2022학년도 2학기

운영체제

-REPORT-

수업명	운영체제
과제 이름	assignment3
담당 교수님	최상호 교수님
학번	2018204058
이름	김민교

1. Introduction :

멀티 프로세스와 멀티 스레드를 이용해본다. 멀티 프로세스를 사용하는 방법과 멀티 스레드를 사용할 수 있다. 둘의 차이점을 알 수 있다. 스케줄링 정책을 직접 설정해보고 우선순위를 부여해본다. 이를 통해 어떤 작업에는 어떤 스케줄링 방식의 시간이 적게 걸리고 어떤 우선순위일 때 시간이 적게 걸리는지 확인할 수 있다. 프로세스의 정보를 저장하고 있는 태스크 스트럭를 통해 프로세스의 정보를 알아낸다. 또 fork() 함수가 어떤식으로 동작되는지 확인한다.

2. Conclusion & Analysis :

assignment3-1

[결과 출력]

MAX PROCESSES = 8

```
os2018204058@ubuntu:~/practice3/assign301$ ./fork
value of fork : 136
Time (Milli) : 1.333129
os2018204058@ubuntu:~/practice3/assign301$ ./thread
value of thread : 136
 Time (Milli) : 1.947481
```

MAX PROCESSES = 64

```
os2018204058@ubuntu:~/practice3/assign301$ ./fork
value of fork : 64
Time (Milli) : 13.125220
os2018204058@ubuntu:~/practice3/assign301$ ./thread
value of thread : 8256
 Time (Milli) : 9.566185
```

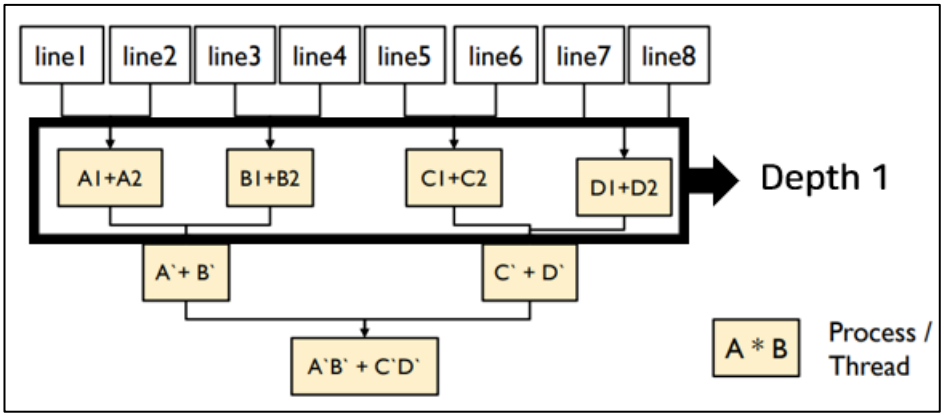
[numgen.c파일 설명]

```c
#include <stdio.h> // for using fopen

#define MAX_PROCESSES 64

int main(void){

        char *fname = "temp.txt";
        char *mode = "w";

        FILE *f_write = fopen(fname,mode);

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

        fclose(f_write);
        return 0;
}
```

파일 이름은 temp.txt로 한다. 파일을 열 때 모드를 w로 설정한다. MAX_PROCESS의 두배만큼 1부터 쓴다. 그리고 파일을 닫는다.

[다중 프로세스를 통한 구현 방법]

1. 파일은 Depth 1의 계산을 위해 MAX_PROCESS/2 번만 읽는다.



2. Depth 1의 계산 결과는 프로그램 내의 배열 result_arr[MAX_PROCESS]에 저장된다.
3. Depth 2부터는 파일에서 값을 읽어서 계산하는 것이 아니라, result_arr에서 읽어서 계산한다.
4. 재귀적으로 프로세스를 만들고 값을 계산한다.
5. 계산하는 데에 필요한 프로세스의 개수가 1이 되면 재귀함수를 빠져나와서 최종 결과 계산하고 출력한다.

[fork.c 코드 설명]

-메인 함수

```
42 int main(void)
43 {
44
45         pid_t pid,child_pid;
46         int status;
47         char *fname = "temp.txt";
48         int result_arr[MAX_PROCESSES]={0}; //array to store caculate
49
50         struct timespec begin, end;
51
52         clock_gettime(CLOCK_MONOTONIC, &begin);
```

line 45 ~ 47 : pid는 자식 프로세스인지, 부모 프로세스인지 판단하는 용도, child_pid는 부모 프로세스에서 wait()으로 자식 프로세스의 pid를 받을 때 저장하는 용도. status는 자식 프로세스가 exit()하면서 넘겨준 값을 저장하는 용도. fname 은 읽을 파일의 이름

line 48 : 파일안의 숫자들을 읽어 덧셈을 한 후 result_arr에 저장할 것이다.

line 50 ~ 52 : 시간을 측정하기 위한 변수를 선언하고 시작 시간을 측정한다.

```
54          //first, you have to read number in file.
55          // and store our array.
56          for(int i=0;i<MAX_PROCESSES;i++)
57          {
58                  pid = fork(); //if child,  pid == 0, if pareent, pid == child pid
59
60                  if(pid == -1){ printf("can't fork, error\n");exit(0);}
61
```

line 56~60 : MAX_PROCESSES만큼 프로세스를 생성한다. 반환값이 -1이면 에러를 의미하고 프로그램을 종료한다.

```
63                  if(pid == 0) //create child success
64                  {
65
66                          FILE *file = fopen(fname,"r"); //file open
67                          char first_line[5];
68                          char second_line[5];
69
70                          //for file pointer
71                          for ( int k=0;k<i;k++){
72                                  fgets(first_line,sizeof(first_line),file);
73                                  fgets(first_line,sizeof(first_line),file);
74                          }
75
76                          fgets(first_line,sizeof(first_line),file); // one line read
77                          fgets(second_line,sizeof(second_line),file); // two line read
78
79                          //string to int
80                          int num1 = atoi(first_line);
81                          int num2 = atoi(second_line);
82
83                          //close file
84                          fclose(file);
85                          exit(num1+num2);
86                  }
87          }
```

line 66 : 파일을 연다.

line 67 ~ 68 : 첫번째 숫자와 두번째 숫자를 읽어서 저장할 char형 배열

line 71 ~ 74 : 각 프로세스가 적절한 숫자를 읽게 하기 위해 파일 포인터를 옮기기 위한 반복문이다. 5번째 프로세스이면 5번의 반복을 돈다. 이 반복문 안에서 두 줄씩 숫자를 읽는다. 이렇게 되면 5번째 프로세스가 읽어야하는 적절한 파일 포인터에 도달하게 된다.

line 76 ~ 77 : 앞서 반복문을 통해 적절한 파일 포인터에 도달하였으니, 여기서부터 fgets를 써서 첫번째 숫자와 두번째 숫자를 읽는다.

line 80 ~ 81 : atoi함수를 통해 string을 int형으로 바꾼다.

