프로그램을 돌리고, 경과 시간을 priority나 nice 값에 따라 비교한다. 마지막 과제는 pid에 해당하는 task_struct 내용과 fork 수를 추적하는 모듈을 만든다. 이 때 fork 횟수를 추적하기 위해 task_struct와 fork와 관련된 system call을 변형시키며, task_struct의 구조를 파악하여 이를 출력한다.

2. Conclusion & Analysis

A. Assignment 3-1



```
os2018202046@os2018202046:~/hs_3_2018202046_0/assignment3_1
```

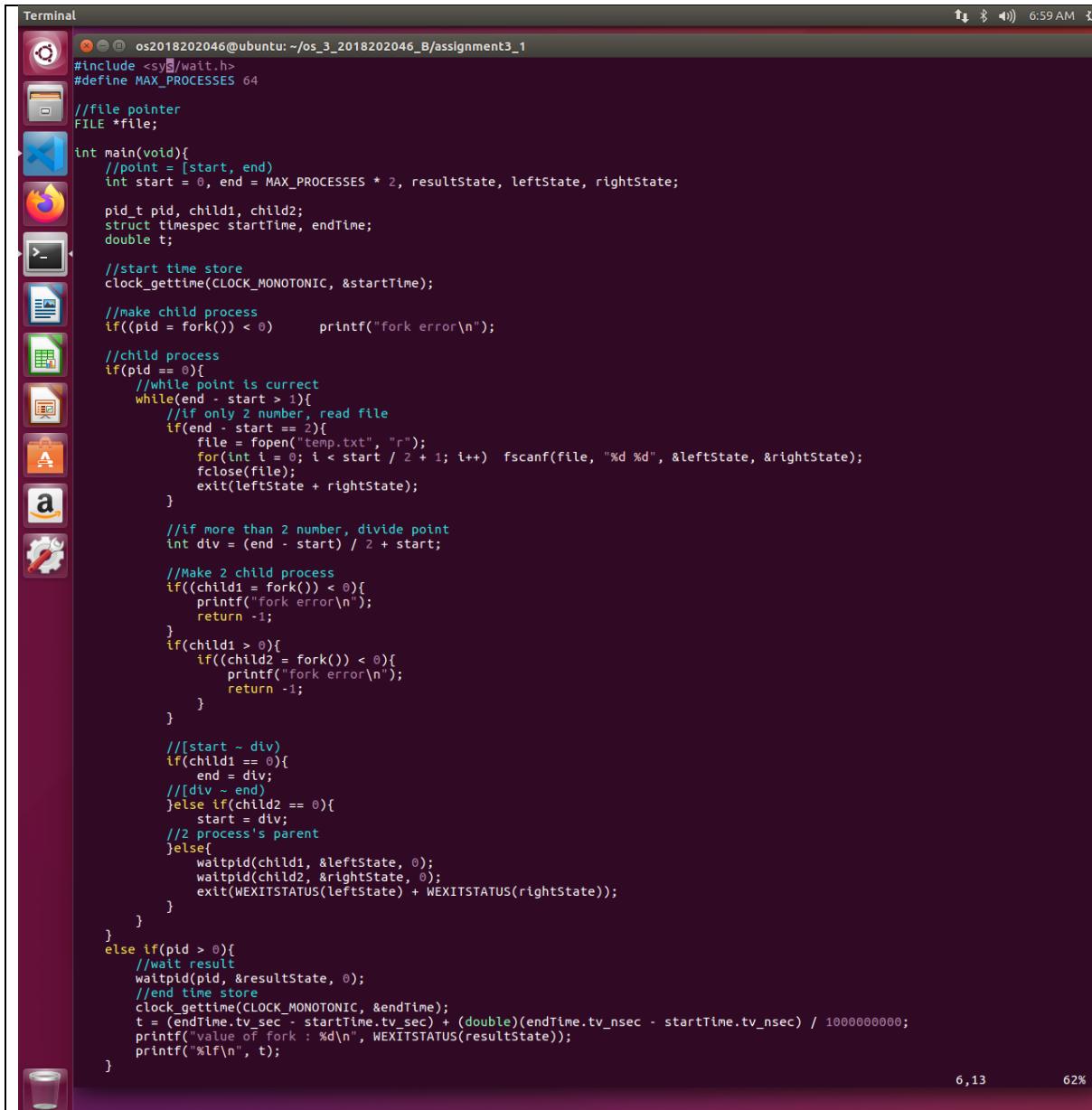
```
#include <stdio.h>
#define MAX_PROCESSES 64

int main()
{
    FILE *file = fopen("temp.txt", "w");
    for(int i = 1; i < MAX_PROCESSES * 2; i++) fprintf(file, "%d\n", i + i);
    fclose(file);

    return 0;
}
```

1,1 All

해당 파일은 numgen.c로 MAX_PROCESSES의 2배만큼 순차적인 변수를 temp.txt에 생성한다.



```

Terminal
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1
#include <sys/wait.h>
#define MAX_PROCESSES 64
FILE *file;
int main(void){
    //point = [start, end]
    int start = 0, end = MAX_PROCESSES * 2, resultState, leftState, rightState;
    pid_t pid, child1, child2;
    struct timespec startTime, endTime;
    double t;
    //start time store
    clock_gettime(CLOCK_MONOTONIC, &startTime);
    //make child process
    if((pid = fork()) < 0)      printf("fork error\n");
    //child process
    if(pid == 0){
        //while point is current
        while(end - start > 1){
            //if only 2 number, read file
            if(end - start == 2){
                file = fopen("temp.txt", "r");
                for(int i = 0; i < start / 2 + 1; i++) fscanf(file, "%d %d", &leftState, &rightState);
                fclose(file);
                exit(leftState + rightState);
            }
            //if more than 2 number, divide point
            int div = (end - start) / 2 + start;
            //Make 2 child process
            if((child1 = fork()) < 0){
                printf("fork error\n");
                return -1;
            }
            if(child1 > 0){
                if((child2 = fork()) < 0){
                    printf("fork error\n");
                    return -1;
                }
            }
            //[[start ~ div]
            if(child1 == 0){
                end = div;
                //[[div ~ end]
                else if(child2 == 0){
                    start = div;
                    //2 process's parent
                }else{
                    waitpid(child1, &leftState, 0);
                    waitpid(child2, &rightState, 0);
                    exit(WEXITSTATUS(leftState) + WEXITSTATUS(rightState));
                }
            }
            else if(pid > 0){
                //wait result
                waitpid(pid, &resultState, 0);
                //end time store
                clock_gettime(CLOCK_MONOTONIC, &endTime);
                t = (endTime.tv_sec - startTime.tv_sec) + (double)(endTime.tv_nsec - startTime.tv_nsec) / 1000000000;
                printf("value of fork : %d\n", WEXITSTATUS(resultState));
                printf("%f\n", t);
            }
        }
    }
}

```

해당 파일은 fork.c로 MAX_PROCESS만큼 프로세스를 생성하여 병렬 덧셈 연산을 수행한다.

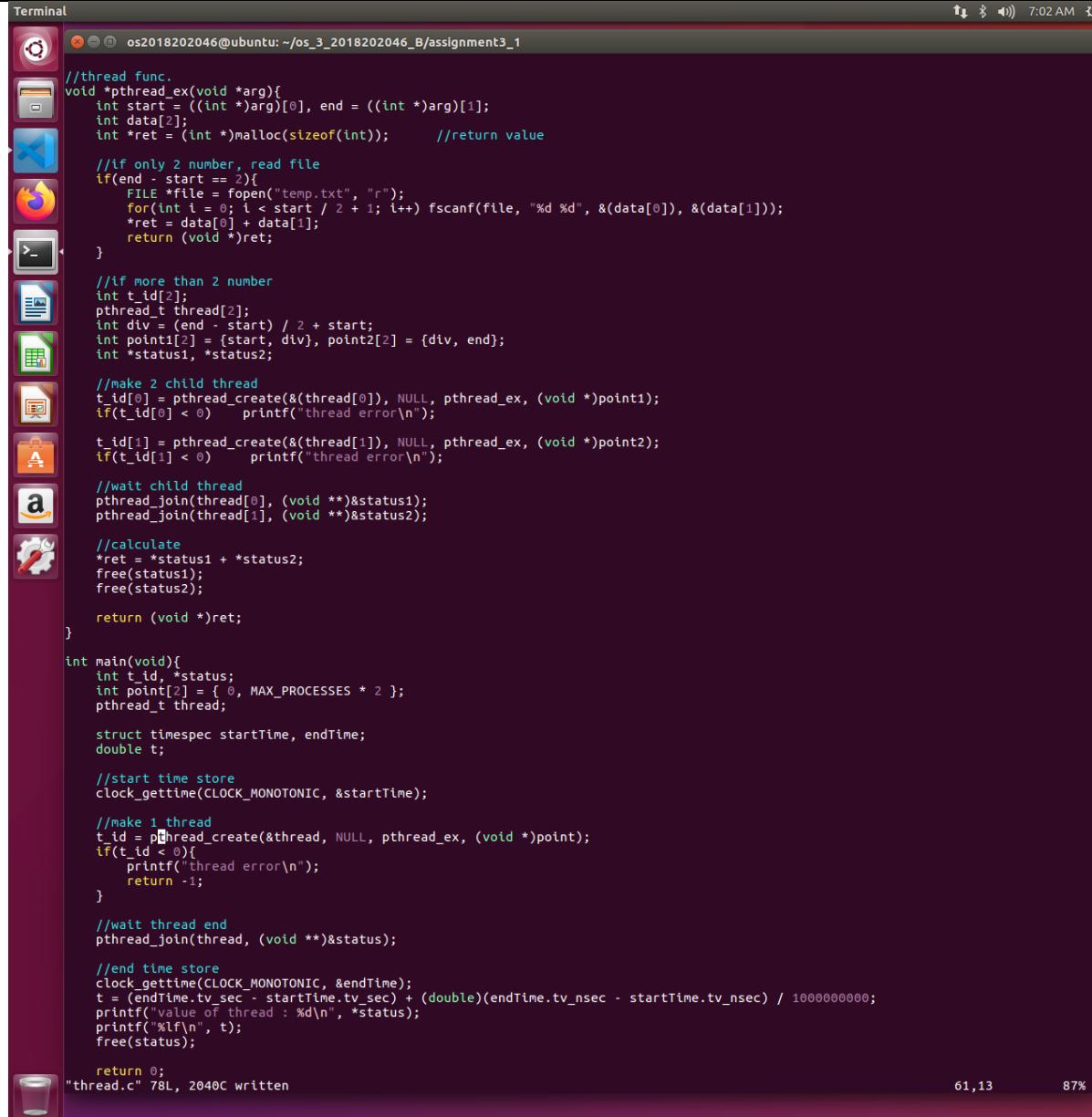
우선 start와 end는 숫자의 구간을 의미하며 0 ~ MAX_PROCESSES * 2로 설정한다. 이후 clock_gettime()함수를 이용하여 timespec 구조체 startTime의 시작 시간을 기록한다. 다음으로 fork()를 통해 자식 프로세스를 만든 후 자식 프로세스에서는 다음 반복문을 수행한다.

