

운영체제

Assignment3

한준호

2018741035

최상호 교수님

Assignment 3-1

- Introduction

'temp.txt' 라는 텍스트 파일을 생성하고 MAX_PROCESSES 값의 두 배 만큼의 양수 값을 기록하는 숫자 생성기인 'numgen.c'을 만든다. fork() 함수나 pthread_create() 함수를 이용해 MAX_PROCESSES 만큼의 프로세스/스레드를 생성한다.

'numgen.c' 의 기록된 숫자를 이용하여 생선한 프로세스/스레드 마다 2개의 숫자를 읽고, 잃어온 두 숫자를 더한 후, exit() 함수를 이용해 부모 프로세스/스레드에게 값을 전달한다. 최종적으로 나온 값을 결과로 출력한다. 이 때, clock_gettime() 함수를 이용해 전체 프로그램의 수행 시간을 측정한다.

- Conclusion & Analysis

• 'numgen.c'

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

#define MAX_PROCESSES 8

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

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

    fclose(f_write);
}

~
~
~
"numgen.c" 20L, 285C 1,1 All
```

모드(mode)	스트림의 성격	파일 없으면
r	읽기가능	에러
w	쓰기가능	생성
a	파일끝 덧붙여 쓰기가능	생성
r+	읽기/쓰기 가능	에러
w+	읽기/쓰기 가능	생성
a+	읽기/덧붙여 쓰기 가능	생성

fopen 함수를 이용하여 생성한 'temp.txt' 텍스트 파일에 fprintf() 함수를 이용하여 MAX_PROCESSES 값의 두 배 만큼의 양수 값 기록하고 기록을 완료한 파일을 덮어썼다.

```
os2018741035@ubuntu:~/assign_3/3-1$ vi temp.txt
```

```
"temp.txt" 8L, 16C      1,1      All
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
~
~
~
~
~
"temp.txt" 16L, 39C
1,1 All
```

운영체제

```
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
128,1 Bot
```

MAX_PROCESSES = 64 -> 1~128 기록

- 'fork.c' (완벽하게 구현을 하지 못했기 때문에 과정을 첨부합니다.)

1. 큐 구조의 배열 이용 - 'fork2.c'

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

#define MAX_PROCESSES 4

int MAX = MAX_PROCESSES*2;
int front = -1;
int rear = -1;
int queue[MAX_PROCESSES*2];

unsigned char get_index(unsigned char num);
int IsEmpty(void);
int IsFull(void);
void addq(int value);
int deleteq();

void main(void)
{
    pid_t pid[MAX_PROCESSES];
    pid_t wpid;
    int status = 0;
    int i;
    char line[255];
    int f_num;
    int result;
    struct timespec begin, end;
    f_num = get_index(MAX_PROCESSES);

    FILE *fp = fopen("temp.txt", "a+");

    while(fgets(line, sizeof(line), fp) != NULL)
    {
        rear = rear + 1;
        queue[rear] = atoi(line);
    }

    for(i=0; i<f_num+1; i++)
    {
        pid[i] = fork();
    }
}
```

```
unsigned char get_index(unsigned char num)
{
    char i = 0;
    while(1)
    {
        if(num==1)
            return i;
        i++;

        if( (num%2) == 0)
            num = num/2;
        else
        {
            printf("ERROR \n");
            return 0;
        }
    }
}

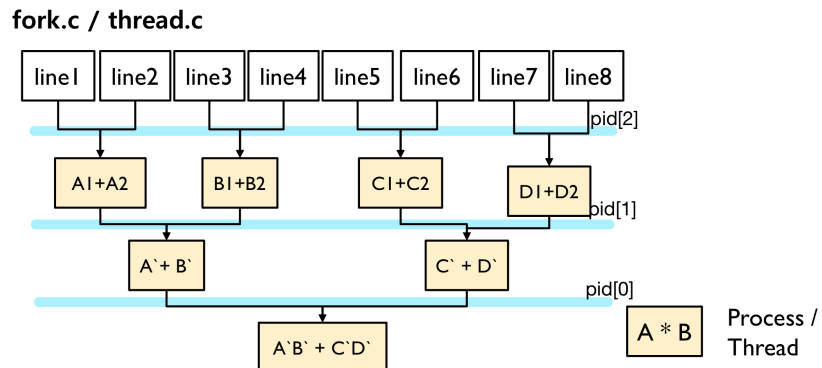
int IsEmpty(void)
{
    if(front==rear)
        return 1;
    else
        return 0;
}

int IsFull(void){
    int tmp = (rear+1)%MAX;
    if(tmp == front)
        return 1;
    else
        return 0;
}

void addq(int value)
{
    if(IsFull())
        printf("Queue if Full.\n");
    else
    {
        rear = rear+1;
        queue[rear] = value;
    }
}

int deleteq()
{
    if(IsEmpty())
        printf("Queue is Empty.\n");
    else
    {
        int del_num = queue[0];
        for(int i =0; i<MAX-1; i++)
            queue[i] = queue[i+1];
        queue[rear] = 0;
        front =0;
        rear -=1;
        return del_num;
    }
}
```

과제를 수행하면서 여러 시도 끝에 큐 구조의 배열을 이용하여 다중 프로세스로 수행하는 프로그램을 만들었다. 우선 'temp.txt'에 저장된 값 모두를 한 줄씩 읽어와 atoi() 함수를 이용해 읽어온 문자를 숫자로 변환해 queue라는 배열에 차례로 저장해주었다.



우선 한번 fork()를 할 때마다 부모 프로세스와 자식 프로세스로 2개로 나뉘지기 때문에 MAX_PROCESSES 만큼 프로세스가 만들어지기 위해서는

위와 같이 MAX_PROCESSES = 4 일 때는

$$2^x = \text{MAX_PROCESSES} = 4$$

$$x = 2$$

$$\text{fork() 수} = x + 1 = 3$$

같은 방법으로 MAX_PROCESSES = 8 일 때는

$$2^x = \text{MAX_PROCESSES} = 8$$

$$x = 3$$

$$\text{fork() 수} = x + 1 = 4$$

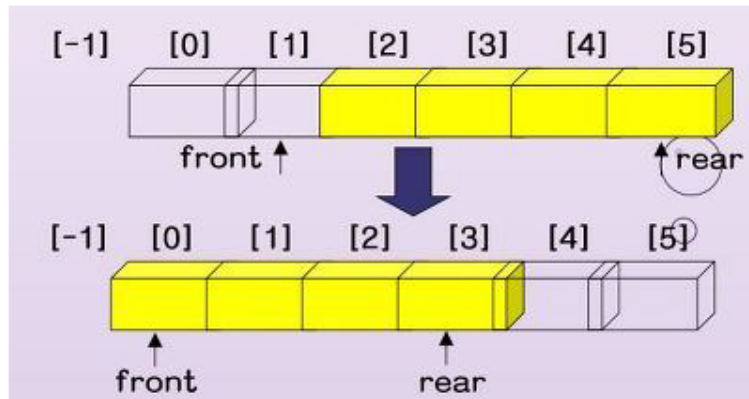
MAX_PROCESSES = 64 일 때는

$$2^x = \text{MAX_PROCESSES} = 64$$

$$x = 6$$

$$\text{fork() 수} = x + 1 = 7$$

라는 규칙으로 fork()를 해주기 위해, $2^x = \text{MAX_PROCESSES}$ 의 해를 구하는 함수인 get_index라는 함수를 구현해주었고 그 해를 f_num이라는 변수에 저장해주었다. for 문을 이용해서 f_num+1 만큼 fork()를 해줄 수 있도록 하였다.