line 84 ~ 85 : 파일을 닫고, 프로세스를 종료한다. exit()에는 읽은 숫자들의 합을 넣는다. 이 값은 부모 프로세스가 wait을 호출하여 전달받을 수 있다.

```
90          if (pid!=0){ //this means parent
91                  for(int i=0; i<MAX_PROCESSES; i++)
92                  {
93                          child_pid = wait(&status);
94                          int int_status = status>>8;
95                          result_arr[pid-child_pid] =int_status;
96                  }
97          }
```

line 90 : pid가 0이 아니면 부모 프로세스란 의미이다.

line 91 ~ 93 : 앞서 생성한 자식프로세스들이 끝날 때 까지 기다린다.

line 94 : exit()를 통해 전달한 덧셈값을 얻기 위해 status에 8번 오른쪽으로 shift 한다.

line 95 : result_arr에 덧셈 값을 저장한다. 어떤 프로세스가 먼저 끝날지 모르기 때문에, 이것을 판단하는 방법은 pid – child_pid로 구했다. 자식 프로세스는 순서대로 생겨나고 id도 순서대로 부여받기 때문에 index를 구하는데 적합하다고 판단했다.

```
 99          my_fork(result_arr,MAX_PROCESSES/2);
100
101         int final_result = result_arr[0]+result_arr[1];
```

line 99 :my_fork 함수를 호출한다. my_fork에서 생성할 프로세스 수는 MAX_PROCESSES의 절반 만큼만 생성하면 된다.

line 101 : 최종 결과값은 result_arr[0]과 result_arr[1]에 저장되어 있는 값을 더하면 된다.

```
103         clock_gettime(CLOCK_MONOTONIC, &end);
104
105         // time print
106         struct timespec temp;
107
108         long msec = end.tv_nsec - begin.tv_nsec;
109         if(msec <0)
110                 msec = 1000000000 + msec;
```

line 103 : 종료 시간을 측정한다.

line 106 ~ 110 : 시작 시간에서 종료 시간을 빼서 걸린 시간을 구한다. end.tv_nsec에서 begin.tv_nsec을 빼는데, 이 값이 음수인 경우가 있다. 이런 경우는 end가 1000000000이 넘어가서 다시 리셋되어 처음부터 세는 경우이다. 그래서 이때는 end - begin에다가 1000000000를 더해준다.

```
112         printf("value of fork : %d\n",final_result);
113         printf("Time (Milli) : %lf\n", (double) msec/1000000);
```

line 112 ~ 113 : 결과를 출력한다.

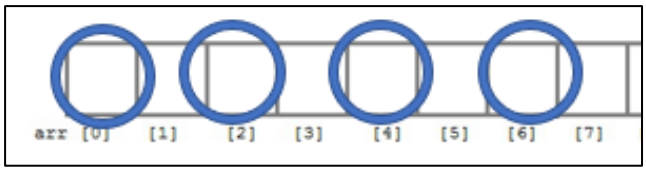
-my_fork 함수

```
12 void my_fork(int result_arr[],int process_num){
13         if(process_num == 1){ //this means caculation ends.
14                 return;
15         }
16
```

line 13 ~ 15 : process_num이 1이 될 때, 재귀함수를 종료한다. process_num이 1이란 말은 계산을 하는데에 필요한 프로세스가 한 개란 뜻이다. 이는 최종적으로 부모 프로세스에서 한번 계산하면 된다. 더 이상 자식 프로세스를 만들지 않고 main 함수로 돌아가 부모 프로세스에서 계산하겠다는 의미이다.

```
17         pid_t pid, child_pid;
18         int status;
19         // we have to caculate two numbers at once
20         for(int i=0; i<process_num*2;i=i+2){
21                 pid= fork();
22
23                 if(pid ==-1) { printf("can't fork, sorry\n"); return; }
24                 if(pid == 0) //this mean  child process
25                 {
26
27                         int sum=result_arr[i]+result_arr[i+1];
28                         exit(sum); //child process terminate with number sum
29                 }
30         }
31
```

line 20 : 배열의 짝수 번째 인덱스에 접근하여 두 개의 숫자를 더한다. 그리고 종료할 때 합을 exit()를 통해 부모 프로세스에게 전달한다.




```
31
32         if(pid!=0){ //this mean parent process
33                 for(int i=0;i<process_num;i++){
34                         child_pid = wait(&status); //wait until child process terminates
35                         int int_status = status>>8;
36                         result_arr[pid-child_pid]=int_status;
37                 }
38         }
39         my_fork(result_arr, process_num/2);
40 }
```

line 32 : 부모 프로세스일 경우에 수행한다.

line 33~34 : 자식 프로세스가 끝날 때 까지 기다린다. 자식 프로세스를 process_num개를 만들었으니 wait도 그만큼 수행한다.

line 35 : exit()를 통해 전달한 덧셈값을 얻기 위해 status에 8번 오른쪽으로 shift 한다.

line 36 : result_arr에 덧셈 값을 저장한다. 어떤 프로세스가 먼저 끝날지 모르기 때문에, 이것을 판단하는 방법은 pid – child_pid로 구했다. 자식 프로세스는 순서대로 생겨나고 id도 순서대로 부여받기 때문에 index를 구하는데 적합하다고 판단했다.

line 39 : my_fork() 함수를 호출한다. 계산하는데 필요한 프로세스의 개수는 현재 생성한 프로세스 개수의 절반이다.

[결과 분석]

MAX_PROCESS = 8

```
value of fork : 136
Time (Milli) : 1.785032
```

MAX_PROCESS = 64

```
value of fork : 64
Time (Milli) : 37.469055
```

반환값을 8 bit 오른쪽 shift 한 이유 :

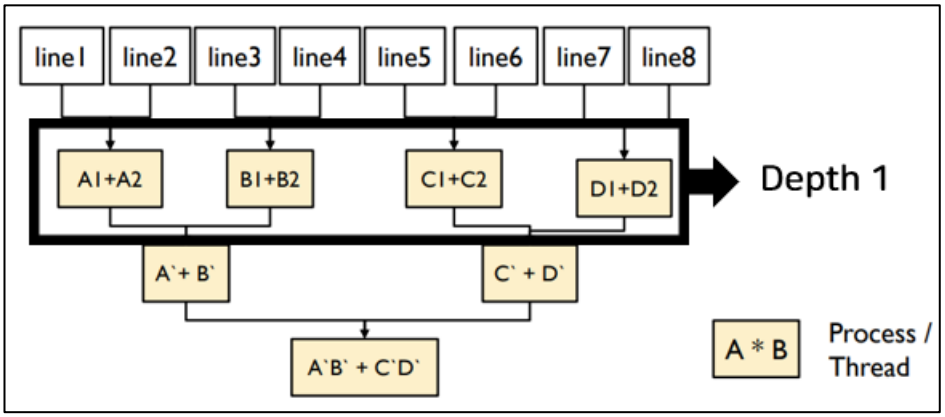
exit()에서 인수로 전달한 값은 부모 프로세스에서 wait(int * status) 함수를 통해 status에 저장된다. 정상 종료될 시 status의 하위 8 bit에는 0이 저장되고, 상위 8bit에는 exit()의 인수가 저장된다. 그래서 exit()의 인수를 받기 위해서 오른쪽으로 8만큼 shift를 수행한다. .