반복문에서는 end - start > 1 이상, 즉 구간 안에 2개의 수 이상이 있을 때만 반복을 수행한다. if문을 통해 만약 구간 내에 2개의 수가 있을 경우 temp.txt에서 해당 순서의 수 2개를 읽어온다. 그 후 해당 프로세스는 두 수를 더한 값을 exit의 인자로 사용하여 종료한다.

만약 구간 안에 수가 2개 초과일 경우에는 div를 통해 구간을 나눌 곳을 지정한 후 2번의 fork를 통해 2개의 자식을 생성한다. 이후 하나의 자식은 start ~ div까지, 다른 자식은 div ~ end까지 구간을 다시 반복을 통해 수행하며 현 프로세스는 두 pid를 기다린다. 이 때 반환되는 값은 leftState와 rightState에 저장된다.

Waitpid()로 받은 **leftState**과 **rightState** 변수의 경우 **exit()**의 인자(상위 8 bits) 뿐만 아니라 종료된 상태(하위 8 bits) 또한 포함하기 때문에 해당 변수를 그대로 사용하면 안 된다. 따라서 **WEXITSTATUS()** 또는 **>> 8**을 통해 해당 값을 추출해야 한다.

가장 부모 프로세서는 **waitpid**를 통해 결과를 반환 받는다. 이후 **clock_gettime()**을 통해 종료 시간을 측정하고 걸린 시간 t를 계산한다. 그 후 해당 결과들을 출력하고 프로그램을 종료한다.



```

Terminal
os2018202046@ubuntu: ~/os_3_2018202046_B/assignment3_1

//thread func.
void *pthread_ex(void *arg){
    int start = ((int *)arg)[0], end = ((int *)arg)[1];
    int data[2];
    int *ret = (int *)malloc(sizeof(int)); //return value

    //if only 2 number, read file
    if(end - start == 2){
        FILE *file = fopen("temp.txt", "r");
        for(int i = 0; i < start / 2 + 1; i++) fscanf(file, "%d %d", &(data[0]), &(data[1]));
        *ret = data[0] + data[1];
        return (void *)ret;
    }

    //if more than 2 number
    int t_id[2];
    pthread_t thread[2];
    int div = (end - start) / 2 + start;
    int point1[2] = {start, div}, point2[2] = {div, end};
    int *status1, *status2;

    //make 2 child thread
    t_id[0] = pthread_create(&(thread[0]), NULL, pthread_ex, (void *)point1);
    if(t_id[0] < 0) printf("thread error\n");

    t_id[1] = pthread_create(&(thread[1]), NULL, pthread_ex, (void *)point2);
    if(t_id[1] < 0) printf("thread error\n");

    //wait child thread
    pthread_join(thread[0], (void **)&status1);
    pthread_join(thread[1], (void **)&status2);

    //calculate
    *ret = *status1 + *status2;
    free(status1);
    free(status2);

    return (void *)ret;
}

int main(void){
    int t_id, *status;
    int point[2] = {0, MAX_PROCESSES * 2 };
    pthread_t thread;

    struct timespec startTime, endTime;
    double t;

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

    //make 1 thread
    t_id = pthread_create(&thread, NULL, pthread_ex, (void *)point);
    if(t_id < 0){
        printf("thread error\n");
        return -1;
    }

    //wait thread end
    pthread_join(thread, (void **)&status);

    //end time store
    clock_gettime(CLOCK_MONOTONIC, &endTime);
    t = (endTime.tv_sec - startTime.tv_sec) + (double)(endTime.tv_nsec - startTime.tv_nsec) / 1000000000;
    printf("value of thread : %d\n", *status);
    printf("%lf\n", t);
    free(status);

    return 0;
}
"thread.c" 78L, 2040C written
61,13 87%

```

위의 함수는 **thread**를 이용하여 병렬 덧셈 연산을 수행하는 **thread.c** 파일이다.

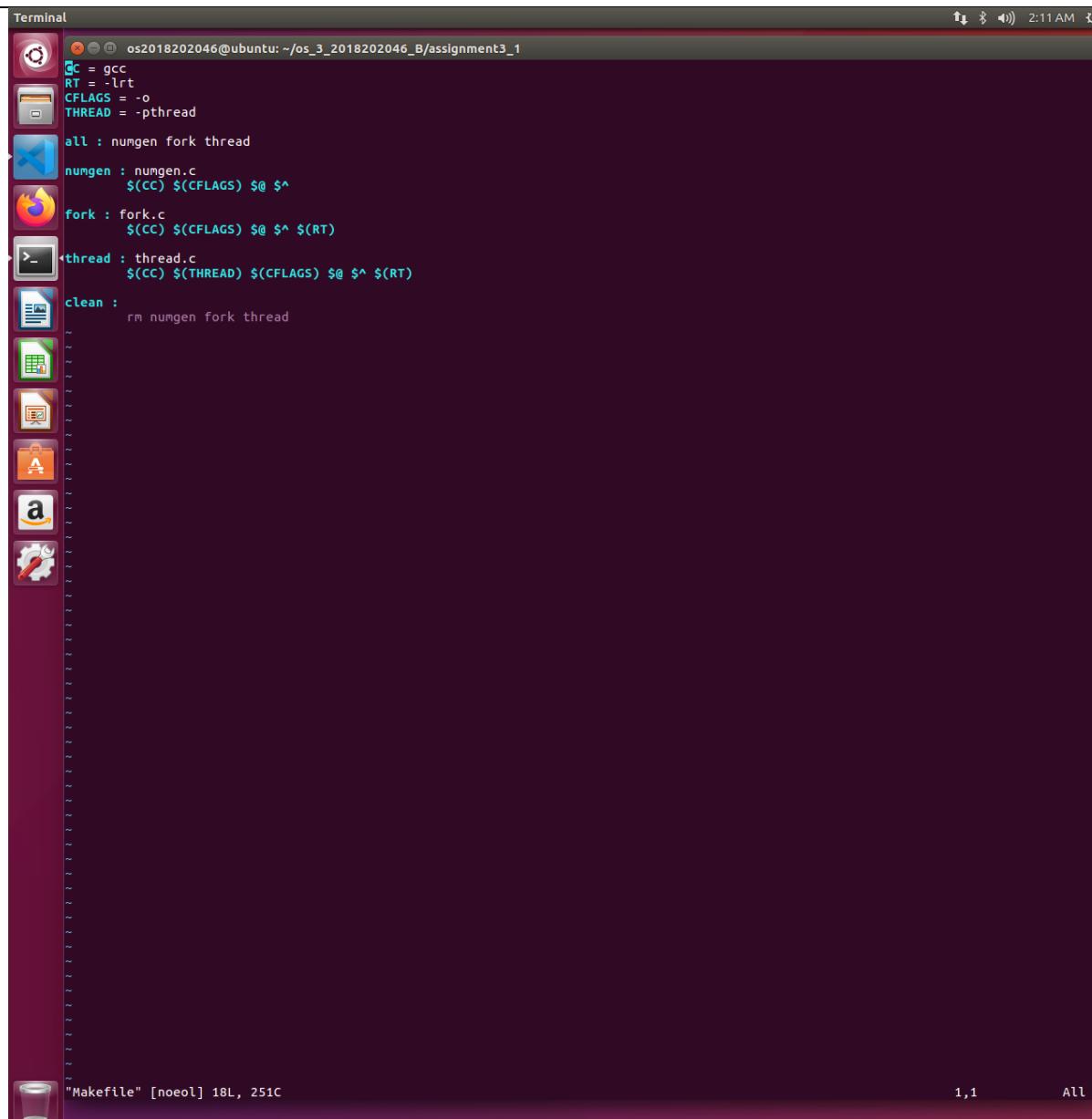
Void *pthread_ex(void *arg) 함수는 **thread**에서 수행할 함수다. 우선 **void *arg**에서 저장된 **start**와 **end**를 추출한다. 그리고 함수 반환 시 전달할 **heap** 변수 **ret**를 미리 생성한다. 만약 현재 구간 안에 2개의 수만 있을 경우 **temp.txt**에서 해당 위치의 값을 읽어 들인다. 그 후 **ret**에 읽은 두 수를 더한 값을 저장하고 해당 **ret**를 반환한다.

만약 두 구간 안에 2개의 수만 있지 않을 경우 구간을 분리하는 작업을 수행한다. 우선 **point1**

과 point2 변수에 구분한 구간을 저장한다. 그 후 pthread_create를 통해 thread 2개를 생성하며 함수는 pthread_ex를 수행하게 하고, point를 arg로 사용한다. 이후 pthread_join 함수를 사용하여 두 thread가 종료되길 기다리고, 반환되는 값을 status가 가리키도록 한다. Status1과 status2를 더한 값을 ret에 저장하고, 사용한 값은 메모리 할당 해제를 진행한다. 그 후 ret를 반환한다.

Main 함수는 다음과 같이 수행된다.

우선 point를 시작과 종료 구간으로 설정한다. 그 후 clock_gettime()을 통해 시작 시간을 측정한다. Pthread_create를 통해 pthread_ex 함수를 point arg를 사용하여 수행하는 thread를 생성하고 pthread_join을 통해 이를 기다린다. 이후 종료 시간을 측정하며 thread의 반환값이 저장된 status와 시간을 출력한다. 마지막으로 status를 메모리 할당 해제 후 프로세스를 종료한다.