다음으로는, 큐 구조의 배열을 구현하였다.

배열에 값을 추가하여 저장할 때는 배열 끝에 값을 저장하고 배열의 저장된 값 중 유효한 값의 끝을 알려주는 rear라는 변수의 값을 +1 해주는 함수인 addq()라는 함수를 구현하였다.

배열에 값을 빼서 사용할 때는 배열에 [0]번 째 값을 반환하고 배열을 전체적으로 위 그림처럼 전체적으로 왼쪽으로 이동시키고 배열의 저장된 값 중 유효한 값의 끝을 알려주는 rear라는 변수의 값을 -1 해주는 함수인 deleteq()라는 함수를 구현하였다.

```
clock_gettime(CLOCK_MONOTONIC, &begin);
for(i=0; i<f_num+1;i++)
{
    if(pid[f_num-i] == 0)
    {
        int n1 = deleteq();
        int n2 = deleteq();
        printf("n1: %d, n2: %d\n", n1, n2);
        result = n1+n2;
        addq(result);

        exit(status);
    }
    else if(pid[f_num] >0)
    {
        while((wpid = wait(&status))>0);
    }
}
clock_gettime(CLOCK_MONOTONIC, &end);

fclose(fp);

for(int k=0; k<MAX; k++)
    printf("%d ", queue[k]);
printf("\n");

long time = (end.tv_sec - begin.tv_sec) + (end.tv_nsec
- begin.tv_nsec);
printf("value of fork : %d\n", queue[0]);
printf("%lf\n", (double)time/1000000000);
}
```

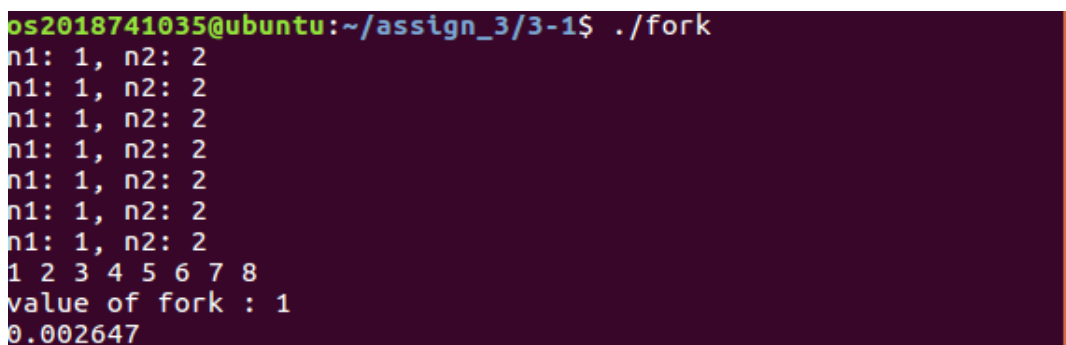
전체 프로그램의 수행시간을 측정하기 위해 'begin', 'end'라는 <time.h>의 timespec 구조체를 선언해주었고 clock_gettime() 함수를 이용해 전체 프로그램의 시작 시간과 끝 시간을 저장해주었다. 저장된 값의 차를 이용해서 전체 프로그램의 수행시간을 초 단위로 출력할 수 있도록 하였다.

MAX_PROCESSES = 4 일 때, pid[0] -> pid[1] -> pid[2] 순으로 fork()를 하였기 때문에 가장 마지막(최상단) 자식 프로세스부터 실행되어야 하기 위해 for문을 이용해서 가장 마지막 pid[]의 자식 프로세스부터 실행될 수 있도록 하였다.

pid[] = 0 일 때, 즉 자식 프로세스는 위에서 만들어둔 deleteq() 함수를 이용해 2개의 숫자를 읽어왔고 이 값들은 더한 후, addq() 함수를 이용해 queue 배열에 더한 값을 넣어주었다. 자식 프로세스에서 모든 수행이 끝나면 exit() 함수와 status 변수를 통해 현재 자식 프로세스의 상태를 부모 프로세스에게 전달해주었다. pid[] > 0 일 때, 즉 부모 프로세스는 while문 내부에서 전달된 status 변수를 wait() 함수를 실행해 모든 자식 프로세스가 종료될 때까지 대기하도록 하였다.

이 과정을 반복하여 pid[0]의 자식 프로세스와 부모 프로세스까지 수행을 마치면 최종적인 값이 queue[0]에 저장되어 있을 것이고 이 값을 출력할 수 있을 것이라고 생각했다.

-> 결과



```
os2018741035@ubuntu:~/assign_3/3-1$ ./fork
n1: 1, n2: 2
n1: 1, n2: 2
n1: 1, n2: 2
n1: 1, n2: 2
n1: 1, n2: 2
n1: 1, n2: 2
n1: 1, n2: 2
n1: 1, n2: 2
1 2 3 4 5 6 7 8
value of fork : 1
0.002647
```

하지만 결과를 확인해본 결과, 같은 pid[]의 모든 자식 프로세스가 동시에 배열의 2개의 숫자 읽기 때문에 Critical Section 문제, 즉 동시성 문제가 발생했고 구현한 큐 구조의 배열을 예상했던 것처럼 사용할 수 없었다. 결과적으로 최종적인 queue 배열에도 변화가 없었다. 수업 시간에 배운 lock 함수를 이용해 프로세스들을 동기화해 Critical Section 문제를 해결해보려고 시도해봤지만 성공하지 못했다.

2. fopen(), fseek(), fgets() 함수의 파일 포인터 이용 - 'fork.c'

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

#define MAX_PROCESSES 4

unsigned char get_index(unsigned char num);

void main(void)
{
    pid_t pid[MAX_PROCESSES];
    int i;
    int j = 0;
    int f_num = 0;
    char line[255];
    int result;

    struct timespec begin, end;

    f_num = get_index(MAX_PROCESSES);

    FILE *fp = fopen("temp.txt", "a+");
    FILE *fp0 = fopen("temp0.txt", "a+");

    for(i=0; i<f_num+1; i++)
    {
        pid[i] = fork();
    }
```

```
unsigned char get_index(unsigned char num)
{
    char i = 0;
    while(i)
    {
        if(num==1)
            return i;
        i++;
        if( (num%2) == 0)
            num = num/2;
        else
        {
            printf("ERROR \n");
            return 0;
        }
    }
}
```

125,1 Bot

1 방법과 동일한 방법으로 fork() 함수를 이용하여 MAX_PROCESSES 만큼 프로세스를 생성해주었다.

