운영체제 실습 Report

실습 제목: Assignment 3

실습일자: 2022년 10월 07일 (금)

제출일자: 2022년 11월 10일 (목)

학 과: 컴퓨터정보공학부

담당교수: 최상호 교수님

실습분반: 금요일 5,6교시

학 번: 2018202065

성 명: 박 철 준

- Introduction

1. 제목

Assignment 3

2. 과제 요구사항

Assignment 3-1

- numgen (numgen.c) 프로그램을 만들어야한다.

1. 특정 파일("temp.txt")를 생성한다.
2. i번째 값을 integer형 양수으로 생성할 프로세스 수의 2배만큼 (MAX_PROCESSES) 기록한다.

- fork.c, thread.c 프로그램

1. MAX_PROCESSES만큼 프로세스 또는 스레드를 생성한다.
2. 최상단 프로세스/스레드마다 2개의 숫자를 읽는다.
3. 각 프로세스/스레드는 두 개의 숫자를 더한 후, 부모 프로세스/스레드에게 값을 전달한다. (fork → exit() 사용)
4. 최종적으로 나온 값, 전체 프로그램 수행시간 측정한다. (clock_gettime() 사용)

Assignment 3-2

각 프로세스에서 CPU 스케줄링 정책을 변경한다.

- filegen (filegen.c) 프로그램을 만들어야한다.

1. 특정 디렉토리("temp")를 생성한다.
2. 이에 무작위의 integer형 양수(≤9)가 기록되어 있는 파일 (./temp/0, ./temp/1, ./temp/2, ...)을 생성할 프로세스만큼(MAX_PROCESSES) 생성

- schedtest (schedtest.c) 프로그램

1. 매 실험 전에 소스코드에 위치한 디렉토리에서 아래의 명령어를 수행할 것 캐시 및 버퍼를 비워서 실험에 영향을 주는 요소를 제거하기 위함이다.

```
$ rm -rf tmp*
$ sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
```

2. MAX_PROCESSES 만큼 프로세스를 생성한다. (fork() 사용)
3. 각 프로세스에서 CPU 스케줄링 정책을 변경 (sched_setscheduler() 사용)
4. 각 i(단, 0 ≤ i < MAX_PROCESSES) 번째 프로세스에서 미리 만들어져 있는 i 번째 파일(temp/i)에서 integer 데이터 읽는다.
5. 성능을 비교할 수 있을 정도의 프로세스를 생성하여 실험하여야 한다. ((ex.) MAX_PROCESS = 10000)
6. Linux에서 지원하는 3가지 CPU 스케쥴링와 Priority, Nice의 값을 각각 highest, defalut, lowest로 설정한다.
7. 각 CPU 스케쥴링의 Priority, Nice의 highest, default, lowest 설정한 값을 보고서에 작성한다.

Assignment 3-3

PID를 바탕으로 아래와 같은 프로세스의 정보를 출력하는 Module 작성한다.

(1) 프로세스 이름

(2) 현재 프로세스의 상태

Running or ready, Wait with ignoring all signals, Wait, Stopped, Zombie process, Dead, etc.

(3) 프로세스 그룹 정보 - PGID, 프로세스 이름

(4) 해당 프로세스를 실행하기 위해 수행된 context switch 횟수

(5) fork()를 호출한 횟수

(6) 부모(parent) 프로세스 정보 - PID, 프로세스 이름

(7) 형제자매(sibling) 프로세스 정보 - PID, 프로세스 이름, 총 형제자매 프로세스 총 개수

(8) 자식(child) 프로세스 정보 - PID, 프로세스 이름, 총 자식 프로세스 수

2차 과제에서 작성한 ftrace 시스템 콜(336번)을 다음 함수로 wrapping하여 사용

Hooking 함수 명 : process_tracer

ex. asmlinkage pid_t process_tracer(pid_t trace_task);

Return value

정상적으로 정보를 출력한 경우; 입력 받은 PID 값

출력을 완료하지 못한 경우; -1

Input value

pid_t trace_task : 정보를 출력할 프로세스의 ID 값

메시지 출력 형식

1. Kernel ring buffer에 본 자료에 포함된 예시와 같은 형태로 출력한다.
2. 프로세스 이름은 축약하지 않은 형태로 출력한다.

현재 프로세스의 상태 정의

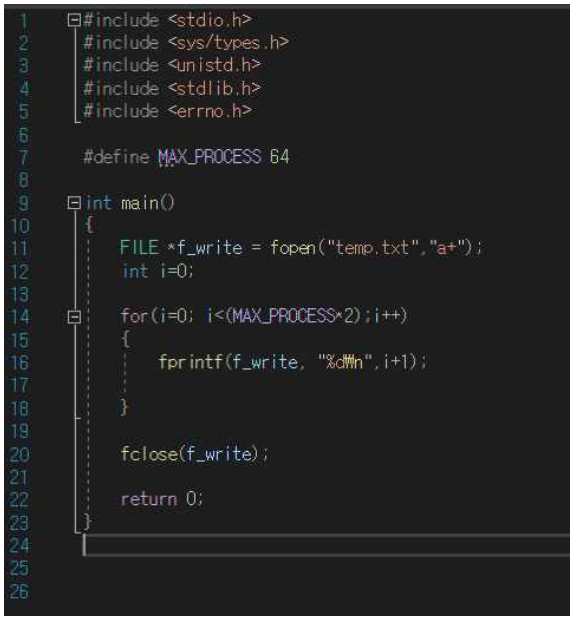
다음과 같은 7가지의 상태로 구분

1. Running or ready
2. Wait with ignoring all signals
3. Wait
4. Stopped
5. Zombie process
6. Dead
7. etc.

- Conclusion & Analysis

Assignment 3-1

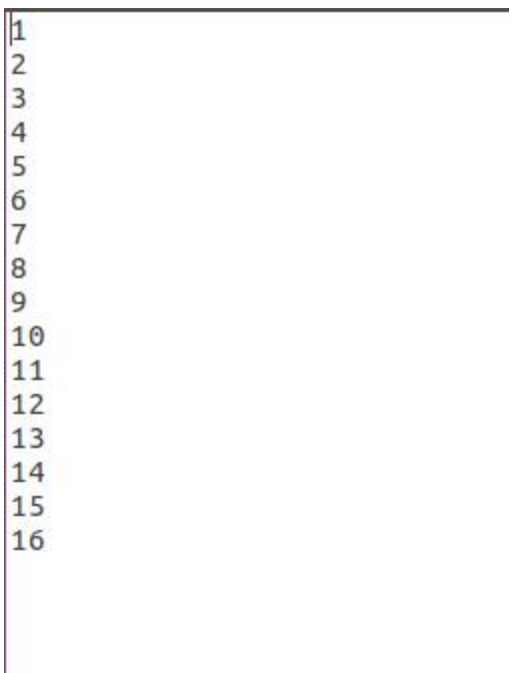
- numgen (numgen.c) 프로그램



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

#define MAX_PROCESS 64

int main()
{
    FILE *f_write = fopen("temp.txt","a+");
    int i=0;

    for(i=0; i<(MAX_PROCESS*2);i++)
    {
        fprintf(f_write, "%d\n",i+1);