The screenshot shows a terminal window on a dark-themed desktop environment. The terminal title is "Terminal" and the command line shows the user's session: "os20182046@ubuntu: ~/os_3_20182046_B/assignment3_1". The terminal content displays a Makefile with the following rules:

```
CC = gcc
RT = -lrt
CFLAGS = -o
THREAD = -pthread

all : numgen fork thread

numgen : numgen.c
        $(CC) $(CFLAGS) $@ $^

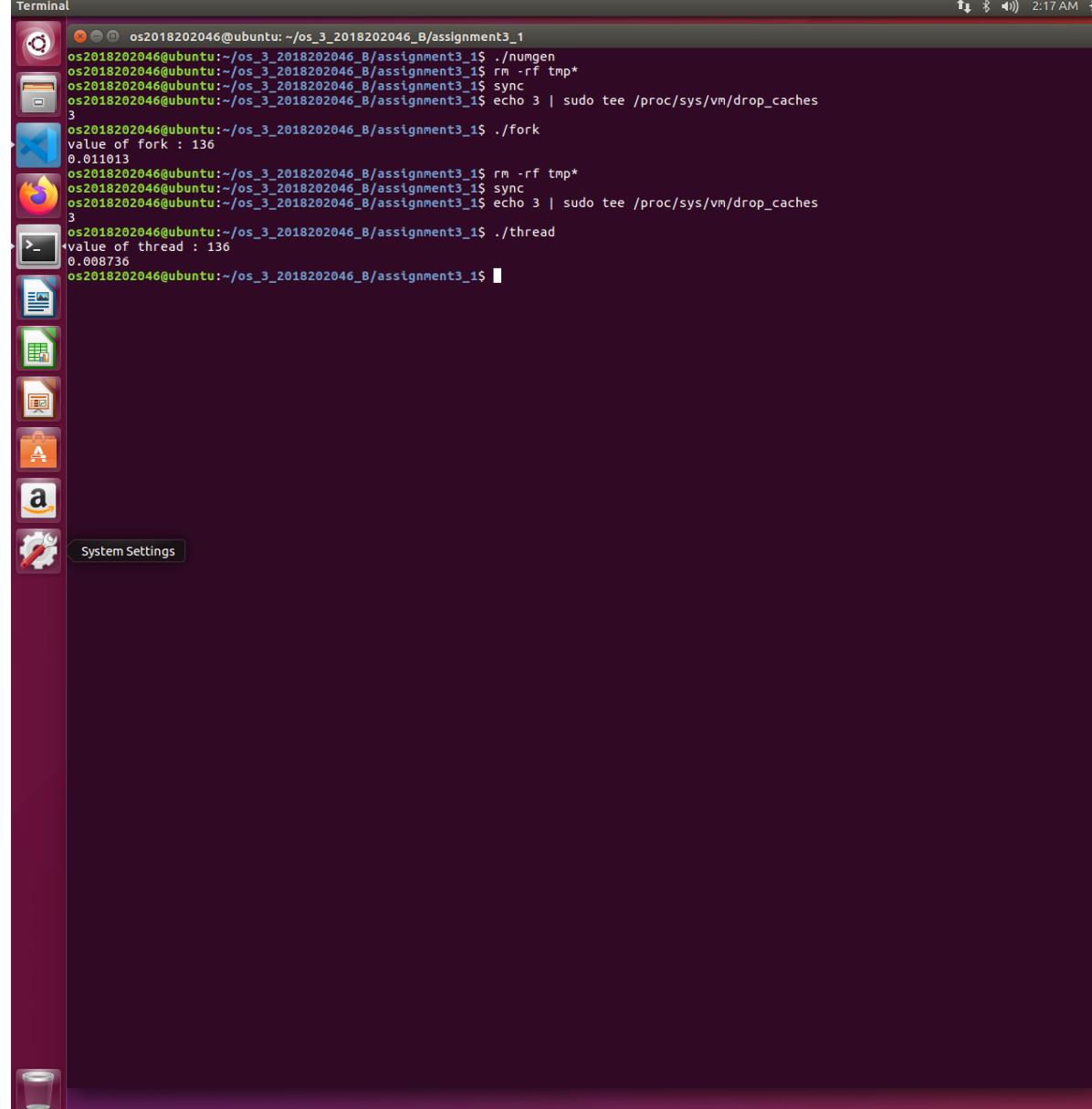
fork : fork.c
        $(CCC) $(CFLAGS) $@ $^ $(RT)

thread : thread.c
        $(CC) $(THREAD) $(CFLAGS) $@ $^ $(RT)

clean :
        rm numgen fork thread
```

The terminal window also shows a vertical dock on the left containing icons for various applications like a file manager, terminal, browser, and system settings. The bottom right corner of the terminal window shows the status bar with "1,1" and "All".

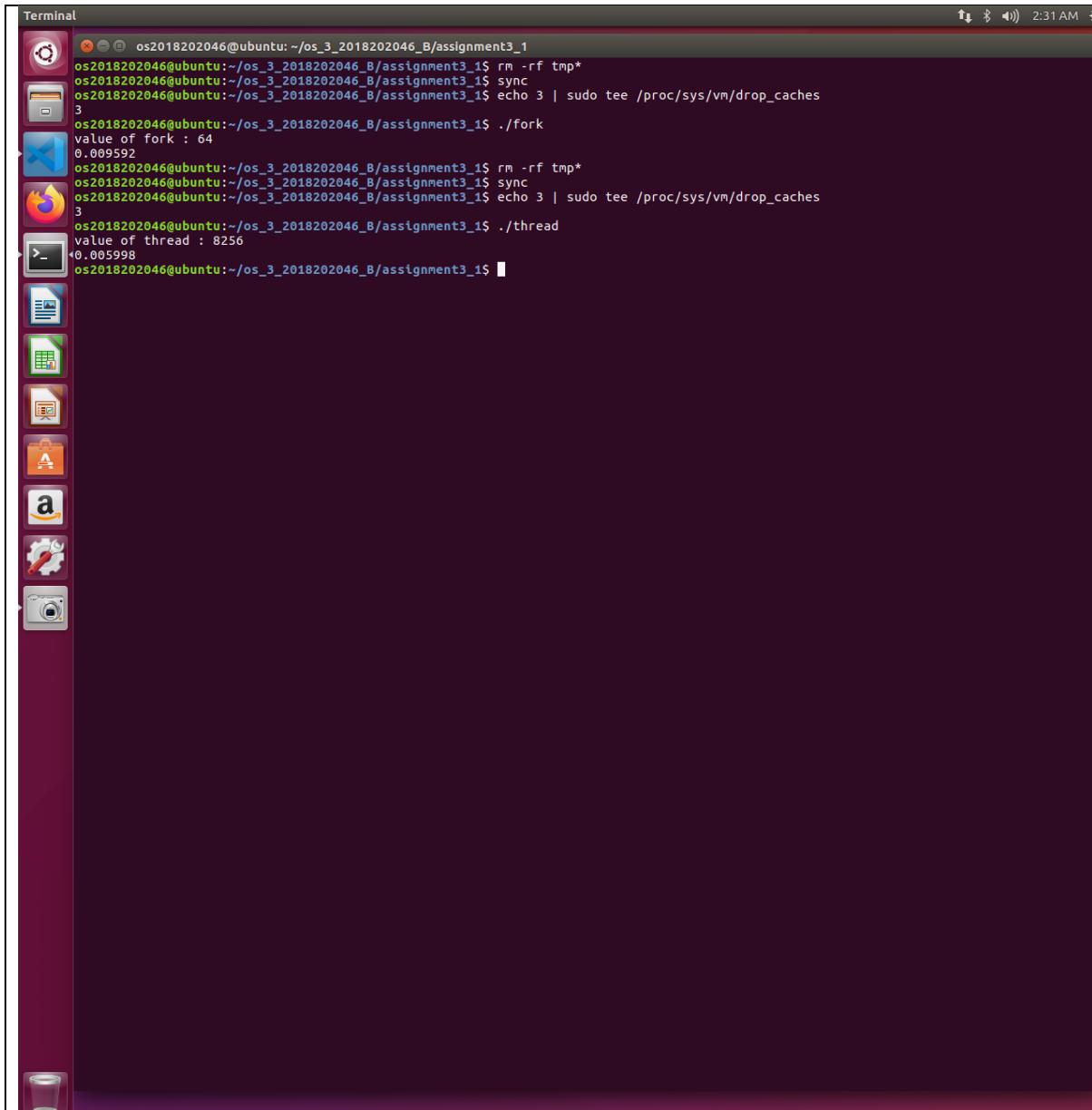
다음 파일은 Makefile 파일이다. 이 때 -lrt 옵션을 libraryrt를 사용하여 clock_gettime을 사용하기 위한 옵션이며, thread target의 경우 -pthread 옵션을 추가해야 컴파일이 가능하다.



The screenshot shows a standard Ubuntu desktop environment. A terminal window is open in the top right corner, displaying a series of command-line operations. The terminal window title is "Terminal". The system tray at the top right shows icons for battery, signal, and volume, with the time "2:17 AM". The desktop background is dark purple. A dock on the left side contains icons for various applications: Dash, Home, Applications, Places, Nautilus, Evolution, Gedit, GIMP, and System Settings. The "System Settings" icon is currently selected.

```
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ ./numgen
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ rm -rf tmp*
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ sync
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ ./fork
value of fork : 136
0.01013
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ rm -rf tmp*
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ sync
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ ./thread
value of thread : 136
0.008736
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$
```

다음 결과는 MAX_PROCESSES를 8로 설정하였을 때의 두 결과를 나타낸다. Fork.c와 thread.c 모두 같은 value값을 나타내며 이는 1 ~ 16까지를 모두 더한 값과 일치하기 때문에 정상적으로 구현됨을 알 수 있다. 두 시간을 비교하였을 때 thread가 fork보다 더 빠르게 수행됨을 확인하였다.

A screenshot of an Ubuntu desktop environment. On the left, there's a vertical dock containing icons for various applications like Dash, Home, Applications, and System Settings. In the center, a terminal window titled 'Terminal' is open, showing a session of a C program. The program performs several operations: it removes temporary files, syncs the system, writes the value '3' to /proc/sys/vm/drop_caches, forks a process, and creates a thread. The output shows the values of fork (64) and thread (8256).

```
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ rm -rf tmp*
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ sync
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ ./fork
value of fork : 64
0.009592
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ rm -rf tmp*
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ sync
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$ ./thread
value of thread : 8256
0.005998
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_1$
```

다음 결과는 MAX_PROCESSES를 64로 하였을 때의 결과를 나타낸 것이다. 위의 결과에서 fork.c의 값은 64, thread.c의 값은 8256으로 차이나 나는 것을 알 수 있다. 이는 exit 시 값의 8bit만 결과를 받을 수 있기 때문에 2^8 의 범위를 초과한 값에서 상위 비트가 누락되어 64로 계산된 것이다. 실제로 8256은 16진수로 2040으로 타나내며 상위 비트가 누락될 경우 40 \rightarrow 64로 결과가 일치한다. 또한 수행 시간이 더욱 짧아지는 것을 볼 수 있다. 하지만 수행 시간의 경우 process가 적기 때문에 스케줄러에 의해 간간히 fork가 더 빠르게 수행되는 경우가 있다. 기본적으로 fork의 경우 새로운 영역을 생성하는 방면 thread의 경우 기존 heap과 code를 그대로 사용할 수 있기 때문에 속도의 차이가 난다.