```
clock_gettime(CLOCK_MONOTONIC, &begin);

if (pid[f_num] == 0)
{
    fseek(fp, 0, SEEK_CUR);
    fgets(line, sizeof(line), fp);
    int n1 = atoi(line);

    fseek(fp, 0, SEEK_CUR);
    fgets(line, sizeof(line), fp);
    int n2 = atoi(line);

    result = n1 + n2;

    exit(result);
}
else if(pid[f_num] > 0)
{
    wait(&result);
    fprintf(fp0, "%d\n", result>>8);

    fclose(fp);
    fclose(fp0);
}
```


1 방법과 동일하게 전체 프로그램의 수행 시간을 측정하기 위해 clock_gettime() 함수를 사용하였다.

1 방법인, 큐 구조의 배열을 이용하였을 때 같은 pid[]의 모든 자식 프로세스가 동시에 배열의 2개의 숫자 읽기 때문에 Critical Section 문제, 즉 동시성 문제가 발생했고 구현한 큐 구조의 배열을 예상했던 것처럼 사용할 수 없었다. 결과적으로 최종적인 queue 배열에도 변화가 없었다. lock 함수를 이용해 프로세스들을 동기화해 Critical Section 문제를 해결해보려고 시도해봤지만 성공하지 못했다.

따라서, Critical Section 문제를 해결하기 위해 fopen(), fseek(), fgets() 함수를 이용해서 같은 pid[]의 모든 자식 프로세스가 동시에 같은 파일을 읽더라도 읽히는 파일의 파일 포인터는 하나라는 점을 이용했다.

가장 마지막(최상단) 자식 프로세스는 fopen() 함수를 이용해 MAX_PROCESSES*2 만큼의 수가 기록된 'temp.txt'를 열었다. fseek() 함수와 fgets() 함수를 이용해 'temp.txt'를 한 줄씩, 한 줄 씩 총 두줄을 읽어와서 atoi() 함수를 이용해 읽어온 문자를 숫자로 변환 하였고 각각을 n1, n2 변수에 저장했다.

SEEK_SET
파일의 시작
SEEK_CUR
파일 포인터의 현재 위치
SEEK_END
파일의 끝

이 때, fseek()의 flag를 SEEK_CUR로 설정하여 다수의 자식 프로세스가 동시에 한 파일을 읽더라도 파일의 포인터는 파일에 하나라는 점을 이용하여 한 자식 프로세스가 2개의 숫자를 읽어왔을 때 다른 자식 프로세스는 숫자를 읽어온 파일 포인터의 현재 위치에서 파일 포인터를 한줄 더 이동해 다음 줄을 읽어오도록 하였다.

n1, n2에 저장된 2개의 숫자를 더한 후, result 변수에 그 값을 저장하였고, exit() 함수를 이용하여 자식 프로세스가 종료될 때 저장된 result 값을 부모 프로세스에게 전달하였다. 부모 프로세스에서는 wait() 함수를 이용해 자식 프로세스가 종료될 때까지 대기하도록 하였다.

부모 프로세스로 전달된 result 값을 'temp0.txt' 라는 새로운 파일에 기록하였다. 이 때, wait로 자식 프로세스의 status를 받아올 때, 최하위 바이트(8비트)에 시그널 번호가 저장되기 때문에 종료값 * 256으로 부모 프로세스로 넘어온다. 그렇기 때문에 부모 프로세스에서 전달된 값을 사용하기 위해서는 (result)>>8로, 즉 2의 8승으로 나눠주어서 사용했다.

fopen의 모드를 'a+'로 하였기 때문에 'temp.txt' 파일에 뒷 부분에 result 값을 덧붙여 쓰기 할 수도 있었지만, 덧붙여 쓰기를 한다면 파일의 포인터가 파일의 끝으로 이동하기 때문에 각각의 자식 프로세스가 'temp.txt'에 모든 숫자를 읽어오지 못하는 문제가 발생했다. 그래서 'temp0.txt'이라는 새로운 텍스트 파일을 생성하고 그 파일에 결과 값을 저장해주었다.

가장 마지막(최상단)의 부모 프로세스까지 종료된다면 숫자를 읽어오기 위한 'temp.txt' 와 결과를 저장하기 위한 'temp0.txt'를 닫아주었다.

```

for(i=0; i<f_num; i++)
{
    char file_in[100];
    char file_out[100];
    sprintf(file_in, "temp%d.txt", i);
    sprintf(file_out, "temp%d.txt", i+1);

    FILE *fp_in = fopen(file_in, "a+");
    FILE *fp_out = fopen(file_out, "a+");

    if (pid[f_num-i-1] == 0)
    {
        fseek(fp_in, 0, SEEK_CUR);
        fgets(line, sizeof(line), fp_in);
        int n1 = atoi(line);

        fseek(fp_in, 0, SEEK_CUR);
        fgets(line, sizeof(line), fp_in);
        int n2 = atoi(line);

        result = n1 + n2;

        exit(result);
    }
    else if(pid[f_num-i-1] > 0)
    {
        wait(&result);
        fprintf(fp_out, "%d\n", result>>8);

        fclose(fp_in);
        fclose(fp_out);
    }
}

clock_gettime(CLOCK_MONOTONIC, &end);

char file_final[100];
sprintf(file_final, "temp%d.txt", f_num);
FILE *fp_final = fopen(file_final, "r+");
fseek(fp_final, 0, SEEK_CUR);
fgets(line, sizeof(line), fp_final);
int final = atoi(line);

long time = (end.tv_sec - begin.tv_sec) + (end.tv_nsec - begin.tv_nsec);
printf("value of fork : %d\n", final);
printf("%lf\n", (double)time/1000000000);
}

```

(ex) pid[1] 자식 프로세스: 'temp0.txt' 에서 값 두개씩 읽어와 더함 -> pid[1] 부모 프로세스: 'temp1.txt'에 자식 프로세스로부터 전달된 값 저장)

-> 결과

- ```
os2018741035@ubuntu:~/assign_3/3-1$./numgen
os2018741035@ubuntu:~/assign_3/3-1$./fork
value of fork : 36
0.001694
os2018741035@ubuntu:~/assign_3/3-1$ ls
fork fork.z~ numgen.c temp2.txt thread.c
fork.c Makefile temp0.txt temp.txt thread.c~
fork.c~ numgen temp1.txt thread
```

```
1
2
3
4
5
6
7
8
~
~
~
~
"temp.txt" 8L 16C 1.1 All
```

```
3
7
11
15
~
~
~
~
~
~
~
~
~
"temp0.txt" 4L 10C 1.1 All
```

[illegible][illegible]

## 운영체제

- MAX\_PROCESSES = 8

```
os2018741035@ubuntu:~/assign_3/3-1$./numgen
os2018741035@ubuntu:~/assign_3/3-1$./fork
value of fork : 72
0.002401
os2018741035@ubuntu:~/assign_3/3-1$ ls
fork fork.z~ numgen.c temp2.txt thread
fork.c Makefile temp0.txt temp3.txt thread.c
fork.c~ numgen temp1.txt temp.txt thread.c~
```

'temp.txt' -> 'temp0.txt' -> 'temp1.txt' -> 'temp2.txt' -> 'temp3.txt'

- MAX\_PROCESSES = 64