    }

    fclose(f_write);

    return 0;
}
```

MAX_PROCESS의 2배만큼 temp.txt에 숫자를 기록하는 방식이다.

- 결과(MAX_PROCESS 8의 경우)

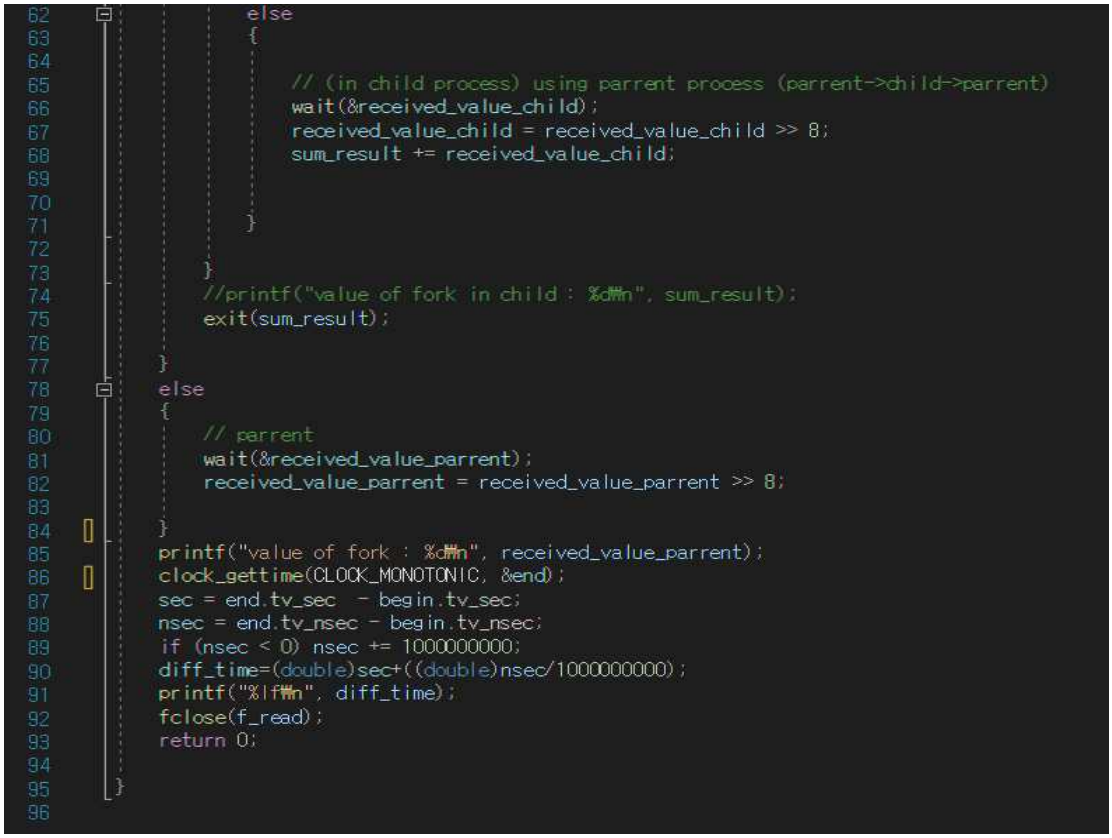


```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

MAX_PROCESS의 2배인 16까지의 숫자를 기록하는 모습을 볼 수 있다.

- fork.c 프로그램

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <math.h>
#include <time.h>

#define MAX_PROCESS 64

int main()
{
    struct timespec begin, end; //시간을 측정하기 위한 변수
    time_t sec;
    long nsec;
    double diff_time;
    clock_gettime(CLOCK_MONOTONIC, &begin);
    int sum_result=0;
    FILE *f_read = fopen("temp.txt","r");
    pid_t bottom_pid;
    int i, sum;
    int received_value_child,received_value_parrent;

    pid_t top_pid = fork();
    if (top_pid < 0)
    {
        printf("fork error.₩n");
        return 0;
    }
    else if (top_pid == 0)
    {
        for (i = 0; i < MAX_PROCESS; i++)
        {
            if ((bottom_pid = fork()) < 0)
            {
                printf("fork error.₩n");
                return 0;
            }
            else if (bottom_pid == 0)
            {
                //(in child process) using child process (parrent->child->child)
                int num1 = 0, num2 = 0;
                fscanf(f_read, "%d₩n", &num1);
                fscanf(f_read, "%d₩n", &num2);
                //printf("child에서 num1 %d 보냅니다.₩n", num1);
                /*
                if (feof(f_read) != 0)
                {
                    printf("파일의 끝에 도달했습니다.₩n");

                }
                */
                sum = num1 + num2;
                //printf("child에서 sum %d 보냅니다.₩n", sum);
                //fprintf(f_write, "%d₩n",sum);
                exit(sum);