B. Assignment 3-2

해당 파일은 해당 과제에서 읽을 파일을 생성하는 filegen.c이다. 해당 프로세스는 우선 temp 디렉토리를 mkdir() 함수를 통해 생성한다. 그 후 chdir() 함수로 현재 working directory를 변경 한다. 그 후 MAX_PROCESSES만큼 반복을 수행한다. sprintf() 함수로 현재 반복자로 이름을 설정한 후 fopen()을 통해 파일을 생성한다. 그 후 파일에 1 ~ 9 사이의 난수를 작성한다.

```
Terminal os2018202046@ubuntu: ~/os_3_2018202046_B/assignment3_2
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sched.h>
#define MAX_PROCESSES 10000

FILE *file;

int main(void){
    char path[20];

    int i = 0, buf;

    pid_t pid = getpid(), child = 0;
    pid_t pid_array[MAX_PROCESSES];

    struct timespec startTime, endTime;
    double t;

    //setting priority
    struct sched_param param;

    // int set_priority = sched_get_priority_min(SCHED_FIFO);
    // printf("priority = %d\n", set_priority);
    // param.__sched_priority = set_priority;

    //cd temp
    chdir("temp");

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

    //make MAX_PROCESSES process
    for(; i < MAX_PROCESSES && pid > 0; i++){
        if((pid = fork()) < 0) printf("fork error\n");
        if(pid > 0) {
            // if(sched_setscheduler(pid, SCHED_FIFO, &param) < 0) printf("sched error\n");
            // nice(-20); //nice = [-20, 20]
            pid_array[i] = pid;
        }
    }

    //child process, read each index's file
    if(pid == 0){
        sprintf(path, "%d.txt", i-1);
        file = fopen(path, "r");
        fscanf(file, "%d", &buf);
        fclose(file);
    }

    //parent process, wait i(MAX_PROCESSES) number of process
    } else{
        for(int j = 0; j < i; j++){
            waitpid(pid_array[j], NULL, 0);
        }
    }

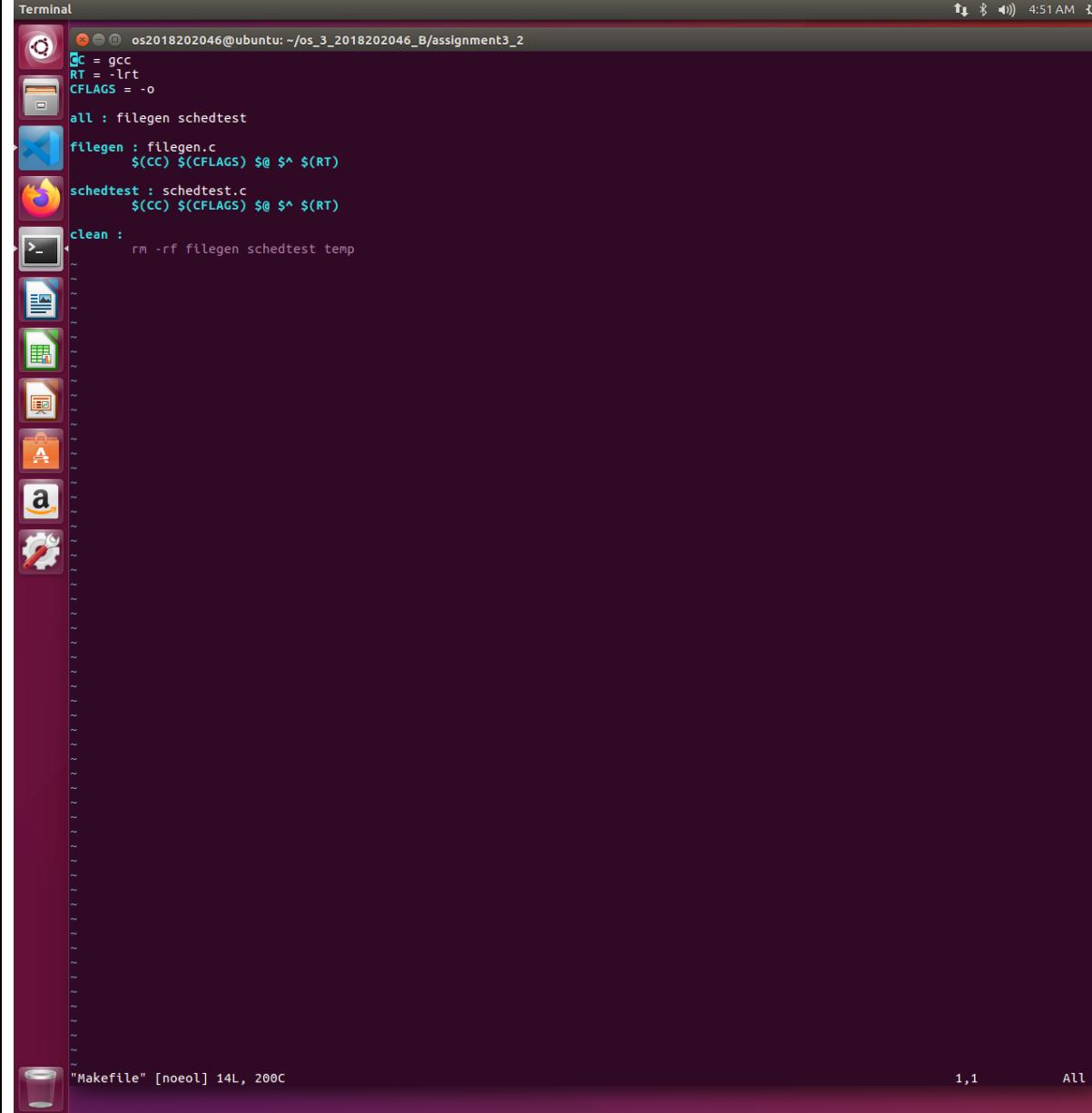
    //end time store
    clock_gettime(CLOCK_MONOTONIC, &endTime);
    t = (endTime.tv_sec - startTime.tv_sec) + (double)(endTime.tv_nsec - startTime.tv_nsec) / 1000000000;
    printf("%lf\n", t);
}

return 0;
}

"schedtest.c" 64L, 1612C
```

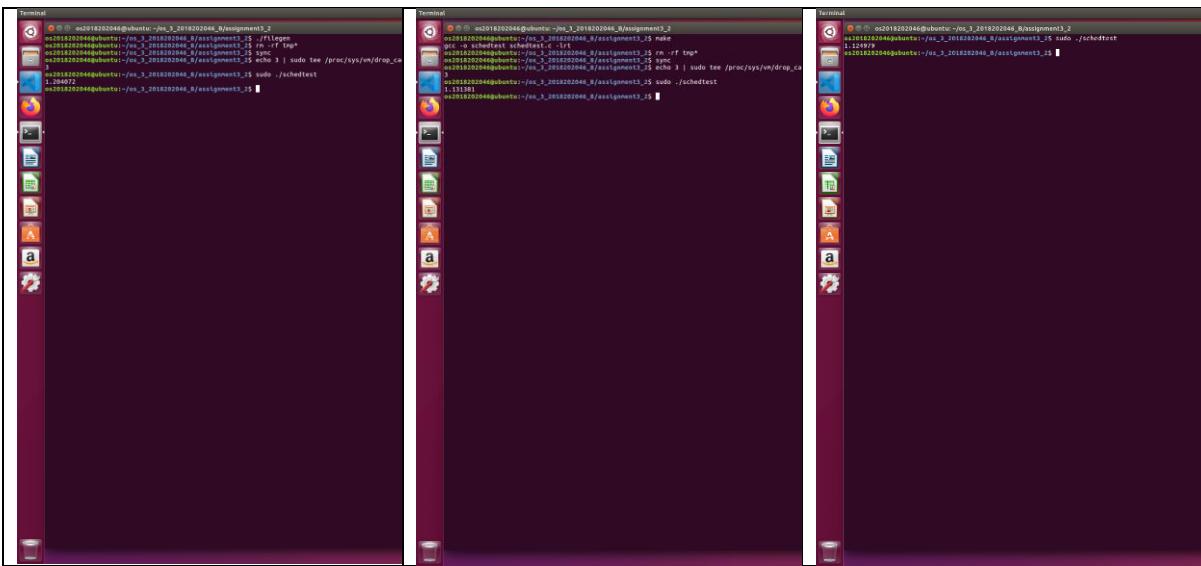
해당 파일은 shcedtest.c로 각 스케줄링 별로 성능을 테스트한다.

Struct `sched_param` `param`은 `priority`를 저장하는 변수다. `Priority`를 각 성능 비교에 맞게 값을 입력한다. 이후 `temp directory`로 이동한다. `Clock_gettime()` 함수를 통해 시작 시간을 저장한다. 그 후 `for`문을 통해 `MAX_PROCESSES`만큼 프로세스를 생성하며, 해당 생성 프로세서들의 `scheduler`를 조정한다. 만약 `SCHED_OTHER`의 경우 `nice`값을 변경하여 비교를 수행한다. 자식 프로세서들은 `sprint()`를 통해 설정된 각 인덱스의 파일을 읽는다. 부모 프로세스는 자식 프로세서의 수만큼 `waitpid()`를 수행한 후 종료 시간을 측정한다. 그 후 시간을 출력하여 이를 비교한다.

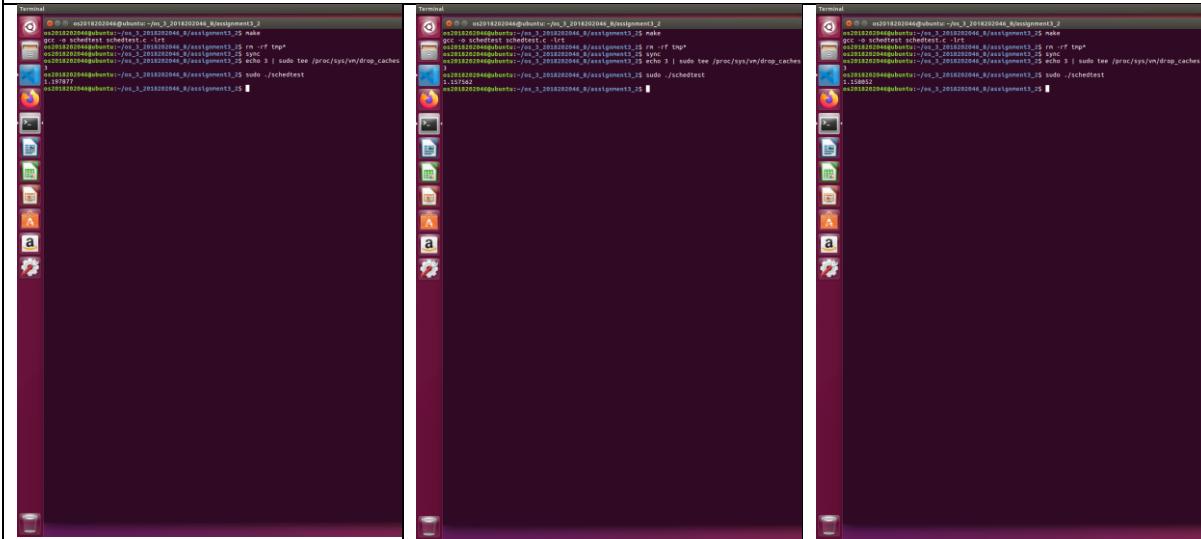
A screenshot of an Ubuntu desktop environment. A terminal window titled "Terminal" is open in the foreground, showing the contents of a Makefile. The terminal window has a dark purple background and a light purple header bar. The desktop background is also a dark purple color. On the left side of the screen, there is a vertical dock containing icons for various applications, including a file manager, a terminal, a browser, and others. The taskbar at the bottom shows the title "Makefile" and some status information like "noeol" and "14L, 200C".

```
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_2
CC = gcc
RT = -lrt
CFLAGS = -o
all : filegen schedtest
filegen : filegen.c
        $(CC) $(CFLAGS) $@ $^ $(RT)
schedtest : schedtest.c
        $(CC) $(CFLAGS) $@ $^ $(RT)
clean :
        rm -rf filegen schedtest temp
```