```
os2018741035@ubuntu:~/assign_3/3-1$./numgen
os2018741035@ubuntu:~/assign_3/3-1$./fork
value of fork : 63
0.018340
os2018741035@ubuntu:~/assign_3/3-1$ ls
fork fork.z~ numgen.c temp2.txt temp5.txt thread
fork.c Makefile temp0.txt temp3.txt temp6.txt thread.c
fork.c~ numgen temp1.txt temp4.txt temp.txt thread.c~
```

'temp.txt' -> 'temp0.txt' -> 'temp1.txt' -> 'temp2.txt' -> 'temp3.txt' ->  
'temp4.txt' -> 'temp5.txt' -> 'temp6.txt'

파일의 포인터를 이용한 방법을 이용하였을 때는 MAX\_PROCESSES = 4 때에는 전체 프로세스의 개수가 비교적 적기 때문에 몇 번에 시도 끝에 원하는 최종값을 출력할 수 있었다. 하지만 MAX\_PROCESSES = 4 일 때도, 프로세서 꽤 많기 때문에 파일의 포인터가 순차적으로 한 줄씩 이동하지 않고 포인터가 튀는 경우가 많았고, 여전히 Critical Section 문제가 있었다. 결과적으로 실행할 때마다 다른 최종값을 출력하는 문제가 있었다.

MAX\_PROCESSES 값을 크게 설정할 수록 파일의 포인터가 순차적으로 이동하는 빈도가 극히 드물었고 이로 인해 예시와 같은 결과값을 얻기는 힘들었다.

• 'thread.c'

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

#define MAX_PROCESSES 4

int MAX = MAX_PROCESSES*2;
int front = 0;
int rear = -1;
int queue[MAX_PROCESSES*2];

pthread_mutex_t lock;
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

void* thread_func(void *arg);

int IsEmpty(void);
int IsFull(void);
void addq(int value);
int deleteq();

int main()
{
 char line[255];
 int result;
 struct timespec begin, end;

 FILE *fp = fopen("temp.txt", "r+");

 while(fgets(line, sizeof(line), fp) != NULL)
 {
 rear = rear + 1;
 queue[rear] = atoi(line);
 }

 for(int k=0; k<MAX; k++)
 printf("%d ", queue[k]);
 printf("\n");

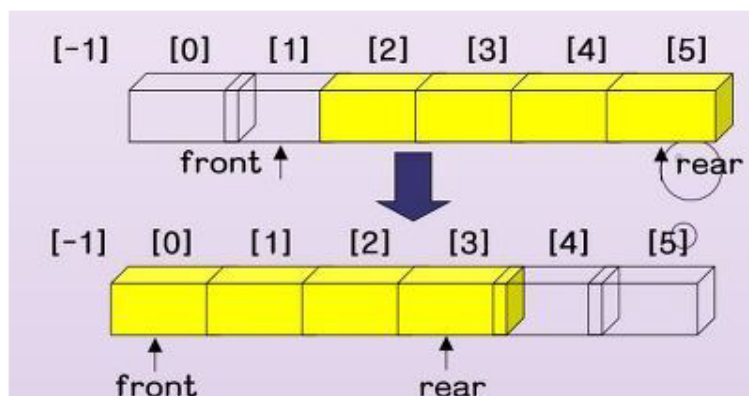
 int IsEmpty(void)
 {
 if(front==rear)
 return 1;
 else
 return 0;
 }

 int IsFull(void){
 int tmp = rear+1;
 if(tmp == front)
 return 1;
 else
 return 0;
 }

 void addq(int value)
 {
 if(IsFull())
 printf("Queue is Full.\n");
 else
 {
 rear = rear+1;
 queue[rear] = value;
 }
 }

 int deleteq()
 {
 if(IsEmpty())
 printf("Queue is Empty.\n");
 else
 {
 int del_num = queue[0];
 for(int i =0; i<MAX-1; i++)
 queue[i] = queue[i+1];
 queue[rear] = 0;
 front =0;
 rear -=1;
 return del_num;
 }
 }
}
```

큐 구조의 배열을 이용하여 다중 스레드로 수행하는 프로그램을 만들었다. 우선 'temp.txt'에 저장된 값 모두를 한 줄씩 읽어와 atoi() 함수를 이용해 읽어온 문자를 숫자로 변환해 queue라는 배열에 차례로 저장해주었다.



다음으로는, 큐 구조의 배열을 구현하였다.

배열에 값을 추가하여 저장할 때는 배열 끝에 값을 저장하고 배열의 저장된 값 중 유효한 값의 끝을 알려주는 rear라는 변수의 값을 +1 해주는 함수인 addq()라는 함수를 구현하였다.

배열에 값을 빼서 사용할 때는 배열에 [0]번 째 값을 반환하고 배열을 전체적으로 위 그림처럼 전체적으로 왼쪽으로 이동시키고 배열의 저장된 값 중 유효한 값의 끝을 알려주는 rear라는 변수의 값을 -1 해주는 함수인 deleteq()라는 함수를 구현하였다.

Critical Section 문제, 즉 동시성 문제가 발생하지 않도록 <pthread.h>의 lock 함수를 이용하기 위해 pthread\_mutex\_t 라는 자료형의 lock 변수를 선언하고 pthread\_mutex\_lock() 함수와 pthread\_mutex\_unlock 함수를 선언했다.

```
void* thread_func(void *arg)
{
 pthread_mutex_lock(&lock);

 int result;
 int n1 = deleteq();
 int n2 = deleteq();
 result = n1+n2;
 addq(result);

 pthread_mutex_unlock(&lock);
}
```

MAX\_PROCESSES 만큼 스레드를 생성하고 생성한 스레드에서 사용하는 함수인 thread\_func을 정의해주었다.

위에서 만들어둔 deleteq() 함수를 이용해 2개의 숫자를 읽어왔고 이 값들은 더한 후, addq() 함수를 이용해 queue 배열에 더한 값을 넣어주었다.

이 때, 위에서 선언한 pthread\_mutex\_lock() 함수와 pthread\_mutex\_unlock 함수를 이용해 Critical Section 문제, 즉 동시성 문제가 발생하지 않도록 동기화 해주었다.

```

int main()
{
 char line[255];
 int result;
 struct timespec begin, end;

 FILE *fp = fopen("temp.txt", "r+");

 while(fgets(line, sizeof(line), fp) != NULL)
 {
 rear= rear +1;
 queue[rear] = atoi(line);
 }

 for(int k=0; k<MAX; k++)
 printf("%d ", queue[k]);
 printf("\n");

 clock_gettime(CLOCK_MONOTONIC, &begin);

 pthread_t tid[MAX_PROCESSES];

 for(int j=MAX_PROCESSES; j>1; j=j/2)
 {
 for(int i=0; i<j; i++)
 {
 pthread_create(&tid[i], NULL, thread_func, (void*)0);
 pthread_join(tid[i], NULL);
 }
 for(int k=0; k<MAX; k++)
 printf("%d ", queue[k]);
 printf("\n");
 }

 clock_gettime(CLOCK_MONOTONIC, &end);

 long time = (end.tv_sec - begin.tv_sec) + (end.tv_nsec - begin.tv_nsec);
 printf("value of fork : %d\n", queue[0]+queue[1]);
 printf("%lf\n", (double)time/1000000000);

 return 0;
}