실제 값은 8256인데 64가 나온 이유 :

8256은 이진수로 1000000100000000이다. 이는 8 bit에 다 담을 수 없어서 잘리게 된다. wati()을 통해 얻은 status의 bit 상태는 01000000 00000000 일 것이다. 그래서 8296이 아닌 64가 나오게 되는 것이다.

[다중 스레드를 통한 구현 방법]

1. 파일은 Depth 1의 계산을 위해 MAX_PROCESS/2 번만 읽는다.



2. Depth 1의 계산 결과는 프로그램 내의 배열 result_arr[MAX_PROCESS]에 저장된다.
3. Depth 2부터는 파일에서 값을 읽어서 계산하는 것이 아니라, result_arr에서 읽어서 계산한다.

4. 재귀적으로 스레드를 만들고 값을 계산한다.
5. 계산하는 데에 필요한 스레드의 개수가 1이 되면 재귀함수를 빠져나와서 최종 결과 계산하고 출력한다.

[thread.c 코드 설명]

```
12 int result_arr[MAX_PROCESSES]={0};
```

line 12 : MAX_PROCESSES 크기의 정수형 배열을 선언한다. 이 배열은 스레드들이 같이 공유할 것이다.

- 메인 함수

```
95 int main(void)
96 {
97         struct timespec begin,end;
98
99         pthread_t tid[MAX_PROCESSES];
100
101        clock_gettime(CLOCK_MONOTONIC, &begin);
```

line 97 : 시간 측정을 위한 변수 선언

line 99 : MAX_PROCESSES만큼 스레드 배열 생성

line 101 : 시간을 측정하고 그 값을 begin 변수에 넣는다.

```
103        // first, you have to read number in file.
104        // and store our array.
105        for(int i=0;i<MAX_PROCESSES;i++){
106                pthread_create(&tid[i], NULL, file_read_store,(0+i));
107        }
108
```

line 105~107 : 처음에 MAX_PROCESSES개의 스레드들이 할 일은 파일에서 값을 읽고 depth1의 계산을 한 후, 공동 배열인 result_arr에 저장하는 것이다. 스레드를 만들고 file_read_store 함수를 수행하도록 한다. file_read_store의 인수로는 스레드의 인덱스 값을 주었다.

```
109        // wait thread terminates,
110        // result_arr will be initialized.
111        for(int i=0; i<MAX_PROCESSES;i++){
112                pthread_join(tid[i],NULL);
113        }
114
```

line 111 ~113 : 앞서 생성한 MAX_PROCESSES 개의 스레드들이 종료가 될 때 까지 기다린다. 스레드가 종료되면 result_arr에 값이 저장되어 있는 것을 확인할 수 있다.

```
os2018204058@ubuntu:~/practice3/assign301$ ./thread
result arr [0] : 3
result arr [1] : 7
result arr [2] : 11
result arr [3] : 15
result arr [4] : 19
result arr [5] : 23
result arr [6] : 27
result arr [7] : 31
```

line 113까지 수행한 결과

```
114
115         my_thread(MAX_PROCESSES/2);
```

my_thread 함수를 호출한다. my_thread에서는 스레드를 생성하고 값을 계산한다. my_thread의 인수로는 몇 개의 스레드를 만들지에 대한 값을 넣어준다. 지금은 MAX_PROCESSES/2만큼 생성하면 된다.

```
117         clock_gettime(CLOCK_MONOTONIC, &end);
118         long msec = end.tv_nsec - begin.tv_nsec;
119         if(msec <0)
120                 msec = 1000000000 + msec;
121
122         printf("value of thread : %d\n ", (result_arr[0]+result_arr[1]));
123         printf("Time (Milli) : %lf\n", (double) msec/1000000);
124         return 0;
125 }
```

끝나는 시간을 측정하고 출력한다.

- file_read_store 함수

```
18 void file_read_store(void *arg)
19 {
20         int index = (int)arg;
21
22         //file open
23         char *fname = "temp.txt";
24         FILE *file = fopen(fname,"r");
25
26         //store number
27         char first_line[5];
28         char second_line[5];
29
30
31         //for file pointer
32         for ( int k=0;k<(int)arg;k++){
33                 fgets(first_line,sizeof(first_line),file);
34                 fgets(first_line,sizeof(first_line),file);
35         }
36
```

line 20 : 스레드의 인덱스 번호를 저장한다

line 23 ~ 24 : 적절한 파일 이름을 넣고 파일을 연다

line 27 ~ 28 : 첫번째 숫자와 두번째 숫자를 저장할 변수

line 32 ~ 35 : 각 스레드가 적절한 숫자를 읽게 하기 위해 파일 포인터를 옮기기 위한 반복문이다. 5번째 프로세스이면 5번의 반복을 돈다. 이 반복문 안에서 두 줄씩 숫자를 읽는다. 이렇게 되면 5번 째 프로세스가 읽어야하는 적절한 파일 포인터에 도달하게 된다.

```
38          fgets(first_line,sizeof(first_line),file); // one line read
39          fgets(second_line,sizeof(second_line),file); // two line read
40
41          //string to int
42          int num1 = atoi(first_line);
43          int num2 = atoi(second_line);
44
45          //sum
46          int file_sum = num1+num2;
47
48          //close file
49          fclose(file);
50
51          //store sum  in array
52          result_arr[index] =file_sum;
53
54          //thread exit
55          pthread_exit(0);
56  }
```

line 38 ~ 39 : 적절한 파일 포인터에 도달하였으니, 여기서부터 fgets를 써서 첫번째 숫자와 두번째 숫자를 읽는다.

line 42 ~ 43 : atoi함수를 통해 string을 int형으로 바꾼다.

line 46 : 숫자들의 합을 file_sum에 저장한다.

line 49 : 파일을 닫는다

line 52 : 공동으로 쓰는 result_arr에 스레드 인덱스 번호에 맞게 숫자들의 합(file_sum)을 넣는다.