            }
```



```
62                  else
63                  {
64
65                      // (in child process) using parrent process (parrent->child->parrent)
66                      wait(&received_value_child);
67                      received_value_child = received_value_child >> 8;
68                      sum_result += received_value_child;
69
70
71                  }
72
73              }
74              //printf("value of fork in child : %d\n", sum_result);
75              exit(sum_result);
76
77          }
78          else
79          {
80              // parrent
81              wait(&received_value_parrent);
82              received_value_parrent = received_value_parrent >> 8;
83
84          }
85          printf("value of fork : %d\n", received_value_parrent);
86          clock_gettime(CLOCK_MONOTONIC, &end);
87          sec = end.tv_sec  - begin.tv_sec;
88          nsec = end.tv_nsec - begin.tv_nsec;
89          if (nsec < 0) nsec += 1000000000;
90          diff_time=(double)sec+((double)nsec/1000000000);
91          printf("%lf\n", diff_time);
92          fclose(f_read);
93          return 0;
94
95      }
96
```

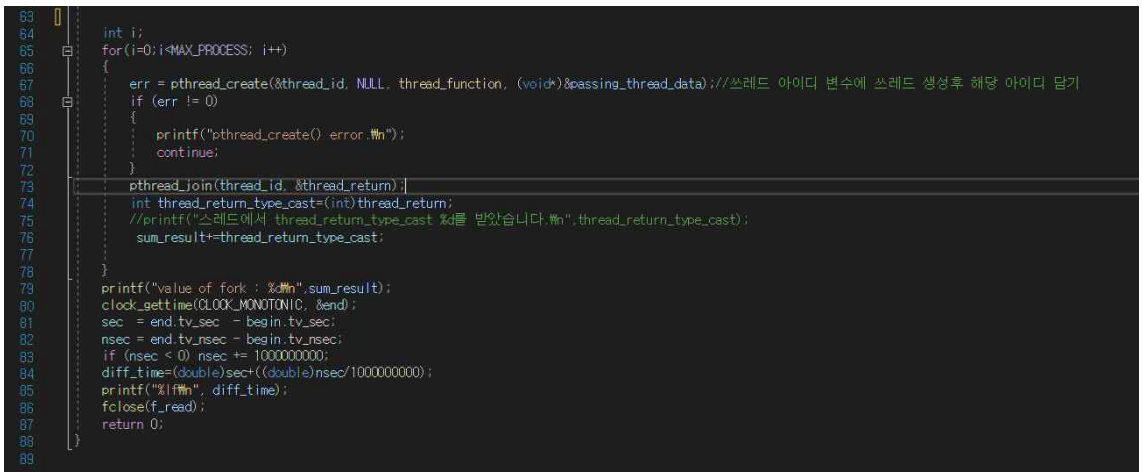
최상단의 프로세스를 구현을 위해 가장 먼저 fork 함수를 실행하여 차일드 프로세스를 만들고 차일드 프로세스에서 for문을 통해 MAX_PROCESS만큼 차일드 프로세스를 만들어 파일에 접근한다. 그래고 연산 결과를 wait을 통해 기다리는 부모에게 exit함수를 통해 전달하는 방식이다.

-thread.c 프로그램

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <math.h>
#include <time.h>
#include <pthread.h>

#define MAX_PROCESS 64

//스레드 함수에 인자값으로 넣어줄 데이터를 구조체를 정의하여 사용
typedef struct passing_thread_data_struct
{
    //쓰레드 함수에서 사용할 변수들을 구성하는 구조체이며
    //쓰레드 함수에서 쓰인다.
    FILE* f_read_in_thread;

}passing_thread_data_struct;// 예명으로 사용한다.
```

```c
void* thread_function(void* passing_thread_data)
{
    //스레드함수이며
    //인자는 void형 포인터 자료형 이는 구조체 변수의 void*형으로 타입 케스트를 통해
    //인자로 넘겨줄 것이다.

    passing_thread_data_struct* passing_thread_data_in_thread_function =(passing_thread_data_struct*)passing_thread
    int num1=0,num2=0;
    fscanf(passing_thread_data_in_thread_function->f_read_in_thread, "%d₩n", &num1);
    fscanf(passing_thread_data_in_thread_function->f_read_in_thread, "%d₩n", &num2);
    //printf("스레드에서 pos %d를 받았습니다.₩n",pos);
    //printf("스레드에서 num1 %d 읽습니다.₩n", num1);
    /*
    if (feof(f_read) != 0)
    {

        printf("파일의 끝에 도달했습니다.₩n");

    }
    */
    int sum=num1+num2;
    //printf("스레드에서 sum %d 보냅니다.₩n", sum);
    return (void*)sum;//값 반환
}
```

```c

int main()
{
    struct timespec begin, end; //시간을 측정하기 위한 변수
    clock_gettime(CLOCK_MONOTONIC, &begin);
    time_t sec;
    long int nsec;
    double diff_time;
    int sum_result=0;
    pthread_t thread_id;//스레드 아이디를 생성한다.
    int err; //스레드 생성에 실패하면 에러를 저장하기 위해 생성
    void* thread_return; //스레드 함수의 리턴값을 저장할 변수
    passing_thread_data_struct passing_thread_data;

    FILE *f_read = fopen("temp.txt","r");
    passing_thread_data.f_read_in_thread = f_read;
```



```
    int i;
    for(i=0;i<MAX_PROCESS; i++)
    {
        err = pthread_create(&thread_id, NULL, thread_function, (void*)&passing_thread_data);//쓰레드 아이디 변수에 쓰레드 생성후 해당 아이디 담기
        if (err != 0)
        {
            printf("pthread_create() error.₩n");
            continue;
        }
        pthread_join(thread_id, &thread_return);
        int thread_return_type_cast=(int)thread_return;
        //printf("스레드에서 thread_return_type_cast %d를 받았습니다.₩n",thread_return_type_cast);
         sum_result+=thread_return_type_cast;

    }
    printf("value of fork : %d₩n",sum_result);
    clock_gettime(CLOCK_MONOTONIC, &end);
    sec  = end.tv_sec  - begin.tv_sec;
    nsec = end.tv_nsec - begin.tv_nsec;
    if (nsec < 0) nsec += 1000000000;
    diff_time=(double)sec+((double)nsec/1000000000);
    printf("%lf₩n", diff_time);
    fclose(f_read);
    return 0;
}
```

fork방식과는 다르게 thread.c는 for문을 통해 MAX_PROCESS만큼 쓰레드를 만들고 쓰레드 함수를 통해 파일에 접근하여 연산한다. 결과는 pthread_join함수의 인자로 들어가는 thread_return 변수를 통해 받는다.

- 결과

(MAX_PROCESS 8의 경우)

```
root@ubuntu:/home/os2018202065/os_3_2018202065_C/assignment3-1# ./folk
value of fork : 136
0.005431
root@ubuntu:/home/os2018202065/os_3_2018202065_C/assignment3-1# ./thread
value of fork : 136
0.001026
root@ubuntu:/home/os2018202065/os_3_2018202065_C/assignment3-1#
```

(MAX_PROCESS 64의 경우)

```
root@ubuntu: /home/os2018202065/os_3_2018202065_C/assignment3-1
root@ubuntu:/home/os2018202065/os_3_2018202065_C/assignment3-1# ./numgen
root@ubuntu:/home/os2018202065/os_3_2018202065_C/assignment3-1# ./folk
value of fork : 64
0.028788
root@ubuntu:/home/os2018202065/os_3_2018202065_C/assignment3-1# ./thread
value of fork : 8256
0.013635
root@ubuntu:/home/os2018202065/os_3_2018202065_C/assignment3-1#
```

MAX_PROCESS 64의 경우에서 fork 방식은 결과 값이 64가 나오고 thread방식은 예상 결과인 8256이 나온 이유를 설명하는 아래와 같다.

먼저 fork방식에 있어 부모 프로세스에 연산결과 8256을 전달하여야 하는데 이럴 경우 차일드 프로세스에서 종료를 위한 exit함수에 인자값으로 8256이 들어가게 되는데 exit함수는 총 16비트를 부모의 wait함수의 주소값 인자 변수에 들어가게 된다. 이때 총 16비트 중 하위 8비트 값이 0이면 상위 8비트 값은 종료할 때 exit 함수에 전달한 인자값이 되게 된다.

이러한 방식으로 부모 프로세스의 wait 함수에서 받은 리턴값을 (>> 8)을 통해 오른쪽으로 8번 shift 하게 되면 exit함수의 입력인자를 출력할 수 있다. 이것을 8256으로 적용하면 0010 0000 0100 0000에서 하위 8비트 0100 0000 만이 exit함수의 입력인자로 입력되고 이걸 wait에서 받으면 0100 0000 0000 0000이 되는데 오른쪽으로 8번 shift 하게 되면 0100 0000 즉 64가 된다. 추가적으로 wait에서 받은 16바트의 값이

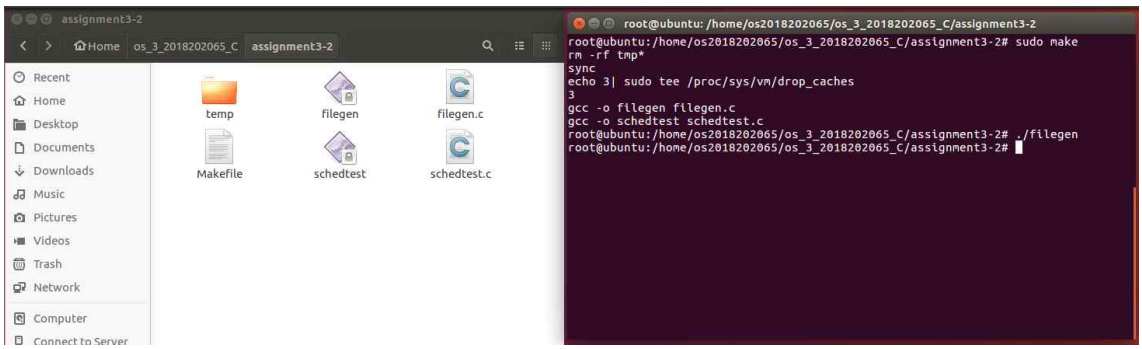
하위 8비트 값이 0x7f일 때는 자식 프로세스가 SIGTSTP이나 SIGSTOP 시그널에 의해 임시 중단 상태이며 상위 8비트에 시그널 번호가 된다. 비정상 종료일 때는 상위 8비트 값은 0이지만 하위 8비트 중에 7비트는 시그널 번호가 오고 맨 첫번째 비트는 코어 플래그가 온다. 코어 플래그 값은 코어 파일 발생 여부 정보를 의미한다. 그림으로 설명하면 아래와 같다.

