운영체제 실습

[Assignment #3]

Class : 목3

Professor : 최상호 교수님

Student ID : 2021202045

Name : 김예은

Introduction.

본 3차 과제는 2차 과제에서 만든 336번 system call table에 만든 후킹 함수 및 모듈을 이용하여 task_struct 구조체를 통해 target pid에 대한 정보를 출력하는 trace code 부분(3-1), temp.txt 파일에 나열된 숫자열중 앞의 두 개씩 두 숫자의 합 연산을 통해 최종적으로 한 값을 도출해내는 code를 다중 프로세스와 다중 쓰레드로 구현하는 부분(3-2), CPU scheduling algorithm 중 FCFS, SJF,RR,SRTF 4가지를 구현하여 미리 burst time과 arrival time을 안 채로 test case를 돌려 간트 차트로 출력해야하는 부분(3-3)으로 구성된다.

Result

(3-1)
우선 fork count 출력을 위해 sched.c와 fork.c 파일에 fork_count를 추가해주었다. sched.c의 경우 task_struct 함수 안에 fork_count 변수를 정의해주었다.

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;
        int fork_count;
        /*
         * This begins the randomizable portion of task_struct. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

        void                            *stack;
```

fork.c 파일의 경우 아래와 같이 새로운 프로세스가 생성 될 때 fork_count를 최기화해주어야 하므로 copy_process 함수 안에 count변수를 0으로 초기화해주었고,

```
        retval = -ENOMEM;
        p = dup_task_struct(current, node);
        if (!p)
                goto fork_out;

        p->fork_count =0;
```

do_fork 함수에서 copy_process 전에 current->fork_count++을 통해 fork가 호출될 때마다 fork_count값이 증가되게끔 했다.

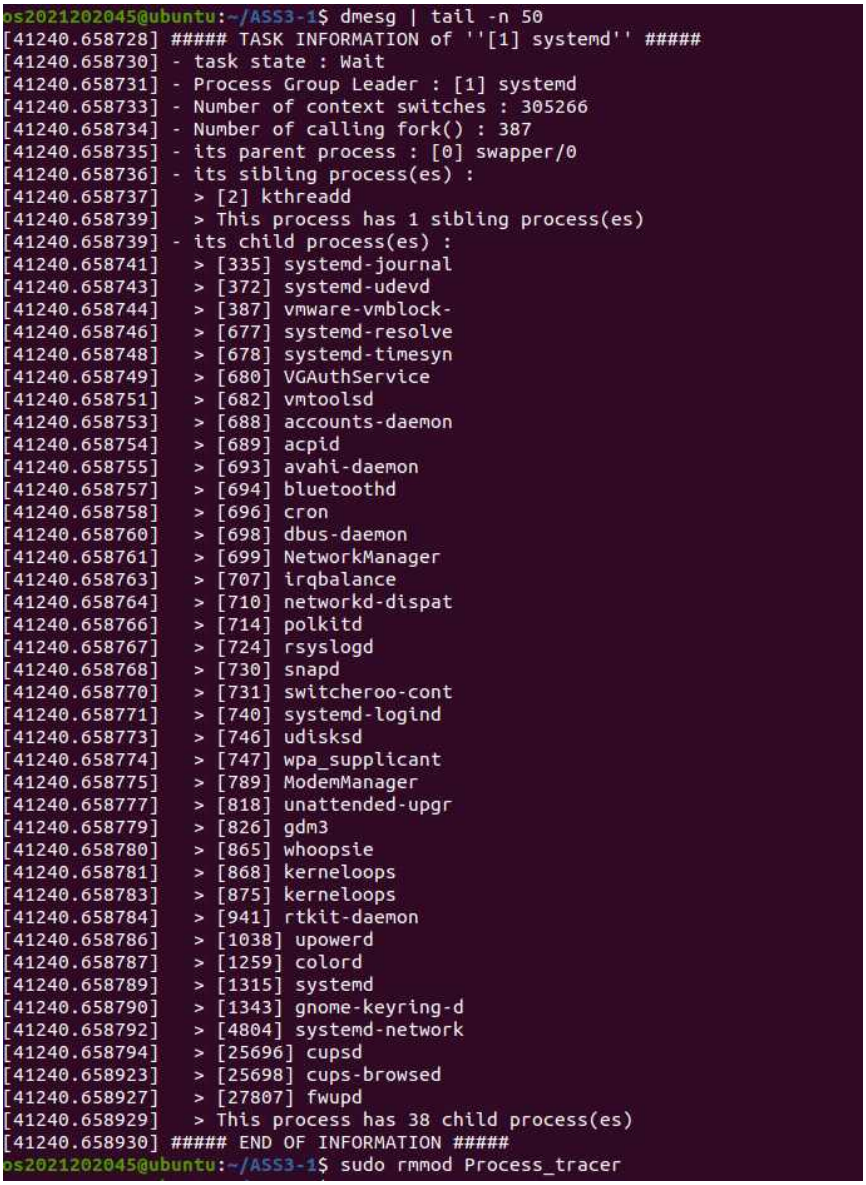
```
 */
long _do_fork(struct kernel_clone_args *args)
{
        u64 clone_flags = args->flags;
        struct completion vfork;
        struct pid *pid;
        struct task_struct *p;
        int trace = 0;
        long nr;