- my_thread 함수

```
69 void my_thread(int thread_number){
70          if(thread_number ==1){
71                  return;
72          }
73
```

line 70 ~ 73 : 필요한 스레드의 개수가 1이면 함수를 종료한다. 왜냐하면 메인 스레드에서 합을 구해서 출력하기만 하면 되기 때문. 멀티 스레딩을 통한 계산이 필요가 없어서 재귀함수 종료

```
74          //the deeper, we need process/2 threads.
75          pthread_t tid[thread_number];
76          int temp_result[thread_number];
```

line 75 ~ 76 : 스레드를 담을 tid 배열, 스레드들이 계산한 합의 결과를 저장할 temp_result 배열을 선언한다.

```
77          // thread create
78          for(int i=0; i<thread_number;i++){
79                  pthread_create(&tid[i], NULL, array_calculate, (0+i));
80          }
81          // wait thread terminates
82          for(int i=0;i<thread_number;i++){
83                  pthread_join(tid[i],&temp_result[i]);
84          }
```

line 78 ~ 80 : thread_number만큼 스레드를 만든다. 이 스레드들이 할 일은 array_calculate 함수에 정의되어 있다.

line 82 ~ 84 : 앞서 생성한 스레드들이 끝날 때 까지 기다린다. 스레드들이 array_caculate를 수행하고 난 다음 반환값이 temp_result의 "스레드 인덱스번호" 번째 에 저장된다.

```
86          // result_arr update
87          for(int i=0;i<thread_number;i++){
88                  result_arr[i] = temp_result[i];
89          }
90
91          my_thread(thread_number/2);
92  }
```

line 87~90 : temp_result에 담긴 값들을 result_arr로 옮긴다. result_arr에서 바로 계산할 수 없는 이유는 result_arr 배열에 동시에 접근하다가 이상한 값이 생길 수 있기 때문이다.

line 91 : 현재 필요한 스레드의 개수 /2 만큼 스레드를 생성하고 계산하게끔 my_thread를 호출한다.

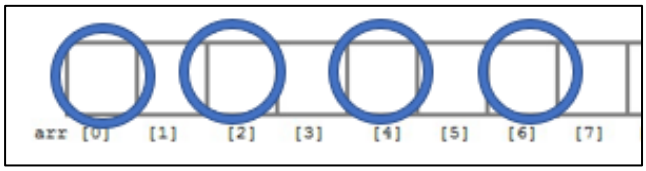
- array_calcaulate 함수

```
59 void* array_calculate(void *arg)
60 {
61          int sum;
62          int index = (int)arg;
63          int array_index = (int)arg *2;
64
65          sum  = result_arr[array_index]+result_arr[array_index+1];
66          return (void*) sum ;
67 }
```

line 61 : 숫자 두개의 합을 저장할 변수 sum

line 62 : thread의 인덱스

line 63 : thread의 인덱스 두배 (짝수) 를 해야 result_arr에 적절하게 접근하여 합을 구할 수 있다.



line 65 : 합을 구한다

line 66 : 합을 반환한다.

[결과 분석]

MAX_PROCESS = 8

```
os2018204058@ubuntu:~/practice3/assign301$ ./thread
value of thread : 136
 Time (Milli) : 2.185028
```

MAX_PROCESS = 64

```
os2018204058@ubuntu:~/practice3/assign301$ ./thread
value of thread : 8256
 Time (Milli) : 14.716276
```

fork()를 통해서 여러 개의 프로세스를 통해 계산한 결과와 여러 스레드를 통해 계산한 결과가 다르다. 프로세스는 exit()으로 상위 8bit에만 값을 넣을 수 있는 반면, 스레드는 변수의 주소값들을 반환하기 때문에 그런 제약이 없다. 그래서 올바른 계산값인 8256이 나왔다.

assignment 3-2

[Filegen.c 설명]

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define MAX_PROCESSES 10000

int main(void){

        char fileName[10];

        for (int i = 0; i<MAX_PROCESSES;i++)
        {
                char dir[20]="./temp/";
                sprintf(fileName,"%d",i);
                strcat(dir,fileName);

                FILE *f_write = fopen(dir,"w");
                fprintf(f_write, "%d", 1+rand()%9);
                fclose(f_write);

        }
        return 0;
}
```

fileName배열에는 1부터 9999까지의 값이 들어간다. 넉넉하게 10으로 사이즈를 설정해주었다. 그리

고 sprintf를 통해 문자를 스트링으로 바꾸어주었다. strcat으로 dir 배열과 filename 배열을 이어서 파일을 생성했다. 그 안에 랜덤 정수값을 작성하고 파일을 닫는다.

[코드 설명]

- 메인함수

```
int main(void)
{
        // SCHED_OTHER
        printf("##### SCHED_OTHER ##### \n");
        printf(" nice - * default * \n");
        my_fork(SCHED_OTHER,0);
        printf(" nice - * high * \n");
        my_fork(SCHED_OTHER,-20);
        printf(" nice - * low * \n");
        my_fork(SCHED_OTHER,19);
        // SCHED_FIFO
        printf("\n\n##### SCHED_FIFO ##### \n");
        printf(" priority - * default * \n");
        my_fork(SCHED_FIFO,50);
        printf(" priority - * high * \n");
        my_fork(SCHED_FIFO,99);
        printf(" priority - *  low * \n");
        my_fork(SCHED_FIFO,1);
        // SCHED_RR
        printf("\n\n##### SCHED_RR ##### \n");
        printf(" priority - * default * \n");
        my_fork(SCHED_RR,50);
        printf(" priority - * high * \n");
        my_fork(SCHED_RR,99);
        printf(" priority - * low *\n");
        my_fork(SCHED_RR,1);
```

```
void my_fork(int schedule, int priority);
```

메인 함수에서 my_fork 함수를 호출한다. my_fork는 자식 프로세스를 생성하고 스케줄링 정책을 설정하고 우선순위를 결정한다. 또 프로세스가 파일을 읽게 동작한다. my_fork의 매개변수로는 스케줄링 정책과 우선순위를 넣을 수 있다.

- my_fork 함수

```
49 void my_fork(int schedule, int priority){
50         struct timespec begin,end;
51         pid_t pid,child_pid;
52         int status;
53
54         clock_gettime(CLOCK_MONOTONIC, &begin);
```

line 50~51 : 시간을 측정하기위해 사용하는 변수 begin과 end를 선언한다.

line 51 : pid는 자식 프로세스인지, 부모 프로세스인지 판단하는 용도, child_pid는 부모 프로세스에서 wait()으로 자식 프로세스의 pid를 받을 때 저장하는 용도

line 52 : status는 wait()에 들어갈 인수로써 자식 프로세스가 exit()을 호출할 때의 값이 들어간다.

line 54 : 시작 시간을 begin에 저장한다.

```
55          for(int i=0; i<MAX_PROCESSES;i++){
56                  pid = fork();
57                  if(pid == -1){ printf("can't fork, error\n");exit(0);}
58                  if(pid == 0) { //success create child process
59                          if(schedule==0){
60                                  nice(priority);
61                          }
62
63                          else{
64                          struct sched_param p;
65
66                          p.sched_priority= priority;
67                          sched_setscheduler(0,schedule,&p);
68
69                          }
```

line 56 : fork()를 수행하고 반환값을 pid에 저장한다.

line 57 : pid가 -1이면 fork()를 정상적으로 수행하지 못했다는 의미이다. 종료한다.

line 58 : pid가 0이면 fork가 된 자식 프로세스란 의미이다. 이제 자식 프로세스가 할 일을 이 if문에 작성한다.

line 59~61 : 스케줄링 정책이 SCHED_OTHER(=0) 이면 nice 값을 설정한다. nice값은 실시간 스케줄링에서는 사용하지 않기 때문에 SCHED_OTHER일 때만 설정해준다. 스케줄링은 기본값이 SCHED_OTHER이기 때문에 따로 설정하지 않았다.

line 64~69 : 스케줄링 정책이 SCHED_FIFO나 SCHED_RR이면 우선순위를 설정해주고, sched_setscheduler()를 통해 스케줄링 정책을 설정한다. 인수로는 0을 준다. 0이면 현재 프로세스를 의미한다. 두번째 인수로는 스케줄링 정책, 세번째 인수로는 우선순위를 설정해놓은 sched_param 구조체 변수를 준다.