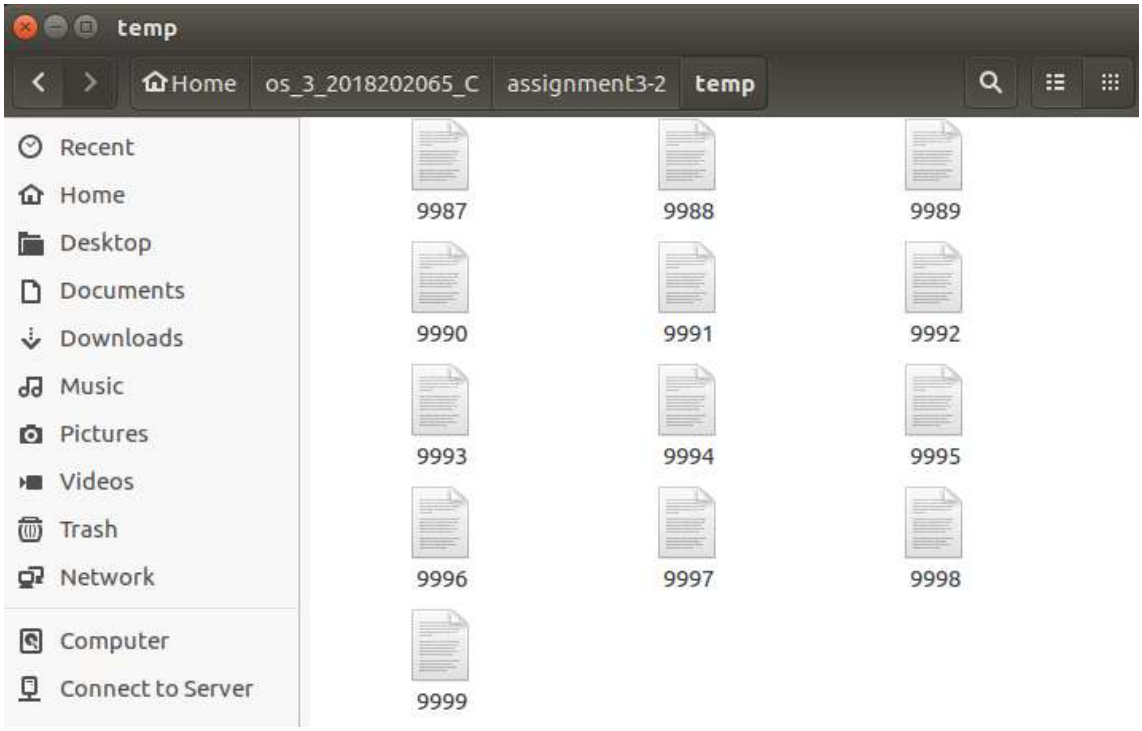
Assignment 3-2

- filegen (filegen.c) 프로그램

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <math.h>
#include <time.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <pwd.h>
#define MAX_PROCESS 10000

int main()
{

    umask(0000);// 초기설정된 umask값을 변경
    int mkdir_err=mkdir("temp", S_IRWXU | S_IRWXG | S_IRWXO);//cache폴더 생성
    if (mkdir_err == -1)
    {
        printf("mkdir error₩n");
        return 0;
    }
    int i;
    for (i = 0; i < MAX_PROCESS; i++)
    {
        char buf[10];
        sprintf(buf, "%d", i);
        char dire[100] = "temp/";
        strcat(dire, buf);
        //printf("%s₩n", dire);
        FILE* f_write = fopen(dire,"w");
        fprintf(f_write, "%d", 1 + rand() % 9);
        fclose(f_write);

    }


}
```

- filegen (filegen.c) 프로그램 결과





0~9999즉 10000개의 파일이 temp 디렉터리에 만들어진 것을 확인할 수 있다.

- schedtest (schedtest.c) 프로그램

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <math.h>
#include <time.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <pwd.h>
#include <sched.h>

#define MAX_PROCESS 10000

int main()
{


    ////////////////////////////////////////////////////////
    struct sched_param param;
    ////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    pid_t pid;
    int received_value_child;
    ////////////////////////////////////////////////////////////////////
```

```c
    for (int policy = 0; policy < 3; policy++)
    {

        ////////////////////////////////////////////////////
        struct timespec begin, end; //시간을 측정하기 위한 변수
        time_t sec;
        long int nsec;
        double diff_time;
        ////////////////////////////////////////////////////

        ////////////////////////////////////////////////////
        int inc;
        int priority=0;
        ////////////////////////////////////////////////////

        for (int select_priority = 0; select_priority < 3; select_priority++)
        {

            ////////////////////////////////////////////////////////////////////
            //결과에 영향을 주는 버퍼, 캐시 제거 및 sync 동기화
            int ret1 = system("rm -rf tmp*");
            int ret2 = system("sync");
            int ret3 = system("echo 3 | sudo tee /proc/sys/vm/drop_caches");
            ////////////////////////////////////////////////////////////////////
```

중첩 for문과 swich문의 조합으로 3개의 정책에 대한 Highest, Default, Lowest 3개의 경우에서 실행시간의 차이를 보기 위해 설계하였고 각 정책의 3가지의 경우를 실행하기 전 system함수를 통해 결과에 영향을 주는 인자들을 제거하는 명령어를 쉘에 입력하여 주었다.

```
56
57          switch (policy)
58          {
59          case 0:
60              printf("The standard round robin with time sharing policy₩n");//The standard round robin with time sharing policy
61              switch (select_priority)
62              {
63              case 0://Highest
64                  //priority자동으로 0으로 고정
65                  inc=-20;
66                  break;
67              case 1://Default
68                  //priority자동으로 0으로 고정
69                  inc = 0;
70                  break;
71              case 2://Lowest
72                  //priority자동으로 0으로 고정
73                  inc = 19;
74                  break;
75              default:
76                  break;
77              }
78              break;
```