        /*
         * Determine whether and which event to report to ptracer.  When
         * called from kernel_thread or CLONE_UNTRACED is explicitly
         * requested, no event is reported; otherwise, report if the event
         * for the type of forking is enabled.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if (args->exit_signal != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
        }
        current->fork_count++;
        p = copy_process(NULL, trace, NUMA_NO_NODE, args);
        add_latent_entropy();

```

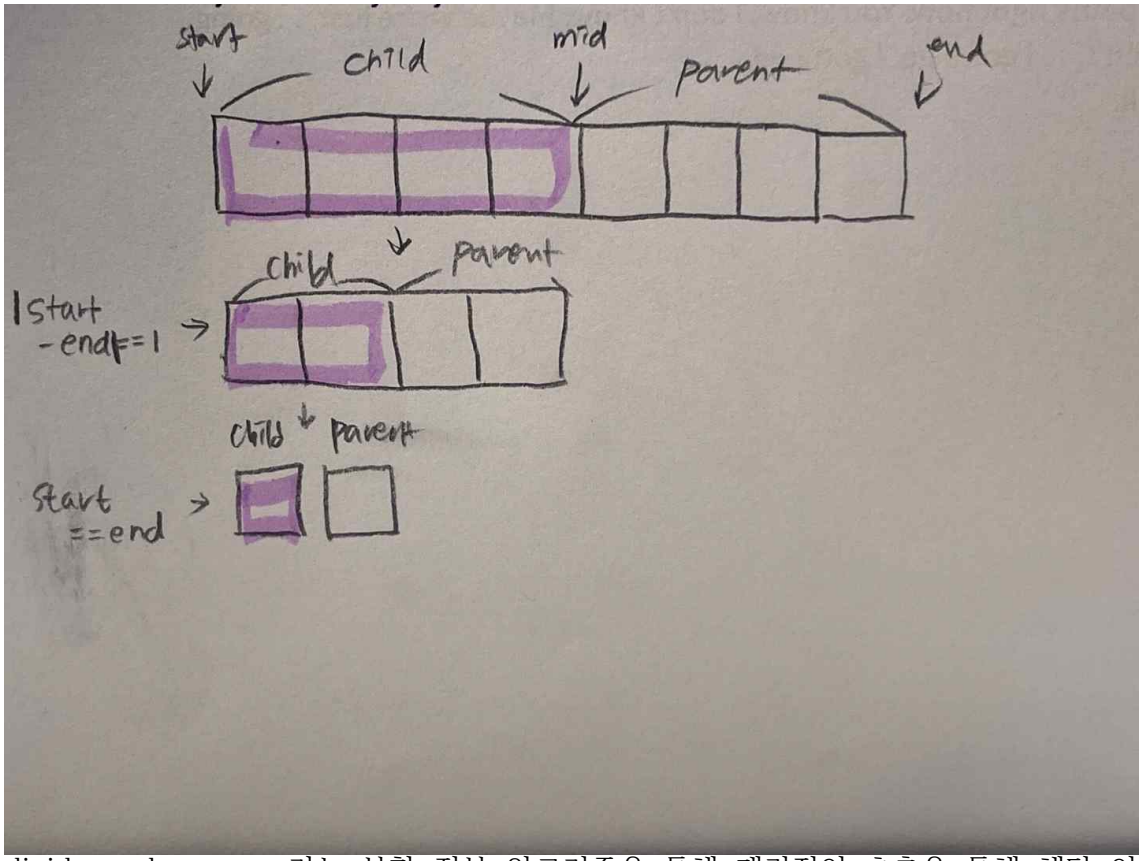
결과화면은 다음과 같다.



(3-2)

기본적인 흐름은 numgen.c파일에서 temp.txt파일에 MAX_PROCESS의 2배만큼 각 줄에 숫자를 하나씩 출력해준다. fork.c의 경우 각 숫자의 두 개씩의 연산은 다중 프로세스로 접근하여 해결하고, thread.c의 경우 해당 연산은 다중 쓰레드로 해결한다.

기본적인 로직은 다음과 같다.



divide and conquer라는 분할 정복 알고리즘을 통해 재귀적인 호출을 통해 해당 연산을 해결하는 접근을 했고, thread의 경우 동일하되 부모와 메모리 등의 자원을 공유하기 때문에 각 쓰레드마다 start와 end, result 정보를 스스로 가지고 있어야만 했기 때문에 구조체를 이용하여 이 정보를 갖고 있도록 하였다.

우선 MAX_PROCESS의 크기가 8인 경우 다중 프로세스와 다중 쓰레드의 결과값을 봐보자.