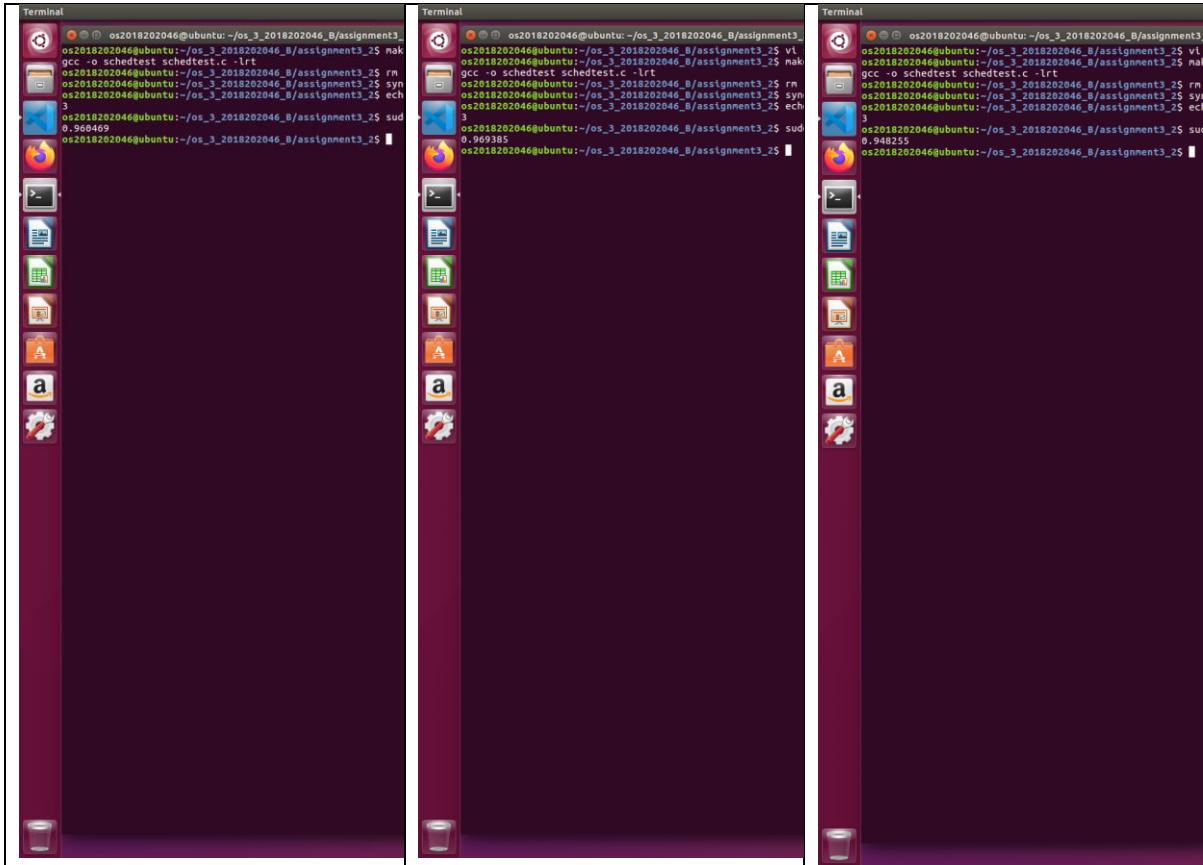
해당 파일은 Makefile로 filegen.c와 schedtest.c 파일을 컴파일한다. 해당 컴파일에서 Libraryrt 를 사용해야 clock_gettime() 함수를 정상적으로 사용할 수 있기 때문에 -lrt를 추가한다. 프로그램을 수행할 때의 유의사항은 sudo로 수행해야 priority나 nice 등을 변경하기에 용이하다는 점이다.



해당 결과는 SCHED_RR(Round Robin)을 사용하여 priority를 각각 min/default/max로 설정하고 실행한 결과다. 다음 결과를 볼 때 min은 1.204초, default는 1.131초, max는 1.124초가 나왔다. Priority가 높아질수록 성능이 개선되는 효과가 나타나는 것을 볼 수 있다. 이는 RR이 Timeslice에 따라 프로그램을 동작시키기 때문에 파일에 접근하는 동안 다른 프로세스가 Timeslice에 의해 수행될 수 있기 때문이다. 하지만 위의 성능 개선폭은 점점 작아지는 것을 볼 수 있는데 이는 priority가 높아질수록 Timeslice의 폭이 작아져 overhead가 커지기 때문이다.

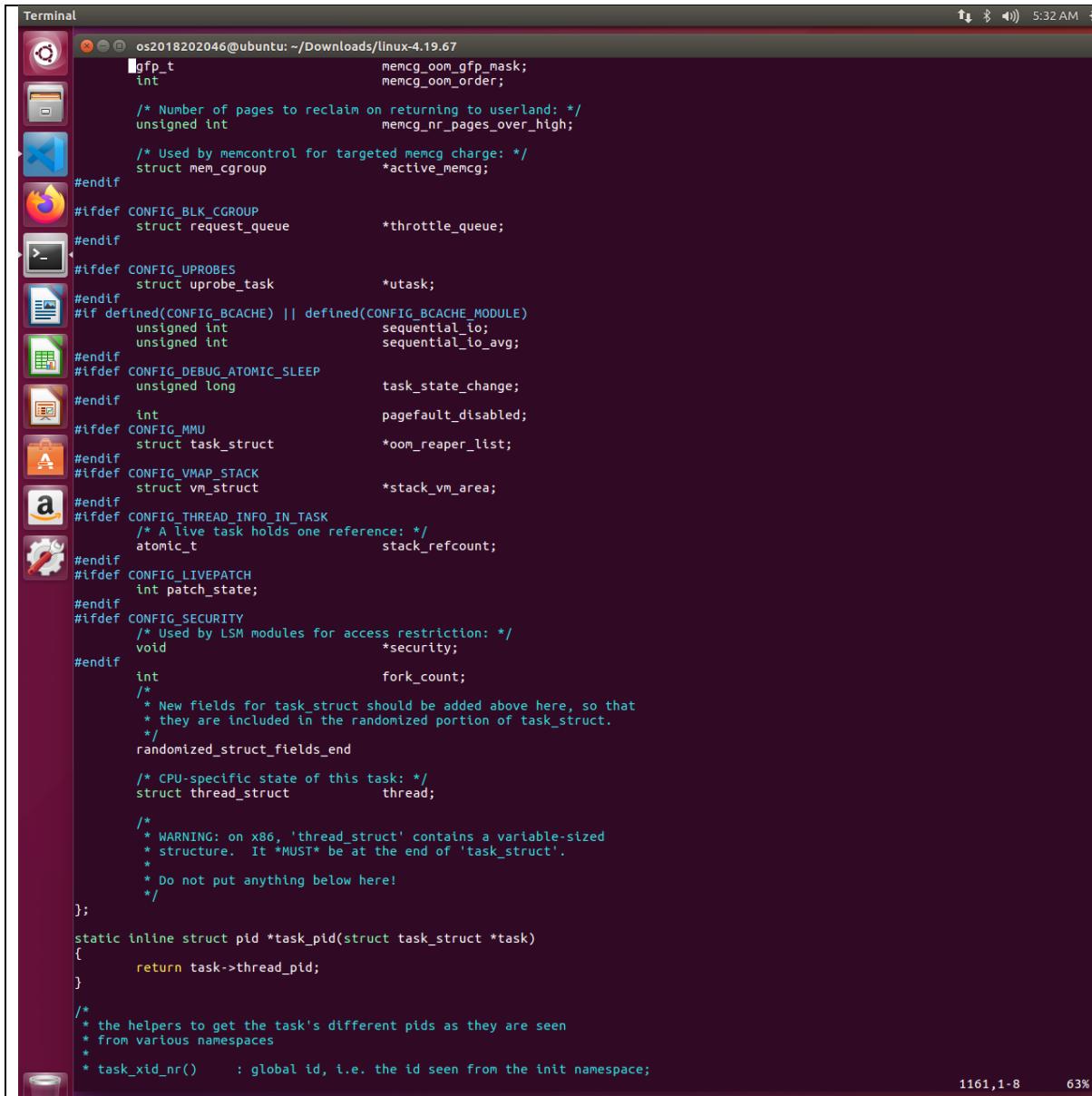


해당 결과는 SCHED_FIFO를 사용하여 priority를 각각 min/default/max로 설정하고 실행한 결과다. 다음 결과를 볼 때 min은 1.197초, default는 1.157초, max는 1.158초가 나왔다. FIFO는 preemptive 방식의 스케줄러이며 timeslice를 사용하지 않는다. Timeslice를 사용하지 않기에 priority가 바뀌더라도 실행 시간이 크게 바뀌지 않는 모습을 볼 수 있다.



해당 결과는 SCHED_OTHER(CFS)를 사용하여 nice 값을 각각 -20/0/19로 설정하고 실행한 결과다. nice값은 -20이 우선 순위가 가장 높으며 +19가 우선순위가 가장 낮다. 또한 0을 기본으로 한다. 해당 스케줄러는 non-preemptive scheduler이기에 만약 파일 읽기와 같은 접근이 발생할 경우 다른 프로세서가 CPU를 사용할 수 있다. 때문에 각 프로세스가 간단한 파일 읽기만 수행하는 해당 nice 값의 변화에 따른 큰 성능차이를 보이지 못한다. 하지만 이전 RR이나 FIFO보다 더욱 효율이 높게 프로그램을 수행시킬 수 있다.