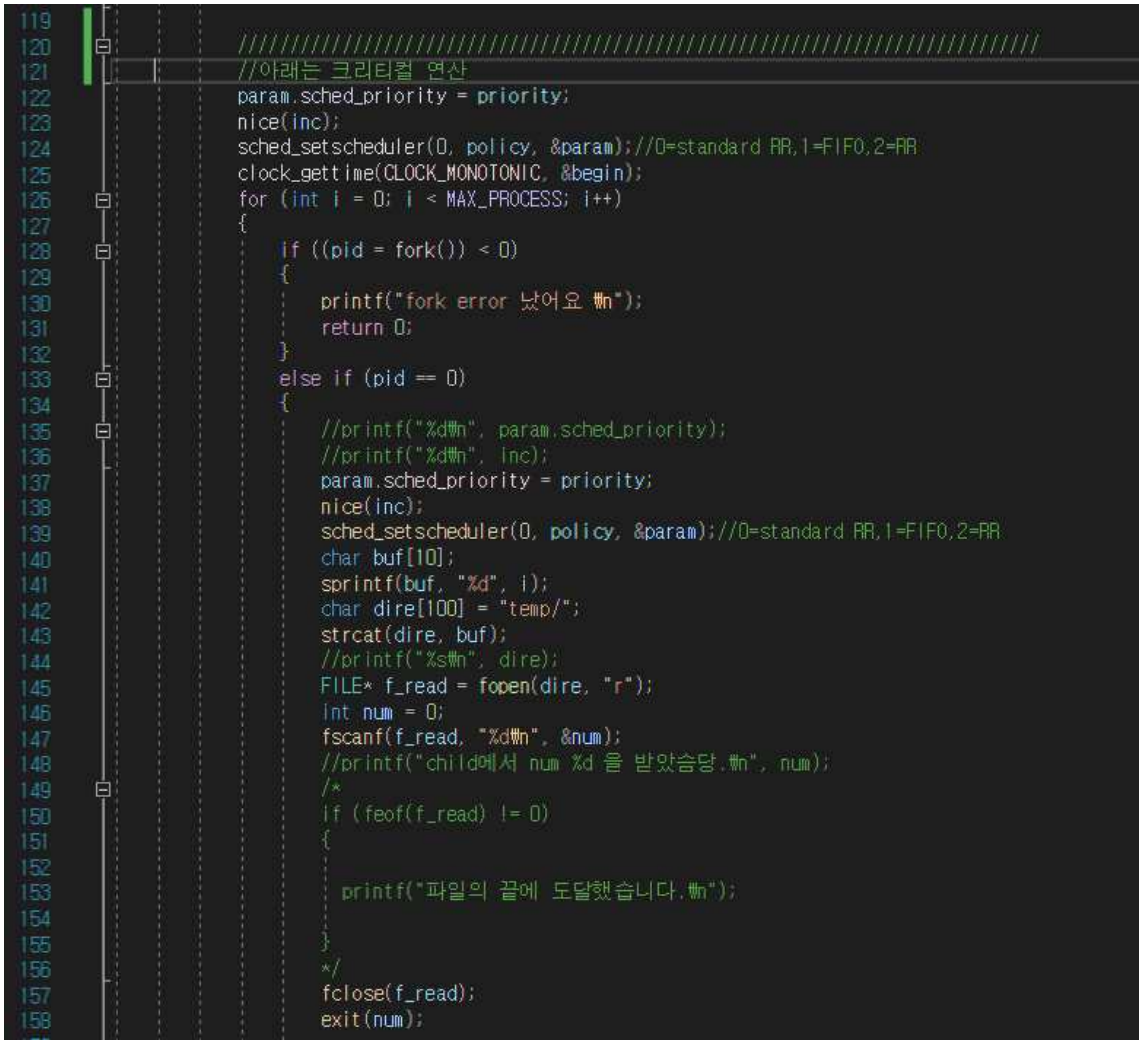
The standard round robin with time sharing policy에서는 Priority가 0으로 고정되어 사용됨으로 nice 값만 변화시키면 된다. 그래서 nice함수의 인자로 들어가는 int inc의 값을 highest, default, lowest로 나누어 설정하였다.

```
79              case 1:
80                  printf("The first in, first out policy₩n");//The first in, first out policy
81                  switch (select_priority)
82                  {
83                  case 0://Highest
84                      priority = 99;
85                      inc = -20;
86                      break;
87                  case 1://Default
88                      priority = 50;
89                      inc = 0;
90                      break;
91                  case 2://Lowest
92                      priority = 1;
93                      inc = 19;
94                      break;
95                  default:
96                      break;
97                  }
98                  break;
```

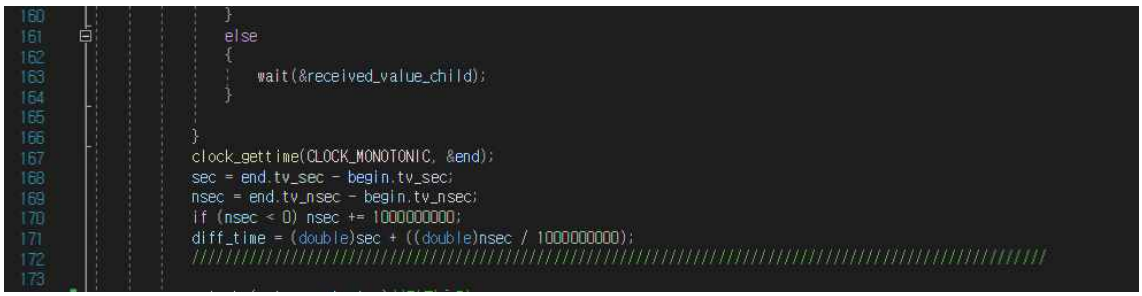
The first in, first out policy에서는 Priority가 1~99까지 존재하고 나이스 함수와 함께 highest, default, lowest 설정을 할 수 있으므로 그래서 nice함수의 인자로 들어가는 int inc와 param.sched_priority = priority에 들어가는 priority변수를 조정하여 highest, default, lowest를 나누어 설정하였다. Highest는 우선순위를 가장 높이기 위하여 priority 99에 inc –20으로 두었고 Default는 priority 부분에 1~99의 중간값인 50 inc에는 0을 Lowest는 우선순위가 가장 낮은 priority 1와 inc를 19로 두었다.

```
99              case 2:
100                 printf("The round robin policy₩n");//The round robin policy
101                 switch (select_priority)
102                 {
103                 case 0://Highest
104                     priority = 99;
105                     inc = -20;
106                     break;
107                 case 1://Default
108                     priority = 50;
109                     inc = 0;
110                     break;
111                 case 2://Lowest
112                     priority = 1;
113                     inc = 19;
114                     break;
115                 default:
116                     break;
117                 }
118             }
```

The round robin policy에서는 Priority가 1~99까지 존재하고 나이스 함수와 함께 highest, default, lowest 설정을 할 수 있으므로 그래서 nice함수의 인자로 들어가는 int inc와 param.sched_priority = priority에 들어가는 priority변수를 조정하여 highest, default, lowest를 나누어 설정하였다. Highest는 우선순위를 가장 높이기 위하여 priority 99에 inc –20으로 두었고 Default는 priority 부분에 1~99의 중간값인 50 inc에는 0을 Lowest는 우선순위가 가장 낮은 priority 1와 inc를 19로 두었다.



```
119
120             ////////////////////////////////////////////////////////////////////////
121             //아래는 크리티컬 연산
122             param.sched_priority = priority;
123             nice(inc);
124             sched_setscheduler(0, policy, &param);//0=standard RR,1=FIFO,2=RR
125             clock_gettime(CLOCK_MONOTONIC, &begin);
126             for (int i = 0; i < MAX_PROCESS; i++)
127             {
128                 if ((pid = fork()) < 0)
129                 {
130                     printf("fork error 났어요 ₩n");
131                     return 0;
132                 }
133                 else if (pid == 0)
134                 {
135                     //printf("%d₩n", param.sched_priority);
136                     //printf("%d₩n", inc);
137                     param.sched_priority = priority;
138                     nice(inc);
139                     sched_setscheduler(0, policy, &param);//0=standard RR,1=FIFO,2=RR
140                     char buf[10];
141                     sprintf(buf, "%d", i);
142                     char dire[100] = "temp/";
143                     strcat(dire, buf);
144                     //printf("%s₩n", dire);
145                     FILE* f_read = fopen(dire, "r");
146                     int num = 0;
147                     fscanf(f_read, "%d₩n", &num);
148                     //printf("child에서 num %d 을 받았습당.₩n", num);
149                     /*
150                     if (feof(f_read) != 0)
151                     {
152
153                     printf("파일의 끝에 도달했습니다.₩n");
154
155                     }
156                     */
157                     fclose(f_read);
158                     exit(num);
```



```
160             }
161             else
162             {
163                 wait(&received_value_child);
164             }
165
166         }
167         clock_gettime(CLOCK_MONOTONIC, &end);
168         sec = end.tv_sec - begin.tv_sec;
169         nsec = end.tv_nsec - begin.tv_nsec;
170         if (nsec < 0) nsec += 1000000000;
171         diff_time = (double)sec + ((double)nsec / 1000000000);
172         ////////////////////////////////////////////////////////////////////////////////
173
```