```
70                          char dir[20] = "temp/";
71                          char fname[10];
72                          sprintf(fname, "%d", i); // int -> string
73                          strcat(dir, fname); //dir+ fname ex) temp/1;
74
75                          FILE *file = fopen(dir,"r"); //file open
76                          char read_integer[10];
77                          fgets(read_integer,sizeof(read_integer),file); // number read
78                          //close file
79                          fclose(file);
80                          exit(0);
81                  }
82          }
```

line 70~73 : sprintf를 통해서 정수를 문자열로 바꾸어준다. strcat으로 디렉토리 경로와 파일 이름을 합쳐서 읽을 파일의 경로를 얻었다.

line 75~79 : 파일을 연다. fgets()를 통해 한 줄을 읽는다. 파일을 닫는다. 프로세스를 종료한다.

```
84          //wait until child process terminates
85          for(int i=0; i<MAX_PROCESSES;i++){
86          child_pid = wait(&status);
87          }
88
89          //end time
90          clock_gettime(CLOCK_MONOTONIC,&end);
91
92          long time;
93          if( (end.tv_sec - begin.tv_nsec)<0)
94                  time =   1000000000+end.tv_nsec - begin.tv_nsec;
95          printf("\t > time (Milli)    : %lf\n",(double) time/100000);
96  }
```

line 85~87 : 시간측정을 하기 위해서 fork한 프로세스들이 종료되길 기다린다.

line 90 : 종료한 시간을 end에 저장한다.

line 92~94 : 시작 시간에서 종료 시간을 빼서 걸린 시간을 구한다. end.tv_nsec에서 begin.tv_nsec을 빼는데, 이 값이 음수인 경우가 있다. 이런 경우는 end가 1000000000이 넘어가서 다시 리셋되어 처음부터 세는 경우이다. 그래서 이때는 end-begin에다가 1000000000를 더해준다.

[schedule 관련 함수]

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);

int sched_getscheduler(pid_t pid);
```

pid : 해당 프로세스 id (0인 경우엔 자신을 의미함)

policy : SCEHD_OTHERS, SCHED_FIFO, SCHED_RR

p : sched_param 구조체

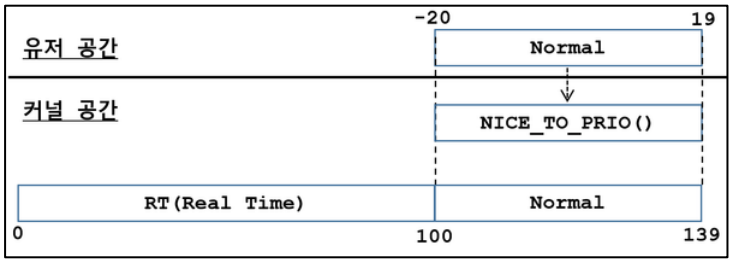
```
struct sched_param {
    ...
    int sched_priority;
    ...
};
```

p.sched_prority를 통해 실시간 우선순위를 설정한다. SCHED_OTHERS인 경우 0을 주어야하고, 실시간 프로세스인 경우 1-99 범위의 값을 준다. SCHED_FIFO와 SCHED_RR은 root권한으로 해야 사용할 수 있다.

결과 분석 – [SCHED_OTHER]

SCHED_OTHER은 일반 스케줄링 정책이다. 일반적인 사용자 프로세스에 적용되는 스케줄링 정책. 타임 슬라이스와 커널에 의해 지속적으로 변경되는 동적 우선순위 사용한다. sched_priority로 우선순위를 설정하지 않는다. nice()를 통해서 nice값을 설정해주어 우선순위를 결정할 수 있다. nice는 유저 공

간에서 설정한 프로세스 실행 우선순위를 뜻한다. -20~19의 값을 가지고, 커널공간에서는 100~139로 변환된다. 이는 실시간 프로세스 (RT Process)보다 우선순위가 무조건 낮다는 것을 뜻한다.




```
##### SCHED_OTHER #####
 nice,    high     : -20 *
 > time (Milli)    : 895.307631
```

nice 값을 가장 우선순위가 높은 -20을 줌

```
##### SCHED_OTHER #####
 nice, default :    0 *
 > time (Milli)    : 910.055652
```

nice 값을 default 값인 0을 줌

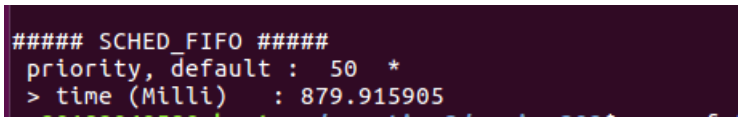
```
##### SCHED_OTHER #####
 nice,    low      :   19 *
 > time (Milli)    : 958.227585
```

nice 값을 low 값인 19를 줌

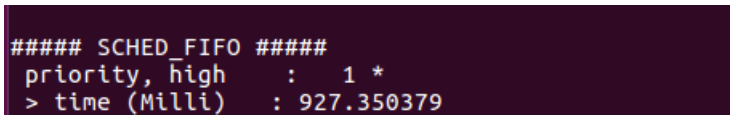
SCHED_OTHER 스케줄 정책을 사용하고 nice값을 default, high, low 값을 주었다. 프로세스의 nice값을 설정하고나서 파일의 입출력이 이루어지면 CPU를 포기하고 O(1)나 CFS 스케줄러에 의해 스케줄링 된다. 우선순위를 가장 높게 주었을 때 프로세스가 빨리 끝나고 가장 낮게 주었을 때 늦게 끝났다.

결과 분석 – [SCHED_FIFO]

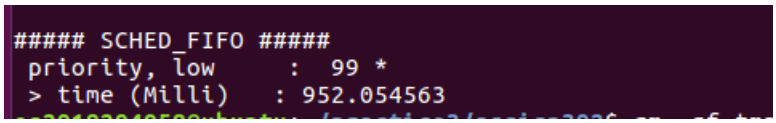
긴급한 실시간 프로세스를 위한 스케줄링 정책이다. 실시간을 보장 받아야할 경우 사용한다. 모든 SHCED_OTHERS 그룹보다 높은 고정 우선순위를 가지며 타임 슬라이스 개념이 없다. IFO 방식은 스스로가 yield를 취하거나 더 높은 priority를 가지는 프로세스에 의해 Interrupt될 때만 우선순위를 선점당한다. 이런 현상이 없으면 CPU를 계속 쓸 수 있다.