C. Assignment 3-3



A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window. The terminal window title is "Terminal" and the command line shows the user is in the directory ~/Downloads/linux-4.19.67. The terminal displays a portion of the Linux kernel source code, specifically the `task_struct` definition from `include/linux/sched.h`. The code includes various conditional compilation directives (`#ifdef`, `#endif`) and defines for different kernel configurations. The terminal window has a dark theme and is located on a desktop with several icons visible in the background.

```
gfp_t memcg_oom_gfp_mask;
int memcg_oom_order;

/* Number of pages to reclaim on returning to userland: */
unsigned int memcg_nr_pages_over_high;

/* Used by memcontrol for targeted memcg charge: */
struct mem_cgroup *active_memcg;
#endif

#ifndef CONFIG_BLK_CGROUP
    struct request_queue *throttle_queue;
#endif

#ifndef CONFIG_UPROBES
    struct uprobe_task *utask;
#endif

#ifndef CONFIG_BCACHE
    unsigned int sequential_io;
    unsigned int sequential_io_avg;
#endif

#ifndef CONFIG_DEBUG_ATOMIC_SLEEP
    unsigned long task_state_change;
#endif

#ifndef CONFIG_MMU
    struct task_struct *oom_reaper_list;
#endif

#ifndef CONFIG_VMAP_STACK
    struct vm_struct *stack_vm_area;
#endif

#ifndef CONFIG_THREAD_INFO_IN_TASK
    atomic_t stack_refcount;
#endif

#ifndef CONFIG_LIVEPATCH
    int patch_state;
#endif

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

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!
 */
};

static inline struct pid *task_pid(struct task_struct *task)
{
    return task->thread_pid;
}

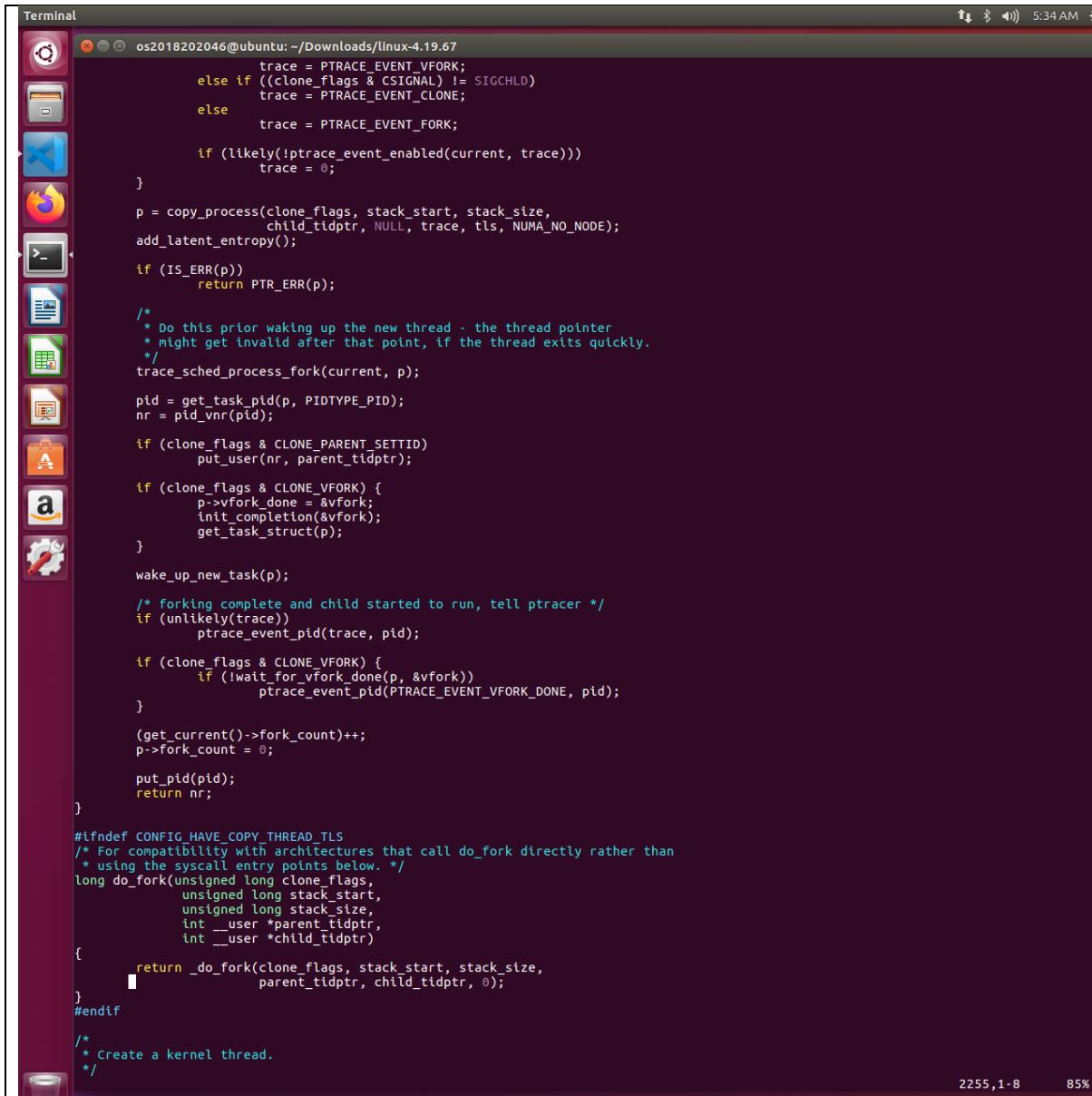
/*
 * the helpers to get the task's different pids as they are seen
 * from various namespaces
 *
 * task_xid_nr() : global id, i.e. the id seen from the init namespace;

```

1161,1-8 63%

해당 파일은 `include/linux/sched.h`의 `task_struct` 구조체의 정의 중 일부를 수정하였다.

`Task_struct`에서 `fork_count` 변수를 추가하여 fork 횟수를 확인할 수 있도록 하였다.



A screenshot of an Ubuntu desktop environment. A terminal window is open in the foreground, displaying a portion of the Linux kernel source code for `kernel/fork.c`. The code is written in C and focuses on the implementation of the `_do_fork()` function. It handles various cloning flags, initializes task structures, and manages fork counts. The terminal window has a dark theme and shows the command prompt as `os2018202046@ubuntu: ~/Downloads/linux-4.19.67`. The background shows the standard Unity interface with icons for various applications like Dash, Home, and System Settings.

```
        trace = PTTRACE_EVENT_VFORK;
    else if ((clone_flags & CSIGNAL) != SIGCHLD)
        trace = PTTRACE_EVENT_CLONE;
    else
        trace = PTTRACE_EVENT_FORK;

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

p = copy_process(clone_flags, stack_start, stack_size,
                 child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
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);

if (clone_flags & CLONE_PARENT_SETTID)
    put_user(nr, parent_tidptr);

if (clone_flags & CLONE_VFORK) {
    p->vfork_done = &vfork;
    init_completion(&vfork);
    get_task_struct(p);
}

wake_up_new_task(p);

/* Forking complete and child started to run, tell ptracer */
if (unlikely(trace))
    ptrace_event_pid(trace, pid);

if (clone_flags & CLONE_VFORK) {
    if (!wait_for_vfork_done(p, &vfork))
        ptrace_event_pid(PTTRACE_EVENT_VFORK_DONE, pid);
}

(get_current()->fork_count)++;
p->fork_count = 0;

put_pid(pid);
return nr;
}

#ifndef CONFIG_HAVE_COPY_THREAD_TLS
/* For compatibility with architectures that call do_fork directly rather than
 * using the syscall entry points below. */
long do_fork(unsigned long clone_flags,
            unsigned long stack_start,
            unsigned long stack_size,
            int __user *parent_tidptr,
            int __user *child_tidptr)
{
    return __do_fork(clone_flags, stack_start, stack_size,
                    parent_tidptr, child_tidptr, 0);
}
#endif

/*
 * Create a kernel thread.
 */
```

해당 파일은 `kernel/fork.c` 파일에서 `_do_fork()` 함수의 일부다. 해당 함수는 `fork()` 실행 시 수행되는 함수로 추가한 내용으로는 `get_current()` 매크로를 통해 현재 `task_struct`를 가져와 `fork_count`를 증가시킨다. 이후 새롭게 생성된 프로세스에 해당하는 `p`의 `fork_count`를 0으로 초기화시켜준다.

```

#include <sys/types.h>
#include <sys/param.h>
#include <sys/conf.h>
#include <sys/conf.h>
#include <sys/proc.h>
#include <sys/proc.h>
#include <sys/r牢.h>
#include <sys/r牢.h>
#define _N# ftrace 336
asmImage int (*real_ftrace)(const struct pt_regs *regs);
void **syscall_table; //System Call Table
static asmlinkage int process_tracer(const struct pt_regs *regs)
{
    pid_t pid = (pid_t)(regs->d1); //Get pid from real_ftrace
    int count = 0;
    //Find task_struct using pid
    struct task_struct *task = pid_task(find_vpid(pid), PIDTYPE_PID);
    struct task_struct *task;
    if(task == NULL) return -1;
    //Print task_struct's info
    printk(KERN_INFO "Process tracer task information of [%d] %s anonymous, findtask->pid, findtask->comm");
    //Print task state
    if(task->state == TASK_RUNNING) printk(KERN_INFO "task state : Running or ready\n");
    else if(task->state == TASK_INTERRUPTIBLE) printk(KERN_INFO "task state : Wait\n");
    else if(task->state == TASK_UNINTERRUPTIBLE) printk(KERN_INFO "task state : Ignoring all signals\n");
    else if(task->state == TASK_STOPPED) printk(KERN_INFO "task state : Stoped\n");
    else if(task->state == TASK_ZOMBIE) printk(KERN_INFO "task state : Zombie process\n");
    else printk(KERN_INFO "task state : etc.\n");
    //Print context switch & fork count
    printk(KERN_INFO "Process Group Leader : [%d] %s, findtask->group_leader->pid, findtask->group_leader->comm");
    printk(KERN_INFO "%d. Number of context switches : %lu", findtask->nivcsw);
    printk(KERN_INFO "%d. Number of forks : %lu", findtask->forks);
    printk(KERN_INFO "%d. Its parent process : [%d] %s, findtask->parent->pid, findtask->parent->comm");
    //Sibling process print
    count = 0;
    list_for_each_entry(task, &findtask->children, sibling);
    if(task->pid == findtask->pid) printk(KERN_INFO "\t[%d] %s, task->pid, task->comm");
    count++;
    if(count == 0) printk(KERN_INFO "\tIt has no sibling\n");
    else printk(KERN_INFO "\t%d child process(es)\n", count);
    //Child process print
    count = 0;
    list_for_each_entry(task, &task->children, sibling);
    if(task->pid == findtask->pid) printk(KERN_INFO "\t\t[%d] %s, task->pid, task->comm");
    count++;
    if(count == 0) printk(KERN_INFO "\t\tIt has no child\n");
    else printk(KERN_INFO "\t\t%d child process(es)\n", count);
    printk(KERN_INFO "Process tracer end of information\n");
    return pid;
}
//syscall table get permission(Read Write)
void make_rw(void *addr)
{
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, level);
    if(pte != NULL) pte->pte |= _PAGE_RW;
}
//syscall table get permission(Read Only)
void make_ro(void *addr)
{
    pte_t *pte = lookup_address((u64)addr, level);
    pte->pte = pte->pte & ~_PAGE_RW;
}
//Module initialize Func
static void __init process_tracer_init(void)
{
    sysctl_table = (void *)kallsyms_lookup_name("sys_call_table");
    real_ftrace = sysctl_table->proc;
    sysctl_table->proc = process_tracer;
    make_rw(sysctl_table);
    return;
}
//Module exit Func
static void __exit process_tracer_exit(void)
{
    sysctl_table->proc = real_ftrace;
    make_ro(sysctl_table);
}
module_init(process_tracer_init);
module_exit(process_tracer_exit);
MODULE_LICENSE("GPL");