해당 부분은 실행시간에 영향을 주는 크리티컬 섹션으로 MAX_PROCESS의 개수만큼 fork하여 차일드 프로세스가 파일에 접근한다. 또한 이 부분에서 정책을 바꾸는 함수인 sched_setscheduler를 사용하였는데 인자값 0을 줌으로써 해당 함수를 실행하는 프로세스 정책을 바꿀 수 있도록 하였고 policy는 중첩 for문의 바깥 for문의 인자로 0,1,2까지 수행되며 0은 The standard round robin with time sharing policy 1은 The first in, first out policy 2는 The round robin policy를 나타낸다. 또한 nice함수도 이 부분에서 사용한다.

다음은 해당 프로그램에서 위의 함수를 사용한 것에 대한 설명이다.

크리티컬 섹션에 들어오자마자 부모 프로세스는 해당하는 정책으로 바꾸게 되고 for문을 수행하게 되는 for문이 돌아가는 동안 fork를 통한 자식 프로세스의 영역에서도 부모 프로세

스 정책과 동일하게 정책을 바꾼다. 이러한 이유는 부모 프로세스의 정책만을 바꾼다 하더라도 자식 프로세스가 정책을 바꾸지 않으면 실질적으로 해당 프로그램은 자식 프로세스의 수행으로 동작하기 때문에 결과 분석에 있어 영향을 미치지 않는다고 생각하였기 때문이다.

- schedtest (schedtest.c) 프로그램 결과(MAX_PROCESS 10000개)

```
3
The standard round robin with time sharing policy
Highest: 2.032255
3
The standard round robin with time sharing policy
default: 2.083592
3
The standard round robin with time sharing policy
Lowest: 2.231423
3
The first in, first out policy
Highest: 2.782981
3
The first in, first out policy
default: 2.816815
3
The first in, first out policy
Lowest: 2.798965
3
The round robin policy
Highest: 2.834279
3
The round robin policy
default: 1.792933
3
The round robin policy
Lowest: 2.794680
```

실험을 진행하면서 영향을 줄 수 있는 요소를 지우기 위하여 매 실행에 있어 버퍼와 캐시를 지우고 sync를 하였다. 결과를 보게 되면 The standard round robin with time sharing policy 같은 경우에는 Lowest가 가장 느린 실행시간을 가지고 있는 것을 볼 수 있다. 또한, Default보다도 Highest가 실행시간이 빠름을 알 수 있다. The first in, first out policy 같은 경우에는 Highest의 수행시간이 가장 빠르고 다음은 Lowest, Default가 가장 느린 실행시간을 가짐을 알 수 있다.

The standard round robin with time sharing policy의 경우에는 default의 수행시간이 가장 빠르고 다음은 Lowest, Highest가 가장 느린 실행시간을 가짐을 알 수 있다. 이렇게 각 3가지 정책의 3가지 경우(3X3=9)마다 실행 후 완료되는 시간이 차이가 있음을 알 수 있다. 워낙 매번 실행마다 편차가 크기 때문에 3가지 정책의 3가지의 경우의 즉 9가지의 방식에 따라 실행시간이 각각 조금의 차이가 있음만을 확인할 수 있었고 어느 것이 우열에 있는 스케줄링 정책인지는 비교할 수 없다. 하지만 대체로 The standard round robin with time sharing policy 경우에는 평균적으로 FIFO나 RR보다 빠른 실행시간을 가짐을 알 수 있었다.

Assignment 3-3

- process_tracer.c

```c
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>
#include <asm/syscall_wrapper.h>
#include <linux/unistd.h>
#include <linux/string.h>
#include <linux/sched.h>
#include <linux/init_task.h>

#define __NR_ftrace 336

char kernel_buffer[1000] = {0,}; // 커널 버퍼 변수

typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs*);
static sys_call_ptr_t *sys_call_table;

sys_call_ptr_t origin_origin_ftrace;
char *system_call_table = "sys_call_table";

void make_rw(void *addr){
    //시스템콜 테이블에 rw권한을 주는 함수
    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){
    //시스템콜 테이블에 rw권한을 뺏는 함수
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);

    pte->pte = pte->pte &~ _PAGE_RW;
}

static asmlinkage pid_t process_trace(const struct pt_regs* regs)
{
    //hooking 후 실행하는 ftrace함수
    int error_check = 1;
    struct task_struct* task;
    struct task_struct* children_task;
    struct list_head* children_list;
    int children_count = 0;
    struct task_struct* sibling_task;
    struct list_head* sibling_list;
    struct task_struct* parent_task;
    int sibling_count = 0;
    pid_t given_pid = regs->di;// 프로세스 피아이디를 저장할 변수
    printk("%d\n", given_pid);
```

```
{
    for_each_process(task)
    {
        if (task->pid == given_pid)
        {
            ////////////////////
            error_check = 0;
            ////////////////////

            ////////////////////////////////////////////////////////////////////////
            printk("##### TASK INFORMAITON OF ''[%d] %s '' #####\n", task->pid, task->comm);
            ////////////////////////////////////////////////////////////////////////

            ////////////////////////////////////////////////////////////////////////
            if (task->state == 0)
                printk("- task state : Running or ready\n");
            else if (task->state == 1)
                printk("- task state : Wait with ignoring all signals\n");
            else if (task->state == 2)
                printk("- task state : Wait\n");
            else if (task->state == 4)
                printk("- task state : Stopped\n");
            else if (task->state == 32)
                printk("- task state : Zombie process\n");
            else if ((task->state == 16) || (task->state == 128))
                printk("- task state : Dead\n");
            else
                printk("- task state : etc.\n");
            ////////////////////////////////////////////////////////////////////////


            ////////////////////////////////////////////////////////////////////////
            printk("- Process Group Leader : [%d] %s\n", task->group_leader->pid, task->group_leader->comm);
            ////////////////////////////////////////////////////////////////////////


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

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


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