priority 값을 default 50을 줌



priority 값을 가장 우선순위가 높은 1을 줌

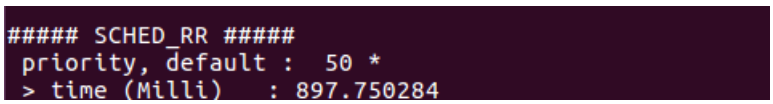


priority 값을 가장 우선순위가 낮은 99를 줌

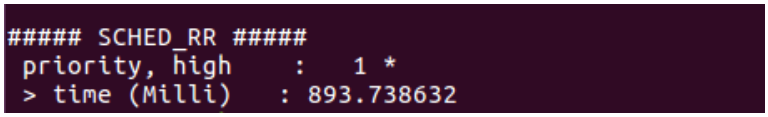
FIFO 방식일 때 우선순위가 제일 높으면 계속 CPU를 점유하고 있다가, 파일 입출력에 의해 CPU를 양도한다. 그리고 양도받은 프로세스가 CPU를 다시 놓으면 이 때 원래 프로세스가 다시 동작하게 된다. 그러니 이 방식일 때는 1번 째 프로세스 -> 2번 째 프로세스 -> 1번째 프로세스 이런식으로 다시 돌아올 것이다. 여러 프로세스를 조금씩 수행하는게 아니라 하나의 프로세스를 거의 끝내고 또 다음 프로세스를 마치는 식일 것이다.

결과 분석 – [SCHED_RR]

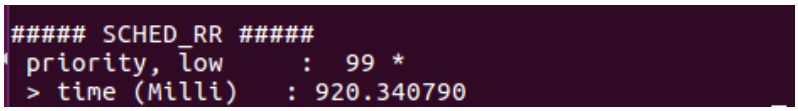
실시간 프로세스를 위한 스케줄링 정책이다. 실시간을 보장 받아야할 경우 사용한다. 모든 SCHED_OTHERS 그룹보다 높은 고정 우선순위를 가진다. RR방식은 같은 priority를 가지는 프로세스 간 time-slice를 통해 Round-Robin 방식을 취한다. 프로세스들이 자신에게 할당된 time을 소진하면 다른 프로세스에게 선점된다. time-slice가 다 소진되었을 때 스케줄링 큐의 맨 마지막으로 삽입된다.



priority 값을 default 50을 줌



priority 값을 가장 우선순위가 높은 1을 줌



priority 값을 가장 우선순위가 낮은 99를 줌

FIFO와 동일하게 동작하지만 time quantum을 소진하면 우선순위의 대기 큐로 돌아간다. 그러니깐 1번째 프로세스 동작하다가 파일 입출력 만나면 우선순위의 제일 뒤로 간다. 또 2번째 프로세스 동작하다가 파일 입출력 만나면 우선순위의 제일 뒤로 간다. 이런식으로 프로세스를 조금씩 조금씩 수행한다. 우선순위가 젤 높은게 빠르다.

assignment 3-3

[include/linux/sched.h 파일 수정]

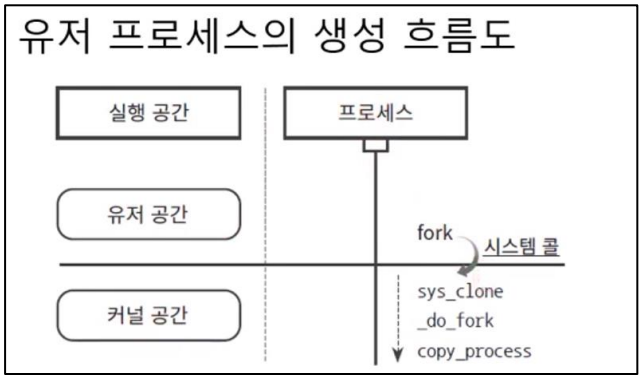
```
#endif

        /*
         * 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!
         */
};
```

task_struct의 끝. randomized_struct_fileds_end 위에 새로운 변수를 추가해야한다고 주석에 작성되어 있다. 그래서 저 주석 위에 fork()를 몇번 호출 했는지 저장하는 변수를 추가한다.

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

fork_call_count는 fork()를 몇번 호출했는지에 대한 정보가 저장될 것이다.

[linux/kernel/fork.c 파일 수정]



fork()를 실행하면 위의 그림과 같은 순서로 함수가 수행된다. 즉 fork.c 파일에서 _do_fork 함수에서 무언가를 수정해야한다.

- _do_fork() 함수

_do_fork()함수 실행 흐름 :

1) 프로세스 생성 : copy_process() 함수를 호출해서 프로세스를 생성한다. copy_process() 함수는 부모 프로세스의 리소스를 자식 프로세스에게 복제한다.

2) 생성한 프로세스를 실행 요청한다. wake_up_new_task() 함수를 호출해서 프로세스를 깨운다. 프로세스를 깨운다는 것은 스케줄러에게 프로세스의 실행 요청을 하는 것이다.

```
2171  */
2172 long _do_fork(unsigned long clone_flags,
2173               unsigned long stack_start,
2174               unsigned long stack_size,
2175               int __user *parent_tidptr,
2176               int __user *child_tidptr,
2177               unsigned long tls)
```

반환값은 long 타입으로 프로세스의 pid를 반환한다. 프로세스는 부모 프로세스의 주요 리소스등을 복제하는 방식으로 생성된다.

unsigned long stack_start : 유저 영역에서 스레드를 생성할 때 복사하려는 스택의 주소

unsigned long stack_size : 유저 영역에서 실행중인 스택의 크기

int __user *parent_tidptr, int __user *child_tidptr : 부모와 자식 스레드 그룹을 관리하는 핸들러 정보

```
2203         p = copy_process(clone_flags, stack_start, stack_size,
2204                          child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
2205
2206         add_latent_entropy();
2207
2208         if (IS_ERR(p))
2209                 return PTR_ERR(p);
2210
2211         current->fork_call_count=current->fork_call_count +1;
```

line 2203 : _do_fork함수에서 copy_process() 호출. copy_process의 역할 : 부모 프로세스의 메모리 및 시스템 정보를 자식 프로세스에게 복사한다.

line 2208~2209 : p 라는 변수에 오류가 있는지 검사. 오류가 있으면 함수 종료.

line 2211: 포크를 호출한 현재 프로세스는 fork를 한번 수행했으니 current -> fork_call_count ++를 한다.

- copy_process 함수

```
1653 static __latent_entropy struct task_struct *copy_process(
1654                                         unsigned long clone_flags,
1655                                         unsigned long stack_start,
1656                                         unsigned long stack_size,
1657                                         int __user *child_tidptr,
1658                                         struct pid *pid,
1659                                         int trace,
1660                                         unsigned long tls,
1661                                         int node)
```

```
1732         p = dup_task_struct(current, node);
1733         if (!p)
1734                 goto fork_out;
1735
1736         p->fork_call_count=0;
```

dup_task_sturct : 생성할 프로세스의 task_struct 구조체와 프로세스가 실행될 스택 공간을 할당 이 함수를 호출해서 태스크 디스크립터를 p에 저장한다

그러면 p는 생성한 자식 프로세스일 것 같다. 또 이 fork를 수행한 프로세스는 current로 접근할 수 있을 것이다. 그러니 p->fork_call_count =0으로 초기화 해준다.