```
os2021202045@ubuntu:~/ASS3-2$ ./numgen
os2021202045@ubuntu:~/ASS3-2$ ./thread
value of thread : 136
0.005975
os2021202045@ubuntu:~/ASS3-2$ ./fork
value of fork : 136
0.002647
os2021202045@ubuntu:~/ASS3-2$
```

위와 같이 결과 나오는데, 다중 프로세스 및 다중 쓰레드 둘 다 136으로 알맞은 값을 출력하는 걸 볼 수 있다. 실행 시간의 경우 다중 프로세스가 0.002647로 다중 쓰레드로 연산했을 때보다 빠르게 계산하는 것을 확인할 수 있다.

물론 아래와 같이 연산 이후 temp.txt에 잘 연산 과정 값이 잘 적히는 걸 확인할 수 있다.

```
 1 1
 2 2
 3 3
 4 4
 5 5
 6 6
 7 7
 8 8
 9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 15
18 7
19 23
20 31
21 27
22 58
23 19
24 11
25 42
26 3
27 100
28 10
29 26
30 36
31 136
```

그럼 다음으로 MAX_PROCESS의 수가 64일 때 실행해보자.

```
os2021202045@ubuntu:~/ASS3-2$ rm -rf tmp*
os2021202045@ubuntu:~/ASS3-2$ sync
os2021202045@ubuntu:~/ASS3-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202045@ubuntu:~/ASS3-2$ make
gcc -Wall -pthread -o fork fork.c
gcc -Wall -pthread -o thread thread.c
gcc -Wall -pthread -o numgen numgen.c
os2021202045@ubuntu:~/ASS3-2$ ./numgen
os2021202045@ubuntu:~/ASS3-2$ ./thread
value of thread : 8256
0.040736
os2021202045@ubuntu:~/ASS3-2$ ./fork
value of fork : 64
0.018668
os2021202045@ubuntu:~/ASS3-2$
```

위와 같이 thread의 경우 8256으로 결과값이 적절히 나온 반면, 다중프로세스로 구현한 경우 64로 값이 이상하게 나오는 걸 확인할 수 있다. 그 이유는 fork의 경우 exit()라고 프로세스를 종료할 때 종료 상태 코드를 반환하는 함수가 있는 종료 상태 코드는 8비트로 제한(POSIX 시스템에서)되어 있기 때문에 0부터 255까지의 범위 안의 값으로 변환되어야한다. 따라서 코드에 이를 반영해준 것이 다음이다.