```

전체 프로그램의 수행시간을 측정하기 위해 'begin', 'end'라는 <time.h>의 timespec 구조체를 선언해주었고 clock\_gettime() 함수를 이용해 전체 프로그램의 시작 시간과 끝 시간을 저장해주었다. 저장된 값의 차를 이용해서 전체 프로그램의 수행시간을 초 단위로 출력할 수 있도록 하였다.

pthread\_create() 함수를 이용해 MAX\_PROCESSES 만큼 스레드를 생성했다. POSIX thread: Creation은 사용자가 지정한 특정 함수(thread\_func)를 호출함으로써 시작한다.

pthread\_join() 함수를 이용해 생성된 thread가 pthread\_join()을 호출한 thread에게 반환값을 전달하고 종료될 수 있도록 하였다. POSIX thread: Join은 지정된 thread가 종료될 때까지 호출 thread의 수행을 중단한다.

최종적으로 queue[0] 값과 queue[1] 값을 더해 출력될 수 있도록 하였다.

- MAX\_PROCESSES = 4

```
os2018741035@ubuntu:~/assign_3/3-1$./numgen
os2018741035@ubuntu:~/assign_3/3-1$./thread
1 2 3 4 5 6 7 8
3 7 11 15 0 0 0 0
10 26 0 0 0 0 0 0
value of fork : 36
0.000987
```

- MAX\_PROCESSES = 8

```
os2018741035@ubuntu:~/assign_3/3-1$./numgen
os2018741035@ubuntu:~/assign_3/3-1$./thread
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
3 7 11 15 19 23 27 31 0 0 0 0 0 0 0 0
10 26 42 58 0 0 0 0 0 0 0 0 0 0 0 0
36 100 0 0 0 0 0 0 0 0 0 0 0 0 0 0
value of fork : 136
0.003770
```

- MAX\_PROCESSES = 64

```
os2018741035@ubuntu:~/assign_3/3-1$./numgen
os2018741035@ubuntu:~/assign_3/3-1$./thread
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 4
5 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 10
5 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128
3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63 67 71 75 79 83 8
7 91 95 99 103 107 111 115 119 123 127 131 135 139 143 147 151
155 159 163 167 171 175 179 183 187 191 195 199 203 207 211 2
15 219 223 227 231 235 239 243 247 251 255 0 0 0 0 0 0 0 0 0 0
0 0
0 0
10 26 42 58 74 90 106 122 138 154 170 186 202 218 234 250 266
282 298 314 330 346 362 378 394 410 426 442 458 474 490 506 0
0 0
0 0
0 0
0 0
36 100 164 228 292 356 420 484 548 612 676 740 804 868 932 996
0 0
0 0
0 0
0 0
136 392 648 904 1160 1416 1672 1928 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0
0 0
0 0
528 1552 2576 3600 0
0 0
0 0
0 0
0 0
2080 6176 0
0 0
0 0
0 0
0 0 0 0 0 0 0
value of fork : 8256
0.023090
```



- 'Makefile'

```
all: numgen fork thread

numgen: numgen.c
 gcc numgen.c -o numgen

fork: fork.c
 gcc fork.c -o fork

thread: thread.c
 gcc thread.c -o thread -pthread

~
~
~
"Makefile" 10L, 151C 1,1 All

os2018741035@ubuntu:~/assign_3/3-1$ ls
fork fork.c~ Makefile numgen.c thread thread.c~
fork.c fork.z~ numgen temp.txt thread.c
```

numgen.c, fork.c, thread.c을 모두 컴파일 하도록 작성하였다. thread.c는 pthread를 사용하기 위해 '-pthread' flag를 추가해주었다.

- 결과 분석

< MAX\_PROCESSES = 4 >

```
os2018741035@ubuntu:~/assign_3/3-1$./fork
value of fork : 36
0.001694
os2018741035@ubuntu:~/assign_3/3-1$./thread
value of fork : 36
0.002507
```

< MAX\_PROCESSES = 8 >

```
os2018741035@ubuntu:~/assign_3/3-1$./fork
value of fork : 72
0.002401
os2018741035@ubuntu:~/assign_3/3-1$./thread
value of fork : 136
0.003533
```

< MAX\_PROCESSES = 64 >

```
os2018741035@ubuntu:~/assign_3/3-1$./fork
value of fork : 112
0.020620
os2018741035@ubuntu:~/assign_3/3-1$./thread
value of fork : 8256
0.032646
```

fork() 함수를 사용해서 다중 프로세스를 만들어서 각 프로세스마다 2개의 숫자를 읽어 오는 프로그램의 방식으로는 파일의 포인터를 사용하는 방식을 사용했고, pthread\_create() 함수를 사용해서 다중 스레드를 만들어서 각 스레드마다 2개의 숫자를 읽어오는 프로그램의 방식으로는 큐 구조의 배열을 사용했기 때문에 두 프로그램에서 숫자를 받아오고 사용하는 방식을 다르게 구현했기 때문에 다중 스레드를 이용하는 프로그램이 다중 프로세스를 이용하는 프로그램보다 빠르지 않았다.

큐 구조의 배열을 이용한 다중 스레드 프로그램은 pthread\_mutex\_lock() 함수와 pthread\_mutex\_unlock 함수를 이용해 Critical Section 문제, 즉 동시성 문제가 발생하지 않도록 동기화 해주었기 때문에 스레드가 순차적으로 작동하고 이로 인해 속도가 상대적으로 느려졌다. 이로 인해 구현한 다중 스레드 프로그램은 다중 스레드 사용의 장점을 완벽히 사용하지 못했다.

### - 고찰

두 가지 방식으로 다중 프로세스 또는 다중 스레드 프로그램을 구현해보면서 수 많은 시행 착오를 겪었다. 이 과정에서 lock 함수없이 Critical Section 문제, 즉 동시성 문제 없이 과제 필요조건에 맞는 프로그램을 구현하기는 무척 어려웠다. 또 처음 사용해 보는 함수가 많았고 처음 배우는 개념을 과제 내용에 적용하고 구현하는 과정에서 어려움을 느꼈다. 구현을 제대로 했다고 생각했지만 막상 실행을 해보면 다중 프로세스와 다중 스레드의 작동 특성 상, 예상하지 못한 결과가 나오는 경우가 많았다.

다중 스레드를 구현하는 과정에서는 lock 함수를 이용해 Critical Section 문제는 해결하였지만 속도가 느려졌고, 이로 인해 스레드 사용의 장점을 완벽하게 사용하지 못하는 문제가 발생했다. 따라서 동시성 문제를 해결하면서 과제의 필요조건에 맞는 프로그램을 구현하기는 매우 어려웠다.

또 다중 프로세스 프로그램과 다중 스레드 프로그램에서 두 개의 숫자를 받아오는 방식을 다르게 구현해주었기 때문에 두 프로그램에 결과를 비교하기는 어려웠다.