[커널 컴파일]

커널 소스를 수정하였으면 커널을 컴파일 해야한다.

```
os2018204058@ubuntu:~/Downloads/linux-4.19.67$ sudo make -j8
```

[process_tracer.c 작성하기]

```
 9 #define __NR_ftrace 336
```

line 9 : 후킹 할 ftrace 시스템콜 번호를 정의한다.

- make_rw, make_ro

```
17 void make_rw(void *addr){
18         unsigned int level;
19         pte_t *pte = lookup_address((u64)addr, &level);
20
21         if(pte->pte &~ _PAGE_RW)
22                 pte->pte |= _PAGE_RW;
23 }
25 void make_ro(void *addr){
26         unsigned int level;
27         pte_t *pte = lookup_address((u64)addr, &level);
28
29         pte->pte = pte->pte &~ _PAGE_RW;
30 }
```

make_rw, make_ro는 addr이 속해 있는 페이지의 읽기 및 쓰기 권한을 부여하는 함수이다. 기본적으로, system call table은 쓰기 권한이 존재하지 않는다. make_rw를 호출하여 쓰기 권한이 없는 system

call table에 쓰기 권한을 부여하고 make_ro를 호출하여 쓰기 권한을 회수한다

- process_tracer 함수

```
32 static asmlinkage pid_t process_tracer(pid_t trace_task){
33
34         //print process name and pid
35         printk(KERN_INFO "##### TASK INFORMATION of '[%ld]' %s\n",current->pid, current->comm);
```

line 35 : current를 통해서 현재 프로세스의 pid와 이름을 출력한다.

```
37         //print task state
38         switch(current->state){
39                 case 0:
40                         printk(KERN_INFO "\t - task state : Running or ready\n");
41                         break;
42                 case 1:
43                         printk(KERN_INFO "\t - task state : Wait with ignoring all signals\n");
44                         break;
45                 case 2:
46                         printk(KERN_INFO "\t - task state : Wait \n");
47                         break;
48                 case 4:
49                         printk(KERN_INFO "\t - task state : Stopped\n");
50                         break;
51                 case 16:
52                         printk(KERN_INFO "\t - task state : Zombie process\n");
53                         break;
54                 case 128:
55                         printk(KERN_INFO "\t - task state : Dead\n");
56                         break;
57                 default:
58                         printk(KERN_INFO "\t - task state : .etc\n");
59         }
```

line 35 : current의 state 멤버를 통해서 상태를 출력한다.

```
/* Used in tsk->state: */
#define TASK_RUNNING                    0x00000000
#define TASK_INTERRUPTIBLE              0x00000001
#define TASK_UNINTERRUPTIBLE            0x00000002
#define __TASK_STOPPED                  0x00000004
#define __TASK_TRACED                   0x00000008
#define TASK_DEAD                       0x00000080
```

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

위의 링크를 참고하여 switch 문으로 분기했다.

```
60         //print Process Group Leader
61         printk(KERN_INFO "\t - Process Group Leader : [%d] %s\n",current->group_leader->pid, current->group_leader->comm);
62
63         //print context switch
64         printk(KERN_INFO "\t - Number of context-switch(es) %d\n",current->nivcsw);
65
66         //print number of calling fork()
67         printk(KERN_INFO "\t - Number of calling fork() : %d\n",current->fork_call_count);
```

line 61 : current의 group_leader를 참조해서 그 group_leader의 멤버인 pid를 출력한다. 또 comm도 참조해 이름도 출력한다.

line 63 : nivcsw 멤버를 통해 context swtich 횟수를 출력한다.

context switch 한 횟수를 저장하는 필드 : nivcsw

```
814         /* Context switch counts: */
815         unsigned long                   nvcsw;
816         unsigned long                   nivcsw;
```

line 64 : fork한 횟수를 세는 fork_call_count를 출력한다. 이 변수는 앞서 sched.h에서 새롭게 추가해 준 변수다.

```
69      //print parent process
70      struct task_struct *my_parent = current->parent;
71      pid_t parent_pid = my_parent -> pid;
72      printk(KERN_INFO "\t -its parent process : [%d] make\n", parent_pid);
73
```

line 70 : current->parent를 통해 현재 프로세스의 부모 프로세스 태스크 디스크립터에 접근할 수 있다. 이 것을 my_parent에 저장한다.

line 71 : parent_pid를 my_parent의 pid멤버에 접근하여 얻는다

line 72 : 부모 프로세스의 pid를 출력한다.

```
74      //print sibling process
75      //get  parent's child
76
77      //struct list_head *my_head_p;
78      struct task_struct *my_sibling;
79      bool only_child=true;
```

line 77 : 형제 프로세스의 태스크 디스크립터를 저장할 my_sibling을 선언한다.

line 79 : 이 프로세스가 형제가 없는지 판단하는 bool 변수를 선언한다.

```
81      printk(KERN_INFO "\t -its sibling process(es) : \n");
82
83      struct list_head *list;
84
85      list_for_each(list,&my_parent->children){
86              my_sibling = list_entry(list,struct task_struct,sibling);
87              if(my_sibling->pid == current->pid){
88                      only_child=true;
89              }
90              else{
91                      only_child=false;
92                      printk(KERN_INFO "\t > [%d] %s\n", my_sibling->pid, my_sibling->comm);
93              }
94      }
95
96      if(only_child)
97              printk(KERN_INFO "\t > It has no sibling.\n");
```

line 85 : 부모의 자식들을 순회한다.

line 86~94 : 부모의 자식 태스크 디스크립터를 my_sibling에 저장한다. 만약 이 자식의 pid가 현재 프로세스의 pid와 같다면 일단 외동인지 판단하는 only_child를 true로 설정한다. 그리고 부모의 다음 자식을 my_sibling에 저장한다. 이 때 현재 프로세스의 pid와 현재 자식프로세스의 pid가 다르다면 이것은 형제가 있다는 의미이고, only_child를 false로 하고 출력한다.

line 96 : 부모의 자식이 현재 프로세스밖에 없다면, 형제 프로세스가 없다는 의미이고 형제 프로세스가 없다는 말을 출력한다.

```
100         //print child process
101
102         printk(KERN_INFO "\t -its child process(es) : \n");
103         struct task_struct *temp_task;
104         //struct list_head *list;
105
106         if(list_empty(&current->children)==0){ //this means list is not empty
107                 list_for_each(list,&current->children){
108                         temp_task = list_entry(list,struct task_struct,sibling);
109                         printk(KERN_INFO "\t > [%d] %s\n", temp_task->pid, temp_task->comm);
110                 }
111         }
112         else{
113                 printk(KERN_INFO "\t > It has no child.\n");
114         }
115
116         printk(KERN_INFO "##### END OF INFORMATION #####\n");
117 }
```

line 106 ~ 114 : 현재 프로세스의 자식 프로세스 리스트가 비어있는지 먼저 판단한다. 비어있으면 자식이 없다는 말을 출력한다. 자식 프로세스 리스트가 비어 있지 않다면 리스트 순회를 통해 출력한다.