```
102
103
104                 ////////////////////////////////////////////////////////////////////////////////////////////////
105                 printk("- it's sibling process(es) : \n");
106                 list_for_each(sibling_list, &parent_task->children)//부모의 자식은 task의 어떠한 형제이다.
107                 {
108                     sibling_task = list_entry(sibling_list, struct task_struct, sibling);
109                     if (sibling_task->pid != task->pid) {
110                         printk("    > [%d] %s\n", sibling_task->pid, sibling_task->comm);
111                         sibling_count++;
112                     }
113
114                     // task now points to one of task's sibling
115                 }
116                 if (sibling_count == 0) printk("    > It has no sibling.\n");
117                 printk("    > This process has %d sibling process(es) \n", sibling_count);
118                 ////////////////////////////////////////////////////////////////////////////////////////////////
119
120                 ////////////////////////////////////////////////////////////////////////////////////////////////
121                 printk("- it's children process(es) : \n");
122                 list_for_each(children_list, &task->children) {
123                     children_count++;
124                     children_task = list_entry(children_list, struct task_struct, sibling);
125                     printk("    > [%d] %s\n", children_task->pid, children_task->comm);
126
127                     // task now points to one of task's children
128                 }
129                 if (children_count == 0) printk("    > It has no child.\n");
130                 printk("    > This process has %d child process(es) \n",children_count);
131                 ////////////////////////////////////////////////////////////////////////////////////////////////
132
133                 ////////////////////////////////////////////////////////////////////////
134                 printk("##### END OF INFORMATION ######\n");
135                 ////////////////////////////////////////////////////////////////////////
136
137
138                 return(task->pid);
139             }
140         }
141
142         if (error_check==1)
143         {
144             printk(KERN_INFO "ERROR CAUSE : Don't print imporation for this PID\n");
145             return -1;
146         }
147         return-1;
148
149     }
150
```

```
151
152 static int __init hooking_init(void)
153 {
154     //struct task_struct *findtask = &init_task;
155     //hooking 모듈 적재 함수
156     sys_call_table = (sys_call_ptr_t *) kallsyms_lookup_name(system_call_table);
157     make_rw(sys_call_table);
158
159     origin_origin_ftrace = sys_call_table[__NR_ftrace];
160     sys_call_table[__NR_ftrace] = (sys_call_ptr_t)process_trace;// 시스템 콜 테이블은 long 타입이기 때문에 타입 cast
161     /*
162     do
163     {
164         //printk("%s[%d] ->", findtask->comm, findtask->pid);
165         process_trace(findtask->pid);
166         findtask = next_task(findtask);
167     } while ((findtask->pid != init_task.pid));
168     process_trace(findtask->pid);
169     */
170
171
172     //printk(KERN_INFO "Operate insmod ftracehooking.\n"); //모듈 적재를 확인하기위한 프린트문
173     return 0;
174 }
175
176 static void __exit hooking_exit(void){
177     sys_call_table[__NR_ftrace]= origin_origin_ftrace;
178     make_ro(sys_call_table);
179     //printk(KERN_INFO "Operate rmmod ftracehooking.\n"); //모듈 해제를 확인하기위한 프린트문
180 }
181
182 module_init(hooking_init);
183 module_exit(hooking_exit);
184 MODULE_LICENSE("GPL");
185
```

해당 커널 소스파일의 실행 알고리즘은 아래와 같다.

1. insmod를 통해 기존의 ftrace 후킹 후 test.c 파일에서 syscall(__NR_ftrace, pid 값) 을 호출하여 process_trace() 함수를 실행한다.
2. 프로세스 pid를 인자를 받은 process_trace() 함수는 해당하는 인자의 task가 리스트에 있는지 확인 있으면 error_check 변수의 값을 0으로 바꿔주고 정상적인 결과 출력 후 해당 pid를 반환한다.
3. 만약 인력받은 pid 가 task 리스트에 없을 시 에러 체크를 통해 –1 반환한다.
4. rmmod 를 통해 원래의 ftrace 로 대체하고 종료한다. 또한, task_struct 의 state 값은 아래와 같이 정의하였다.

```
/* Used in tsk->state: */
#define TASK_RUNNING            0x0000
#define TASK_INTERRUPTIBLE      0x0001
#define TASK_UNINTERRUPTIBLE        0x0002
#define __TASK_STOPPED          0x0004
#define __TASK_TRACED           0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD           0x0010
#define EXIT_ZOMBIE         0x0020
#define EXIT_TRACE          (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED         0x0040
#define TASK_DEAD           0x0080
#define TASK_WAKEKILL           0x0100
#define TASK_WAKING         0x0200
#define TASK_NOLOAD         0x0400
#define TASK_NEW            0x0800
#define TASK_STATE_MAX          0x1000
```

sched.h의 state정보를 통해

Running or ready	0
Wait with ignoring all signals	1
Wait	2
Stopped	4
Zombie process	32
Dead	16 or 128
etc.	그밖에 모든 숫자.

- 리눅스 sched.h의 변경 부분

```
1205
1206        /*
1207         * New fields for task_struct should be added above here, so that
1208         * they are included in the randomized portion of task_struct.
1209         */
1210        randomized_struct_fields_end
1211        int             fork_count;//////////////////////카운터 변수 선언 무조건 여기
1212
1213        /* CPU-specific state of this task: */
1214        struct thread_struct        thread;
1215
1216        /*
1217         * WARNING: on x86, 'thread_struct' contains a variable-sized
1218         * structure.  It *MUST* be at the end of 'task_struct'.
1219         *
1220         * Do not put anything below here!
1221         */
1222    };
```

task_struct의 사용자를 위한 새로운 변수를 정의하기 위해선 다음과 같은 위치에서 선언해야 한다.

- 리눅스 fork.c의 변경 부분

```
2171    long _do_fork(unsigned long clone_flags,
2172            unsigned long stack_start,
2173            unsigned long stack_size,
2174            int __user *parent_tidptr,
2175            int __user *child_tidptr,
2176            unsigned long tls)
2177    {
2178        struct completion vfork;
2179        struct pid *pid;
2180        struct task_struct *p;
2181        int trace = 0;
2182        long nr;
2183
2184        /*
2185         * Determine whether and which event to report to ptracer.  When
2186         * called from kernel_thread or CLONE_UNTRACED is explicitly
2187         * requested, no event is reported; otherwise, report if the event
2188         * for the type of forking is enabled.
2189         */
2190        if (!(clone_flags & CLONE_UNTRACED)) {
2191            if (clone_flags & CLONE_VFORK)
2192                trace = PTRACE_EVENT_VFORK;
2193            else if ((clone_flags & CSIGNAL) != SIGCHLD)
2194                trace = PTRACE_EVENT_CLONE;
2195            else
2196                trace = PTRACE_EVENT_FORK;
2197
2198            if (likely(!ptrace_event_enabled(current, trace)))
2199                trace = 0;
2200        }
2201
2202        p = copy_process(clone_flags, stack_start, stack_size,
2203                 child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
2204        add_latent_entropy();
2205
2206        if (IS_ERR(p))
2207            return PTR_ERR(p);
2208        p->fork_count=0;////////////////////////////////////////////////////////카운터 초기화
2209        p->real_parent->fork_count++;//////////////////////////////////////////////////카운터 증가
2210
2211        /*
2212         * Do this prior waking up the new thread - the thread pointer
2213         * might get invalid after that point, if the thread exits quickly.
2214         */
```

_do_fork함수에서 p변수에 새로 생성된 task_struct * 구조체가 할당되면 해당 구조체의 fork_count값을 0으로 초기화 해당 구조체의 부모의 fork_count 값을 1 증가 시키는 방식으로 구현하였다.

- sched.h와 fork.c를 통한 process_tracer.c의 결과

(pid = 1 인 경우)