### - Reference

- c언어 자식 프로세스가 종료될 때까지 대기 함수 wait(), <https://badayak.com/entry/C언어-자식-프로세스가-종료될-때까지-대기-함수-wait>
- [c언어] 파일 읽기 쓰기(read, write) 추가하기 모드 종류 (wt, rb, a+t), <https://ansan-survivor.tistory.com/1302>
- fseek() - fseeko() — 파일 위치 변경, <https://www.ibm.com/docs/ko/i/7.4?topic=functions-fseek-fseeko-reposition-file-position>

## Assignment 3-2

### - Introduction

'temp'라는 디렉토리를 만들고, 생성한 디렉토리에 무작위의 integer형 양수( $\leq 9$ )가 기록되어 있는 생성한 프로세스(MAX\_PROCESSES)만큼의 '(i).txt'(i = 0~MAX\_PROCESSES) 라는 파일을 만드는 파일 생성기인 'filegen.c'를 만든다.

fork() 함수를 이용해 MAX\_PROCESSES 만큼 프로세스를 생성하고 각 프로세스에서 Linux에서 지원하는 3가지 CPU 스케줄링 정책 변경해보고 Priority, Nice의 값을 각각 highest, default, lowest로 설정해보면서 각 프로세스가 미리 만들어져 있는 i번째 파일에서 integer 데이터를 읽는 성능을 비교해본다.

### - Conclusion & Analysis

#### • 'filegen.c'

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

#define MAX_PROCESSES 8192

int main()
{
 int i;
 char filename[10];

 for(i=0; i<MAX_PROCESSES; i++)
 {
 sprintf(filename, "./temp/%d.txt", i);
 FILE *f_write = fopen(filename, "w+");
 fprintf(f_write, "%d\n", 1+rand()%9);
 fclose(f_write);
 }
}
```

"filegen.c" 21L, 355C 1,1 All

파일의 이름을 저장할 변수인 'filename'이라는 char형 변수를 선언했다.

MAX\_PROCESSES 만큼의 텍스트 파일을 생성하기 위해 for문을 사용했다

for문에 i 값(0~MAX\_PROCESSES)에 따라 생성한 'temp' 디렉토리에 텍스트 파일을 저장하기 위해 sprintf() 함수를 이용해 filename 변수에 저장했다.

fopen 함수를 이용하여 저장한 filename이라는 이름의 텍스트 파일을 생성한다. 이 때, fopen 함수의 모드를 'w+'로 설정하였다. 'w+' 모드는 읽기/쓰기가 가능하고 만약 읽거나 쓰려고 하는 대상 파일이 없다면 파일을 생성하고 대상 파일이 있다면 파일을

덮어쓴다. fopen 함수를 이용하여 생성한 텍스트 파일에 fprintf() 함수를 이용하여 무작위의 integer형 양수 값( $\leq 9$ )을 기록하고 기록을 완료한 파일을 덮어썼다.

```
os2018741035@ubuntu:~/assign_3/3-2/temp$ ls -t
8192.txt 7211.txt 6284.txt 4815.txt 4500.txt 3073.txt 2245.txt 1016.txt
7956.txt 7212.txt 6285.txt 4816.txt 4501.txt 3074.txt 2246.txt 1017.txt
7957.txt 7213.txt 6286.txt 4817.txt 4502.txt 3075.txt 2247.txt 1000.txt
7958.txt 7214.txt 6287.txt 4818.txt 4503.txt 3076.txt 2248.txt 1001.txt
7959.txt 7215.txt 6288.txt 4819.txt 4073.txt 3048.txt 2249.txt 1002.txt
7960.txt 7216.txt 6289.txt 4820.txt 4074.txt 3049.txt 2250.txt 1003.txt
7961.txt 7217.txt 6290.txt 4821.txt 4075.txt 3050.txt 2251.txt 1004.txt
7962.txt 7218.txt 6291.txt 4822.txt 4076.txt 3051.txt 2252.txt 1005.txt
7963.txt 7219.txt 6292.txt 4823.txt 4077.txt 3052.txt 2253.txt 1006.txt
7964.txt 7220.txt 6293.txt 4824.txt 4078.txt 3053.txt 2254.txt 1007.txt
7965.txt 7221.txt 6294.txt 4825.txt 4079.txt 3054.txt 2255.txt 1008.txt
7966.txt 7222.txt 6295.txt 4826.txt 4080.txt 3055.txt 2256.txt 1009.txt
7967.txt 7223.txt 6296.txt 4827.txt 4081.txt 3056.txt 2257.txt 1010.txt
```

#### • 'schedtest.c'

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>
#include <time.h>
#include <sched.h>

#define MAX_PROCESSES 8192

#define MAX_PRIORITY 99
#define MIN_PRIORITY 1
#define DEFAULT_PRIORITY 30
#define OTHER_PRIORITY 0

unsigned int get_index(unsigned int num);
unsigned int get_order(int num);

//SCHED_OTHER : The standard round-robin time-sharing policy
//SCHED_FIFO : A first-in first-out policy
//SCHED_RR : A round-robin policy

unsigned int get_index(unsigned int num)
{
 int i = 0;
 while(1)
 {
 if(num==1)
 return i;
 i++;

 if((num%2) == 0)
 num = num/2;
 else
 {
 printf("ERROR \n");
 return 0;
 }
 }
}

unsigned int get_order(int num)
{
 int order = 1;
 for(int i = num; i>0; i--)
 order = order*2;
 return order;
}
```

99,1 Bot

우선 성능을 비교할 수 있을 정도의 프로세스를 생성하기 위해 MAX\_PROCESSES 값을  $2^{13} = 8192$ 로 설정하였다. MAX\_PROCESSES 만큼의 프로세스를 생성하기 위해 3-1 과제와 마찬가지로 get\_index() 함수를 구현하였다. 추가적으로, i번째 파일에 접근하기 위해  $2^x$  값을 반환하는 get\_order() 함수를 구현하였다.