[모듈 삽입]

```
os2018204058@ubuntu:~/assign303$ sudo insmod process_tracer.ko
```

```
os2018204058@ubuntu:~/assign303$ sudo lsmod |grep trace
process_tracer         16384  0
```

[출력 테스트]

```
#include <linux/unistd.h>
#include <unistd.h>

#define __NR_ftrace 336

int main(void)
{
        syscall(__NR_ftrace, getpid());
        return 0;
}
```

test.c 파일을 만들어서 실행을 해보았다.

```
os2018204058@ubuntu:~/assign303$ ./test
```

```
[ 1618.911348] ##### TASK INFORMATION of '[3528]' test
[ 1618.911349]    - task state : Running or ready
[ 1618.911349]    - Process Group Leader : [3528] test
[ 1618.911350]    - Number of context-switch(es) 0
[ 1618.911350]    - Number of calling fork() : 0
[ 1618.911350]    -its parent process : [2406] make
[ 1618.911350]    -its sibling process(es) :
[ 1618.911351]    > It has no sibling.
[ 1618.911351]    -its child process(es) :
[ 1618.911351]    > It has no child.
[ 1618.911351] ##### END OF INFORMATION #####
```

출력 결과

```
[ 2089.181963] ##### TASK INFORMATION of '[4770]' fork
[ 2089.181964]   - task state : Running or ready
[ 2089.181965]   - Process Group Leader : [4770] fork
[ 2089.181965]   - Number of context-switch(es) : 0
[ 2089.181966]   - Number of calling fork() : 64
[ 2089.181966]   -its parent process : [2406] make
[ 2089.181967]   -its sibling process(es) :
[ 2089.181967]   > It has no sibling.
[ 2089.181967]   -its child process(es) :
[ 2089.181968]   > [4771] fork
[ 2089.181969]   > [4772] fork
[ 2089.181969]   > [4773] fork
[ 2089.181970]   > [4774] fork
[ 2089.181971]   > [4775] fork
[ 2089.181971]   > [4776] fork
[ 2089.181972]   > [4777] fork
[ 2089.181972]   > [4778] fork
[ 2089.181973]   > [4779] fork
[ 2089.181973]   > [4780] fork
```

assignment 3-1에서 만든 fork를 실행해보고 dmesg를 확인했다.

```
[ 2202.633492] ##### TASK INFORMATION of '[4918]' thread
[ 2202.633494]   - task state : Running or ready
[ 2202.633495]   - Process Group Leader : [4918] thread
[ 2202.633496]   - Number of context-switch(es) : 36
[ 2202.633497]   - Number of calling fork() : 126
[ 2202.633498]   -its parent process : [2406] make
[ 2202.633498]   -its sibling process(es) :
[ 2202.633499]   > It has no sibling.
[ 2202.633499]   -its child process(es) :
[ 2202.633500]   > It has no child.
[ 2202.633500] ##### END OF INFORMATION #####
```

assignment 3-1에서 만든 thread를 실행해보고 dmesg를 확인했다.

[부록]

list_for_each : 리스트를 순회한다. list_for_each(pos, head)

pos : 리스트의 현재 위치를 저장할 position. 타입은 리스트의 구조체를 가리키는 포인터야함. 배열의 인덱스라고 생각하자.

head : 연결리스트의 시작주소다.

list_for_each_entry : for 문처럼 각각의 node를 순서대로 접근하는 함수.

list_for_each_entry(pos, list, member) :

pos- 항목을 임시 저장해두는 변수.

head : 연결리스트의 시작주소

member : 링크드 리스트의 list_head 자료형 멤버변수,

리눅스 커널에서 자주 사용되는 함수로, 링크드 리스트를 이용하는 함수다. <linux/list.h>에서 찾을 수 있다.

고찰

[assignment 3-1]

process와 스레드의 차이를 알 수 있었다. 멀티프로세스를 통해 작업을 하면 반환값을 부모 프로세스에게 줄 때 한계가 있었다. 멀티 프로세스를 통해 작업을 하는 것은 생각보다 프로세스끼리 데이터 교환이 쉽지 않은 것 같다. 반면 멀티 스레드를 통해 작업을 하니 데이터 교환이 무척 쉬웠다. 그렇지만 공유 데이터를 접근할 때 조심해야할 것 같다. 스레드를 통한 파일에서 숫자를 읽어서 합을 구하는 코드를 짤 때, 공유 데이터를 한번에 접근하려고 하니 이상한 값이 나왔었다. 멀티 스레드 방식을 채택하려면 항상 데이터 관리와 스레드가 어떤 흐름을 가지는지 알아야 할 것 같다.

[assignment 3-2]

스케줄링 정책을 결정할 때, 운영체제가 결정하는 것으로 알았다. 그러나 이번 과제를 수행하면서 스케줄링을 결정하는 것은 프로세스가 한다는 것을 깨달았다. 스케줄링과 우선순위를 결정하는 것은 사용자가 함부로 사용하면 안되는 것 같았는데, 리눅스 커널을 이용하고 sudo 권한을 통하니 간단한 함수를 통해 스케줄링 정책을 결정할 수 있는 것이 무척 신기했다. 또 FIFO와 RR을 이론 수업에서만 들었을 땐 그렇구나... 하고 넘어갔다. 이번 과제를 수행하면서 직접 코드를 짜고 결과를 확인해보았다. 왜 이런 결과가 나왔는지 곰곰히 생각해볼 수 있는 기회가 되어서 무척 뜻깊었다.

[assignment 3-3]

fork() 함수가 어떤 절차를 걸쳐서 수행되는지 알 수 있었다. fork()가 실행되면 sys_clone() 시스템콜이 호출되서 거기서 _do_fork()함수를 호출한다. do_fork 함수를 통해 부모프로세스에서 자식프로세스를 복사한다. 너무너무 신기했다. 또 내가 프로세스의 정보를 막 추가하고 더할 수 있었다. 특정 프로세스를 수행하면 컴퓨터를 종료하거나 그런 코드도 짤 수 있을 것 같다. 정말 리눅스 커널은 확장성이 최고란 것을 깨달았다.

[Reference]

https://kldp.org/node/138145

https://jhnyang.tistory.com/314

https://hand-over.tistory.com/57

https://pragp.tistory.com/entry/pthread%EC%97%90-%EC%97%AC%EB%9F%AC-%EC%9D%B8%EC%9E%90-%EC%A0%84%EB%8B%AC%ED%95%98%EA%B8%B0

http://mwultong.blogspot.com/2007/08/c-exit.html

https://palpit.tistory.com/entry/Linux-3-%EB%A6%AC%EB%88%85%EC%8A%A4-

%EC%8A%A4%EC%BC%80%EC%A4%84%EB%A7%81

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-priorities_and_policies

https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=ssi5719&logNo=220047644298

https://austindhkim.tistory.com/35

https://wariua.github.io/man-pages-ko/sched_getparam(2)/

https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=ssi5719&logNo=220047644298

http://egloos.zum.com/rousalome/v/10015948

https://elixir.bootlin.com/linux/v4.19.264/source/include/linux/sched.h