```

52,17-24 15N 118,22 Bot

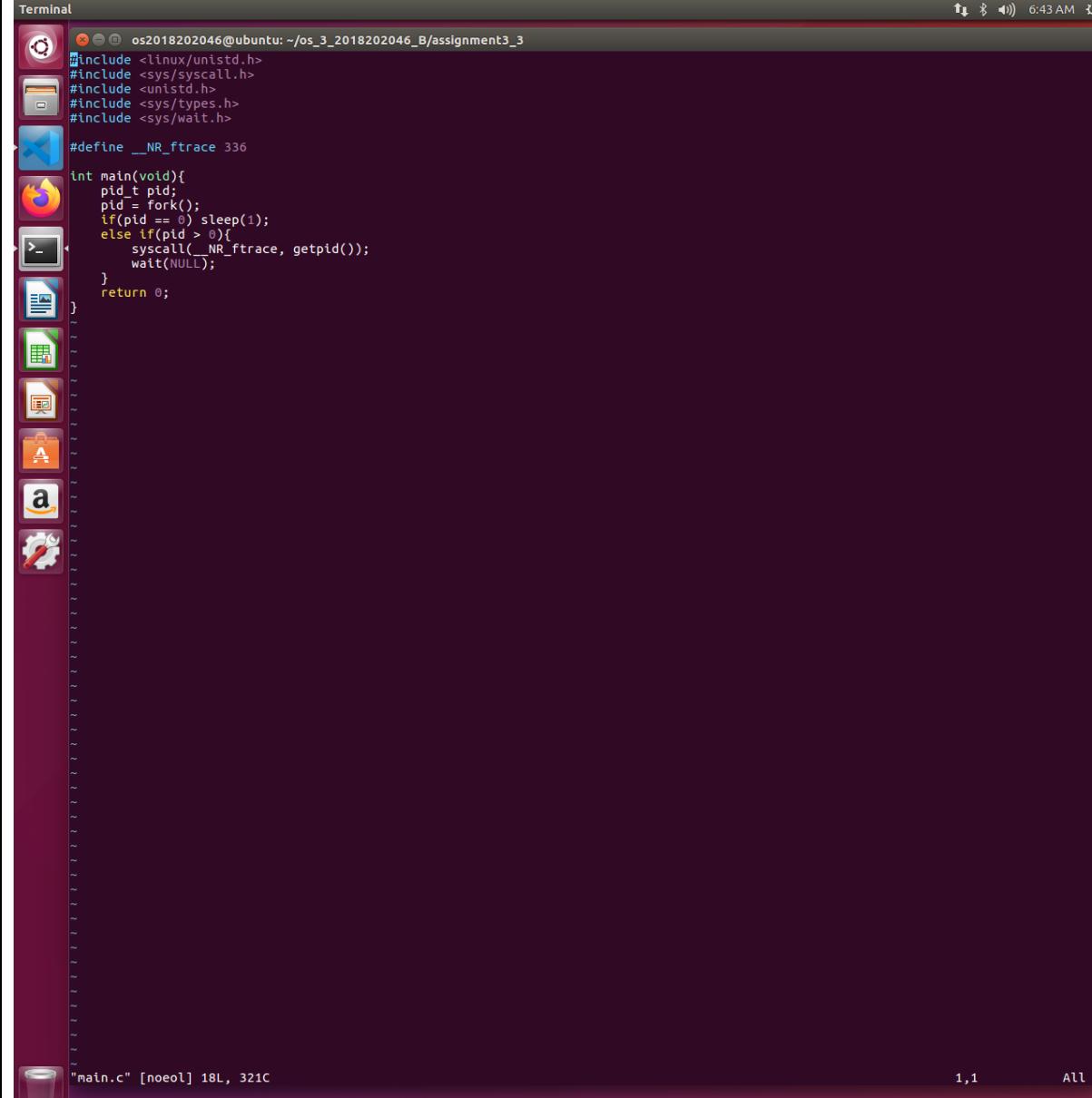
해당 파일은 입력한 pid에 해당하는 task_struct의 정보를 추적할 수 있는 모듈인 process_tracer.c이다.

해당 모듈에서는 우선 기존 ftrace를 저장할 함수 포인터를 생성한다. 또한 syscall_table을 수정하기 위한 void **를 생성한다. Process_tracer() 함수는 현재 pt_regs의 di로부터 입력한 pid 값을 읽어온다. 그 후 pid_task(find_vpid(), PIDTYPE_PID)함수를 통해 pid에 해당하는 task_struct를 찾는다. 만약 찾는데 실패한 경우 -1을 반환한다. 이후 출력문을 수행한다.

Task_struct의 pid와 이름은 pid, comm에 저장되어 있기에 이를 출력한다. State 변수는 현재 상태를 나타내며 각 상태는 define되어 있기에 일치하는 경우에 맞게 출력한다. Processor Group Leader는 group_leader 포인터를 이용해 해당 task_struct에 접근할 수 있다. Context switching 횟수는 nivcsw에 기록되어 있으며 fork 횟수는 이전의 fork_count에 기록되어 있다. Parent process는 parent 포인터로 접근할 수 있다. Sibling process와 child process를 접근하는 방법은 struct list_head children과 sibling을 이용한다. Sibling은 부모의 children list에 자신을 연결시켜 주는 역할을 수행하며 child은 해당 프로세스의 자식 프로세스들이다. Sibling process를 찾기 위해 list_for_each_entry() 매크로를 사용하여 현재 부모의 자식 프로세스 중 sibling에 해당하는 task를 찾아 현재 task가 아닌 경우에만 출력하도록 한다. Child process의 경우 현 task의 children 중 sibling을 찾는 방법으로 접근한다.

Make_rw와 make_ro는 syscall table을 조작하는 함수로 rw에서는 syscall table을 가져와 읽기 쓰기 권한을 부여한다. Make_ro의 경우 쓰기 권한을 뺏는다. Process_tracer_init()은 Module initialize 함수다. 해당 함수에서는 기존 syscall table에서 336번에 해당하는 함수를 임시로 저장하고 현재 함수로 hijack을 수행한다. Process_tracer_exit()함수에서는 변경했던 함수를 원상복구 한다.

Makefile의 경우 이전과 같이 작성할 수 있다. Obj-m에 모듈 이름을 작성하며 KDIR은 modules 컴파일러의 위치 PWD는 현재 위치를 입력하고 CFLAG로 -DEXPORT_SYMTAB은 global 변수가 없기에 선택사항이다.

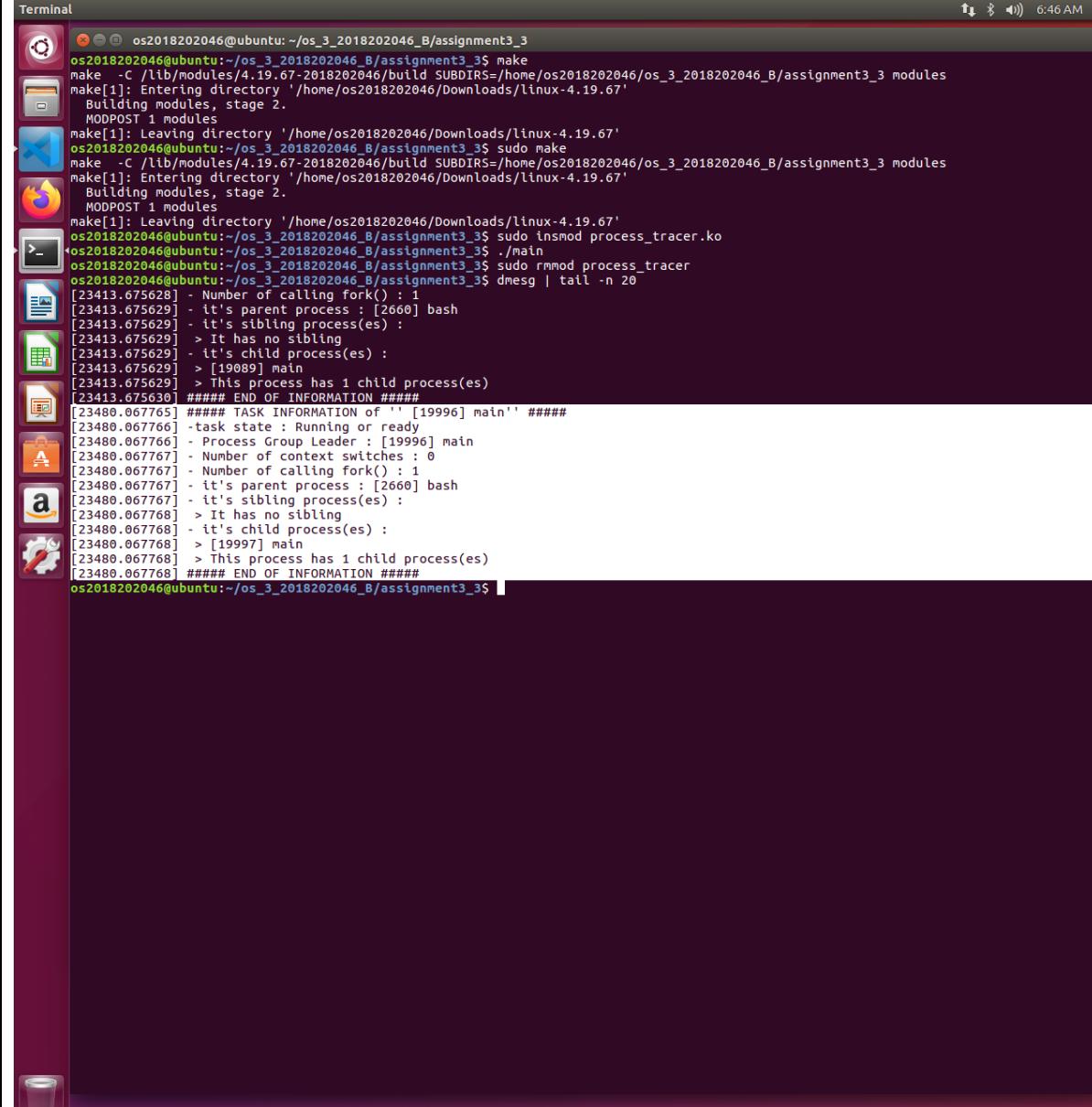
A screenshot of an Ubuntu desktop environment. A terminal window titled "Terminal" is open in the foreground, showing the following C code:

```
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define __NR_ftrace 336

int main(void){
    pid_t pid;
    pid = fork();
    if(pid == 0) sleep(1);
    else if(pid > 0){
        syscall(__NR_ftrace, getpid());
        wait(NULL);
    }
    return 0;
}
```