Linux에서 지원하는 3가지 CPU 스케줄링 3가지는

1. The standard round-robin time-sharing policy : SCHED\_OTHER

-> 프로세스의 우선 순위 값이 범위가 0값 이므로

highest = default = lowest = OTHER\_PRIORITY = 0

2. A first-in first-out policy : SCHED\_FIFO

-> 프로세스의 우선 순위 값의 범위가 1~99 이므로

highest = MAX\_PRIORITY = 99

default = DEFAULT\_PRIORITY = 30

lowest = MIN\_PRIORITY = 1

3. A round-robin policy : SCHED\_RR

-> 프로세스의 우선 순위 값의 범위가 1~99 이므로

highest = MAX\_PRIORITY = 99

default = DEFAULT\_PRIORITY = 30

lowest = MIN\_PRIORITY = 1

로 설정해주었다.

```

void main(void)
{
 struct sched_param param0, paramF, paramR;

 param0.sched_priority = OTHER_PRIORITY;
 paramF.sched_priority = DEFAULT_PRIORITY;
 paramR.sched_priority = DEFAULT_PRIORITY;

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

 pid_t pid[MAX_PROCESSES];
 int i;
 int result;
 int status;
 int f_num;
 char line[255];

 struct timespec begin, end;

 f_num = get_index(MAX_PROCESSES);

 for(i=0; i<f_num+1; i++)
 {
 pid[i] = fork();
 }

 clock_gettime(CLOCK_MONOTONIC, &begin);
 for(i=0; i<f_num+1; i++)
 {
 char filename[100];
 sprintf(filename, "./temp/%d.txt", get_order(i));
 FILE *fp = fopen(filename, "a+");

 if(pid[f_num-i] == 0)
 {
 sched_setscheduler(pid[f_num-i], SCHED_OTHER, ¶m0);
 //sched_setscheduler(pid[f_num-i], SCHED_FIFO, ¶mF);
 //sched_setscheduler(pid[f_num-i], SCHED_RR, ¶mR);

 fseek(fp, 0, SEEK_SET);
 fgets(line, sizeof(line), fp);
 result = atoi(line);
 exit(status);
 }
 fclose(fp);
 }
 clock_gettime(CLOCK_MONOTONIC, &end);
 long time = (end.tv_sec - begin.tv_sec) + (end.tv_nsec - begin.tv_nsec);
 printf("%lf\n", (double)time/1000000000);
}

```

3-1 과제와 마찬가지로 fork() 함수를 사용해 MAX\_PROCESSES 만큼 프로세스를 생성하였다.

sched\_setscheduler() 함수를 이용해 각 프로세스에서 CPU 스케줄링 정책을 변경하였고, sched\_param 구조체를 선언하고 그 값을 설정하여 프로세스의 우선순위 값을 설정하였다.

변경한 CPU 스케줄링 정책과 우선 순위에 따라 각 프로세스가 미리 만들어져 있는 i번째 파일에서 integer 데이터를 읽을 수 있도록 했다.

성능을 비교하기 위해 전체 프로그램의 수행시간을 측정하려고 'begin', 'end'라는 <time.h>의 timespec 구조체를 선언해주었고 clock\_gettime() 함수를 이용해 전체 프로그램의 시작 시간과 끝 시간을 저장해주었다. 저장된 값의 차를 이용해서 전체 프로그램의 수행시간을 초 단위로 출력할 수 있도록 하였다.

- 'Makefile'

```
all: filegen schedtest

filegen: filegen.c
 gcc filegen.c -o filegen

schedtest: schedtest.c
 gcc schedtest.c -o schedtest

~
~
"Makefile" 8L, 124C 1,1 All

os2018741035@ubuntu:~/assign_3/3-2$ ls
filegen Makefile schedtest.c temp
filegen.c schedtest schedtest.c~
```

filegen.c, schedtest.c을 모두 컴파일 하도록 작성하였다.

• 결과 분석

< SCHED\_OTHER, OTHER\_PRIORITY >

```
os2018741035@ubuntu:~/assign_3/3-2$./schedtest
0.000863
```

< SCHED\_FIFO, DEFAULT\_PRIORITY(30: default) >

```
os2018741035@ubuntu:~/assign_3/3-2$./schedtest
0.000477
```

< SCHED\_FIFO, MAX\_PRIORITY(99: highest) >

```
os2018741035@ubuntu:~/assign_3/3-2$./schedtest
0.000373
```

< SCHED\_FIFO, MIN\_PRIORITY(1: lowest) >

```
os2018741035@ubuntu:~/assign_3/3-2$./schedtest
0.000222
```

< SCHED\_RR, DEFAULT\_PRIORITY(30: default) >

```
os2018741035@ubuntu:~/assign_3/3-2$./schedtest
0.000310
```

< SCHED\_RR, MAX\_PRIORITY(99: highest) >

```
os2018741035@ubuntu:~/assign_3/3-2$./schedtest
0.000180
```

< SCHED\_RR, MIN\_PRIORITY(1: lowest) >

```
os2018741035@ubuntu:~/assign_3/3-2$./schedtest
0.000501
```

SCHED\_OTHER 보다 SCHED\_FIFO와 SCHED\_RR 프로세스 스케줄링이 대체로 빠른 속도 프로세스를 완료했다.

SCHED\_FIFO 스케줄링에서는 MIN\_PRIORITY(1: lowest)의 우선 순위에서 속도가 가장 빨랐다.

SCHED\_RR 스케줄링에서는 MAX\_PRIORITY(99: highest)의 우선 순위에서 속도가 가장 빨랐다.



## - 고찰

스케줄링 개념을 학습하고 과제 내용에 적용하고 구현 과정에서 sched\_setscheduler() 함수 사용법을 익힐 수 있었다.

변경한 CPU 스케줄링 정책과 Priority, Nice 값을 변경해보면서 각 경우마다 과제 3-1에서 학습한 clock\_gettime() 함수를 사용해 전체 프로그램의 수행 시간을 측정하는 방식으로 성능을 비교했는데, 오직 수행 시간으로만 성능을 비교 했기 때문에 무슨 CPU 스케줄링 정책이 확실히 좋은 스케줄링 정책인지, 어떤 우선 순위가 가장 좋은 우선 순위인지 판단하기는 어려웠고 설정한 값에 따라 CPU 스케줄링 정책과 Priority, Nice 값이 제대로 변경 되었는지 확인하기도 어려웠다.

각 경우 모두 수행 시간이 모두 충분히 작아서 비교하기 어려움이 있었는데 이는 충분한 양의 프로세스를 생성하지 않았기 때문이라고 예상된다.

CPU 스케줄링 정책과 Priority 값을 변경하면서 각 경우에 따라 성능을 비교해보았지만, Nice을 다루는 방법은 충분히 학습하지 못해 Nice 값까지 변경하면서 각 경우의 성능을 비교하지는 못했다.

## - Reference

- 120. [RTOS] 3편 : C 언어로 프로세스 우선순위 및 스케줄링 정책 변경하기 (chrt, nice), [https://m.blog.naver.com/alice\\_k106/221170316769](https://m.blog.naver.com/alice_k106/221170316769)
- sched\_setscheduler, <https://noel-embedded.tistory.com/464>

## Assignment 3-3

### - Introduction

PID를 바탕으로 아래와 같은 프로세스의 정보를 출력하는 Module 작성

- (1) 프로세스 이름
- (2) 현재 프로세스의 상태
- (3) 프로세스 그룹 정보
- (4) 해당 프로세스를 실행하기 위해 수행된 context switch 횟수
- (5) fork()를 호출한 횟수
- (6) 부모(parent) 프로세스 정보
- (7) 형제자매(sibling) 프로세스 정보
- (8) 자식(child) 프로세스 정보

### - Conclusion & Analysis (완벽히 구현하지 못함)

#### • 'sched.h'

```
/* Used in tsk->state */
#define TASK_RUNNING 0x00000
#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 0x00200
#define TASK_NOLOAD 0x0400
#define TASK_NEW 0x0800
#define TASK_STATE_MAX 0x1000

"sched.h" 28L, 541C
```

태스크의 실행 상태를 저장하는 멤버로, 이 상태에 대해서는 같은 sched.h 헤더파일에 매크로로 정의되어있다.