```
[ 1209.440412] ##### TASK INFORMAITON OF ''[1] systemd '' #####
[ 1209.440413] - task state : Wait with ignoring all signals
[ 1209.440414] - Process Group Leader : [1] systemd
[ 1209.440416] - Number of context switches : 3943
[ 1209.440417] - Number of calling fork() : 245
[ 1209.440418] - it's parnet process : [0] swapper/0
[ 1209.440418] - it's sibling process(es) :
[ 1209.440420]     > [2] kthreadd
[ 1209.440421]     > This process has 1 sibling process(es)
[ 1209.440421] - it's children process(es) :
[ 1209.440423]     > [398] systemd-journal
[ 1209.440424]     > [423] systemd-udevd
[ 1209.440426]     > [435] vmware-vmblock-
[ 1209.440427]     > [444] vmtoolsd
[ 1209.440428]     > [514] systemd-timesyn
[ 1209.440429]     > [923] VGAuthService
[ 1209.440431]     > [924] cron
[ 1209.440431]     > [925] acpid
[ 1209.440432]     > [926] dbus-daemon
[ 1209.440433]     > [952] systemd-logind
[ 1209.440434]     > [974] NetworkManager
[ 1209.440435]     > [976] accounts-daemon
[ 1209.440437]     > [979] rsyslogd
[ 1209.440437]     > [980] bluetoothd
[ 1209.440439]     > [1001] agetty
[ 1209.440440]     > [1004] lightdm
[ 1209.440441]     > [1017] irqbalance
[ 1209.440442]     > [1063] polkitd
[ 1209.440443]     > [1190] rtkit-daemon
[ 1209.440444]     > [1301] upowerd
[ 1209.440445]     > [1354] colord
[ 1209.440446]     > [1420] avahi-daemon
[ 1209.440447]     > [1472] whoopsie
[ 1209.440448]     > [1527] systemd
[ 1209.440450]     > [1534] gnome-keyring-d
[ 1209.440451]     > [2166] udisksd
[ 1209.440452]     > [2268] fwupd
[ 1209.440452]     > [6531] cupsd
[ 1209.440454]     > [6532] cups-browsed
[ 1209.440454]     > This process has 29 child process(es)
[ 1209.440455] ##### END OF INFORMATION #####
```

(pid = 92452의 경우)

```
[28211.835006] ##### TASK INFORMAITON OF ''[92454] cupsd '' #####
[28211.835008] - task state : Wait with ignoring all signals
[28211.835010] - Process Group Leader : [92454] cupsd
[28211.835011] - Number of context switches : 65
[28211.835013] - Number of calling fork() : 8
[28211.835015] - it's parnet process : [1] systemd
[28211.835016] - it's sibling process(es) :
[28211.835017]     > [399] systemd-journal
[28211.835019]     > [415] systemd-udevd
[28211.835021]     > [483] vmware-vmblock-
[28211.835023]     > [487] vmtoolsd
[28211.835024]     > [494] systemd-timesyn
[28211.835032]     > [903] VGAuthService
[28211.835035]     > [904] systemd-logind
[28211.835036]     > [905] cron
[28211.835038]     > [906] bluetoothd
[28211.835040]     > [910] rsyslogd
[28211.835042]     > [912] dbus-daemon
[28211.835042]     > [949] NetworkManager
[28211.835042]     > [955] acpid
[28211.835043]     > [961] accounts-daemon
[28211.835043]     > [993] agetty
[28211.835043]     > [997] lightdm
[28211.835044]     > [1005] polkitd
[28211.835044]     > [1021] irqbalance
[28211.835044]     > [1387] upowerd
[28211.835045]     > [1404] colord
[28211.835045]     > [1426] rtkit-daemon
[28211.835045]     > [1448] whoopsie
[28211.835045]     > [1465] systemd
[28211.835046]     > [1472] gnome-keyring-d
[28211.835046]     > [1980] udisksd
[28211.835046]     > [1982] fwupd
[28211.835047]     > [92455] cups-browsed
[28211.835047]     > [104475] avahi-daemon
[28211.835047]     > This process has 28 sibling process(es)
[28211.835048] - it's children process(es) :
[28211.835048]     > [92457] dbus
[28211.835049]     > [92458] dbus
[28211.835049]     > [92459] dbus
[28211.835050]     > [92460] dbus
[28211.835050]     > [92461] dbus
[28211.835050]     > [92462] dbus
[28211.835051]     > [92463] dbus
[28211.835051]     > [92464] dbus
[28211.835051]     > This process has 8 child process(es)
[28211.835052] ##### END OF INFORMATION #####
```

과제의 요구사항에 알맞은 결과가 나오는 것을 알 수 있다.

- 고찰

이번 과제를 구현하면서 발생한 이슈는 다음과 같다.

1. clock_gettime을 통해 실행시간을 구했을 시 음수 시간이 나오는 이슈
2. 스레드 함수에 인자를 넣어주는 방법에 대한 이슈
3. 3-2과제를 수행하면서 9개의 경우의 수에 관한 결과 출력을 빠르게 하는 법
4. 3-1과제에 있어 공유 자원인 temp.txt를 이용함에 있어 세마포어나 뮤텍스 없이 해당 연산을 수행하는 법에 대한 이슈
5. task_struct 구조체를 통해 parent에 접근할 때 task_struct * parent 구조체와 task_struct * real_parent 구조체 중 어느 것을 사용하는지에 대한 이슈
6. 리눅스의 커널의 버전마다 sched.h의 내부의 변수와 define한 매크로가 다름으로 해당 헤더 파일을 분석하여 결과를 도출해야 함을 알게 되었다.

해결했던 방식은 다음과 같다.

1. nsec = end.tv_nsec – begin.tv_nsec이 가끔 음수가 나오기 때문에
if (nsec < 0) nsec += 1000000000해당 조건문을 추가하여 음수의 nsec가 나오면 양수로 바꿔주어 해결하였다.
2. 인자 저장을 위한 구조체를 정의하였고 스레드 함수에 void *로 형변한 후 스레드 함수에서 사용하여 해결하였다.
3. 중첩 for문과 swich case문을 이용하여 결과를 도출하였다. 또한, 각 경우의 수를 수행하기 전 system함수를 사용하여 쉘에 버퍼와 캐시를 제거하고 sync를 할 수 있는 명령어를 입력하여 해결하였다.
4. for문 내부에서 fork와 thread를 통해 child process와 thread가 생성이 될 것인데 이 때 fork같은 경우 for문 내부의 부모 영역에서 wait함수를 이용하여 해결하였고 thread의 경우 for문 내부에서 thread함수를 실행 시킨후 pthread_join을 통해 thread가 종료 되었을 시 다음 thread를 생성하여 thread 함수를 실행하게 하여 해결하였다.
5. 이를 해결하기 위해선 task_struct * real_parent와 task_struct * parent의 차이를 찾아보았고 task_struct * real_parent는 프로세스를 생성한 부모의 태스크 디스크립터 주소를 저장하고 ask_struct * parent는 부모 프로세스를 의미한다. real_parent는 해당 프로세스를 생성해준 프로세스를 의미한다. 그런데 자식 프로세스 입장에서 부모 프로세스가 소멸된 경우 부모 프로세스를 다른 프로세스로 지정한다. 프로세스 계층 구조에서 지정한 부모 프로세스가 없을 경우 init 프로세스를 부모 프로세스로 변경한다. 이러한 점으로 인해 init 프로세스를 부모를 가지는 프로세스는 진짜 부모 프로세스를 확인하여 결과를 도출해야 한다. 생각하여 task_struct * real_parent을 사용하였다.