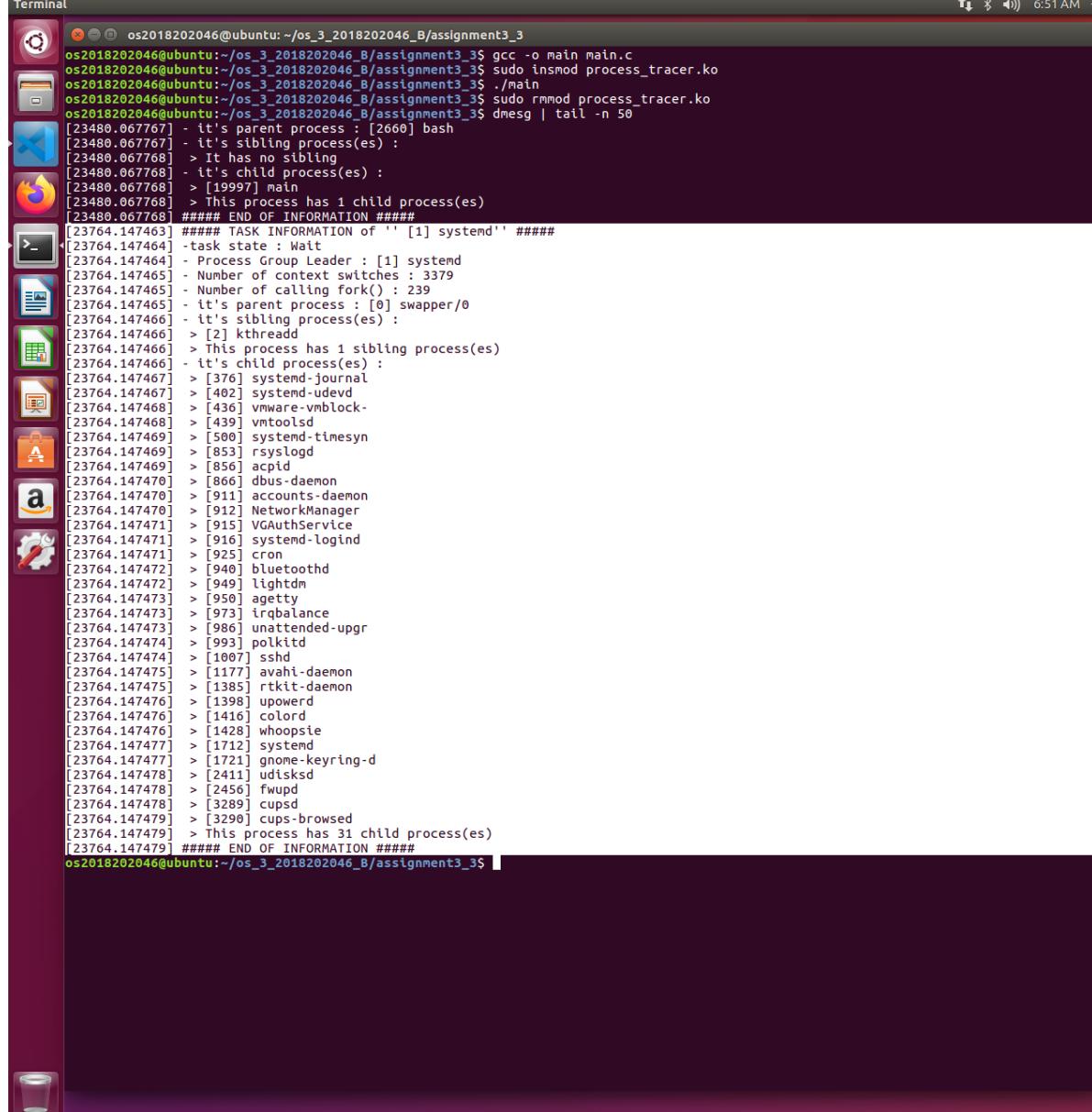
The terminal window also displays the file path "main.c" [noeol] 18L, 321C and the coordinates 1,1 All. The desktop background shows various icons for applications like Dash, Home, and System Settings.

다음 프로그램은 임시로 테스트하기 위한 프로그램이다. Fork()의 수가 정상적으로 세지는지 확인할 수 있다.



```
Terminal
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ make
make[1]: Leaving directory '/home/os2018202046/Downloads/linux-4.19.67'
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ make
make[1]: Entering directory '/home/os2018202046/os_3_2018202046_B/assignment3_3 modules'
Building modules, stage 2.
MODPOST 1 modules
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ sudo make
make -C /lib/modules/4.19.67-2018202046/build SUBDIRS=/home/os2018202046/os_3_2018202046_B/assignment3_3 modules
make[1]: Entering directory '/home/os2018202046/downloads/linux-4.19.67'
Building modules, stage 2.
MODPOST 1 modules
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ sudo insmod process_tracer.ko
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ ./main
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ sudo rmmod process_tracer
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ dmesg | tail -n 20
[23413.675628] - Number of calling fork() : 1
[23413.675629] - it's parent process : [2660] bash
[23413.675629] - it's sibling process(es) :
[23413.675629] > It has no sibling
[23413.675629] - it's child process(es) :
[23413.675629] > [19089] main
[23413.675629] > This process has 1 child process(es)
[23413.675630] ##### END OF INFORMATION #####
[23480.067765] ##### TASK INFORMATION of '' [19996] main' #####
[23480.067766] -task state : Running or ready
[23480.067766] - Process Group Leader : [19996] main
[23480.067767] - Number of context switches : 0
[23480.067767] - Number of calling fork() : 1
[23480.067767] - it's parent process : [2660] bash
[23480.067767] - it's sibling process(es) :
[23480.067768] > It has no sibling
[23480.067768] - it's child process(es) :
[23480.067768] > [19997] main
[23480.067768] > This process has 1 child process(es)
[23480.067768] ##### END OF INFORMATION #####
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$
```

다음 결과를 확인하였을 때 위와 같은 출력을 보이고 있다. Pid와 프로그램 명 main이 정상적으로 출력되며 상태 또한 실행 중으로 정상적으로 보인다. Group leader가 main으로 일치하며 context switch가 사용되지 않을 정도로 짧은 프로그램이기에 0으로 출력된다. Fork()는 1번 수행했기에 1로 정상 출력되며 parent process는 bash shell에서 수행했기에 정상적으로 출력되었다. Sibling은 없으며 child 프로세스는 하나 생성되어 있기에 해당 자식이 제대로 출력되었다.



```

Terminal
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ gcc .o main main.c
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ sudo insmod process_tracer.ko
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ sudo rmmod process_tracer.ko
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ dmesg | tail -n 50
[23480.067767] - it's parent process : [2666] bash
[23480.067767] - it's sibling process(es) :
[23480.067768] > It has no sibling
[23480.067768] - it's child process(es) :
[23480.067768] > [19997] main
[23480.067768] > This process has 1 child process(es)
[23480.067768] ##### END OF INFORMATION #####
[23764.147463] ##### TASK INFORMATION of '' [1] systemd' #####
[23764.147464] -task state : Wait
[23764.147464] - Process Group Leader : [1] systemd
[23764.147465] - Number of context switches : 3379
[23764.147465] - Number of calling fork() : 239
[23764.147465] - it's parent process : [0] swapper/0
[23764.147466] - it's sibling process(es) :
[23764.147466] > [2] kthreadd
[23764.147466] > This process has 1 sibling process(es)
[23764.147466] - it's child process(es) :
[23764.147467] > [376] systemd-journal
[23764.147467] > [402] systemd-udevd
[23764.147468] > [436] vmware-vmblock-
[23764.147468] > [439] vmtoolsd
[23764.147469] > [500] systemd-timesyn
[23764.147469] > [853] rsyslogd
[23764.147469] > [856] acpid
[23764.147470] > [866] dbus-daemon
[23764.147470] > [911] accounts-daemon
[23764.147470] > [912] NetworkManager
[23764.147471] > [915] VAuthService
[23764.147471] > [916] systemd-logind
[23764.147471] > [925] cron
[23764.147472] > [940] bluetoothd
[23764.147472] > [949] lightdm
[23764.147473] > [950] getty
[23764.147473] > [973] irqbalance
[23764.147473] > [986] unattended-upgr
[23764.147474] > [993] polkitd
[23764.147474] > [1007] sshd
[23764.147475] > [1177] avahi-daemon
[23764.147475] > [1385] rtkit-daemon
[23764.147476] > [1398] upowerd
[23764.147476] > [1416] colord
[23764.147476] > [1428] whoopsie
[23764.147477] > [1712] systemd
[23764.147477] > [1721] gnome-keyring-d
[23764.147478] > [2411] udisksd
[23764.147478] > [2456] fwupd
[23764.147478] > [3289] cupsd
[23764.147479] > [3290] cups-browsed
[23764.147479] > This process has 31 child process(es)
[23764.147479] ##### END OF INFORMATION #####
os2018202046@ubuntu:~/os_3_2018202046_B/assignment3_3$ 
```

위의 출력은 syscall로 pid에 1을 넣었을 때의 결과다. 이전과 같이 정상 출력되는 모습이 확인되며, state가 wait으로 표시되고, context switch, sibling process가 정상적으로 출력되는 것을 검증할 수 있다. 이를 통해 해당 모듈이 정상 구현되었음을 확인하였다.

3. Consideration

해당 과제를 통해 thread와 fork의 프로그램 작성 방법을 이해하고 이를 이용한 병렬 연산을 직접 만들어보는 시간을 보냈다. 특히 fork를 사용할 시 반환되는 값이 8bit 밖에

되지 않기에 프로그램 구상 시 이를 고려해야 한다는 점을 알았다. 그리고 `get_clocktime()` 함수를 사용하여 시간을 `clock`을 통해 측정하는 방법을 알 수 있었다. 비록 현재는 `chrono`를 이용하면 더욱 쉽게 가능하지만 이전에는 어떻게 시간을 측정하는지 공부할 수 있었다. 2번 소과제에서는 스케줄러와 `priority` 그리고 `nice`값을 바꾸는 함수를 공부할 수 있었으며 각 값에 따라 어떠한 성능 차이가 있는지 직접 확인할 수 있었다. 마지막 과제에서는 `task_struct` 구조체에 어떠한 변수가 저장되어 있는지 직접 확인함으로써 이해할 수 있었던 과제였다.

4. Reference

- 강의자료만을 참고