- 'fork.c'

```
for(i=0; i<f_num+1; i++)
{
 pid[i] = fork();
 f_count++;
}
```

과제 3-1과 마찬가지로 'fork.c'를 구현해주었고, fork() 함수를 사용해 MAX\_PROCESSES 만큼 프로세스를 생성할 때 fork() 호출될 때마다 f\_count 변수의 값을 1씩 증가시켜주었다.

- 'process\_tracer.c'

```
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h> /* kallsyms_lookup_name() */
#include <linux/syscalls.h> /* __SYSCALL_DFINRX() */
#include <asm/syscall_wrapper.h> /* __SYSCALL_DFINRX() */
#include <unistd.h>

#include "sched.h"

#define __NR_ftrace 336

void **syscall_table;

unsigned int pid_n = 0;

asmlinkage int(*real_ftrace)(pid_t);
asmlinkage int process_tracer(pid_t pid)
{
 struct task_struct *task = current;
 pid_n = task->pid;

 printk(KERN_INFO "##### TASK INFORMATION of '['%d] systemd' #####\n",
pid_t getsid(pid_t pid));

 struct context {
 int eip;
 int esp;
 int ebx;
 int ecx;
 int edx;
 int esi;
 int edi;
 int ebp;
 };

 enum proc_state {Running or ready, Wait with ignoring all signals, Wait
, Stopped, Zombie Process, Dead, etc};

 struct proc {
 char *mem;
 uint sz;
 char *kstack;

 enum proc_state state;
 int pid;
 struct proc *parent;
 void *chan;
 int killed;
 struct file *ofile[NOFILE];
 struct inode *cwd;
 struct context context;
 struct trapframe *tf;
 };
};
```

```

 printk("- task state : %d", task->state);
 printk("- Process Group Leader : %s", pid_t getsid(pid_t pid));
 printk("- Number of context switches : %d",);
 printk("- Number of calling fork() : %d", f_count);
 printk("- it's parent process : [%d] swapper/0", pid_t getppid(void));

 printk("7");
 printk("8");

 return (real_ftrace(pid));
}

void make_rw(void *addr)
{
 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)
{
 unsigned int level;
 pte_t *pte = lookup_address((u64)addr, &level);

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

static int __init process_tracer__init(void)
{
 syscall_table = (void**) kallsyms_lookup_name("sys_call_table");

 make_rw(syscall_table);
 real_ftrace = syscall_table[__NR_ftrace];
 syscall_table[__NR_ftrace] = process_tracer;

 return 0;
}

static void __exit process_tracer__exit(void)
{
 printk(KERN_INFO "##### END OF INFORMATION #####\n")

 syscall_table[__NR_ftrace] = real_ftrace;

 /* Recover the page's permission (i.e. read-only)*/
 make_ro(syscall_table);
}

module_init(process_tracer_init);
module_exit(process_tracer_exit);
MODULE_LICENSE("GPL");

```

106,1

Bot

2차 과제에서 만든 ftrace 시스템콜을 hijack 하여 'process\_tracer' 함수로 대체하기 위해 'process\_tracer.c' 를 작성하였고 'asm linkage' 를 사용하였다. 모듈이 추가될 때 system call table에 읽기 및 쓰기 권한을 부여하고, 모듈이 제거될 때 읽기 및 쓰기 권한을 회수할 수 있도록 함수를 작성하였다.

모듈이 적재 될 때, task\_struct 구조체를 이용해 현재 프로세스의 pid를 'pid\_n' 변수에 저장하고 저장한 PID를 바탕으로 프로세스를 정보를 커널 메시지로 출력할 수 있도록 Module를 작성하였다. 모듈 해제 시 기존의 시스템콜을 원상 복구 하기 위해 기존 ftrace 시스템콜 주소를 'real\_ftrace' 에 저장하였고, 후킹할 'process\_tracer' 시스템콜을 기존의 ftrace 시스템콜 대신 대체하였다.

모듈이 해제 될 때, T종료를 알수있도록 커널 메시지를 출력하도록 하였다. 모듈을 적재 할 때 저장해뒀던 'real\_ftrace'를 이용해 시스템콜을 원상 복구할 수 있도록 하였다.

+) )

- task\_struct 구조체는 커널 메모리 영역 내에 존재하며, 프로세스의 메모리맵, 파일 디스크립터, 프로세스의 권한 등의 정보를 저장한다.

- 프로세스를 식별할 때 사용할 수 있는 PID, 프로세스 그룹, 세션 관련 함수

| 기능            | 함수원형                                              |
|---------------|---------------------------------------------------|
| PID 검색        | pid_t getpid(void);                               |
| 부모 PID 검색     | pid_t getppid(void);                              |
| 프로세스 그룹 ID 검색 | pid_t getpgrp(void);<br>pid_t getpgid(pid_t pid); |
| 프로세스 그룹 ID 변경 | int setpgid(pid_t pid, pid_t pgid);               |
| 세션 리더 ID 검색   | pid_t getsid(pid_t pid);                          |
| 세션 생성         | pid_t setsid(void);                               |

- Proc Structure

```
// the registers x% will save and restore
// to stop and subsequently restart a process
struct context {
 int eip; // Index pointer register
 int esp; // Stack pointer register
 int ebx; // Called the base register
 int ecx; // Called the counter register
 int edx; // Called the data register
 int esi; // Source index register
 int edi; // Destination index register
 int ebp; // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
 RUNNABLE, RUNNING, ZOMBIE };
```

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
 char *mem; // Start of process memory
 uint sz; // Size of process memory
 char *kstack; // Bottom of kernel stack
 // for this process
 enum proc_state state; // Process state
 int pid; // Process ID
 struct proc *parent; // Parent process
 void *chan; // If non-zero, sleeping on chan
 int killed; // If non-zero, have been killed
 struct file *ofile[NOFILE]; // Open files
 struct inode *cwd; // Current directory
 struct context context; // Switch here to run process
 struct trapframe *tf; // Trap frame for the
 // current interrupt
};
```

## • 'Makefile'

```
obj-m := process_tracer.o

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

all:
 $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
 $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean

~
~
~
"Makefile" 11L, 191C 1,1 All
```

## - 고찰

과제 2의 내용과 예전에 배운 내용을 복습하면서 과제 3-3와 관련된 내용을 학습 했지만 과제 내용에 적용하고 구현하는 과정에서 어려움을 느꼈고 결국 과제를 완벽히 구현하지 못했다. 이번 과제 3-3은 완벽히 구현하지 못했지만 과제 2의 Hooking 관련 내용이나 커널 모듈을 추가하고 제거하는 방법에 대한 과제 2에서 이해가 부족했던 내용에 대해서 추가적으로 학습할 수 있었다.

PID를 바탕으로 프로세스의 정보를 출력하기 위해서 현재 프로세스에서 PID를 받아왔지만 받아온 해당 PID에 대한 프로세스의 정보를 찾고 출력하는 것은 어려웠다.

## - Reference

- [시스템프로그래밍] 프로세스 정보, <https://12bme.tistory.com/222>
- [ Linux Kernel ] task\_struct structure, <https://aidencom.tistory.com/272>