```
  if (left_sum > 256) {
      left_sum %= 256;
  }
  exit(left_sum);
```

또한 자식 프로세스가 종료될 때까지 기다리는 wait()의 경우, 16비트 정수로 함수가 반환되어야 하는데, 이 값의 하위 8비트는 종료 상태 코드이고 상위 8비트는 기타 정보를 포함하기 때문에 실제 종료 상태는 하위 8비트에 저장된다. 따라서 8비트만큼 shift 해주어야 우리가 원하는 status를 얻을 수 있다. 그 코드가 다음과 같다.

```
wait(&child_sum);
int left_sum = (child_sum>>8); //shift 8bits
```

다중 쓰레드의 경우 이런 제약이 없고, 부모와 자원을 공유하기 때문에 8256이라는 결과값을 도출해낼 수 있는 것이다.

(3-3)

```
typedef struct {
    int pid;
    int arrival;
    int burst;
    int remaining;
    int start;
    int total_wait;
    int end;
    int final_end;

} process; //store information
```

process마다 위의 data(pid,도착시간,burst,남은시간,첫 실행 시작 시각, 총 wait time, 각 실행마다의 끝점, 최종적으로 완료한 시점)를 가지게 한 뒤 이 data를 활용하여 구현을 했다.

test case 1에 대해 실행해보면 결과화면이 다음과 같이 나온다.

```
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.1 FCFS
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P3 | P3 | P3 |
P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6
Average Waiting Time= 14.17
Average Turnaround Time=21.00
Average Response Time=14.17
CPU Utilization=98.56%
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.1 SJF
Gantt Chart:
| P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P4 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P3 | P5 | P5 | P5 | P5 |
P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1
Average Waiting Time= 10.83
Average Turnaround Time=17.67
Average Response Time=10.83
CPU Utilization=98.56%
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.1 RR 2
time slice: 2
Gantt Chart:
| P1 | P1 | P2 | P2 | P1 | P1 | P3 | P3 | P2 | P2 | P1 | P1 | P4 | P4 | P3 | P3 | P5 | P5 | P6 | P6 | P2 | P2 |
P1 | P1 | P4 | P4 | P3 | P5 | P5 | P6 | P6 | P2 | P2 | P1 | P1 | P5 | P5 | P6 | P6 | P2 | P6
Average Waiting Time= 22.50
Average Turnaround Time=29.33
Average Response Time=4.00
CPU Utilization=94.91%
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.1 SRTF
Gantt Chart:
| P2 | P2 | P2 | P3 | P3 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P2 | P2 | P2 | P2 | P2 | P2 | P5 | P5 | P5 | P5 |
P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1
Average Waiting Time= 10.50
Average Turnaround Time=17.33
Average Response Time=9.00
CPU Utilization=98.32%
```

Average waiting time: RR>FCFS>SJF>SRTF
Average turnaround time: RR>FCFS>SJF>SRTF
Average response time: FCFS>SJF>SRTF>RR
CPU utilization: FCFS=SJF>SRTF>RR

FCFS의 경우 선입선출로, 각 프로세스의 burst time에 상관없이 오직 arrival time에 따라 가장 빠르게 도착한 process부터 레디 큐에 넣어 실행시킨다. 따라서 만약 가장 실행시간이 긴 프로세스가 처음에 실행되어버린다면 전체적으로 효율성이 떨어진다. 위의 예시만 봐도 process 1이 0초에 도착했으나 cpu burst time이 가장 길었어서 response time부분에서 가장 취약한 알고리즘으로 결과가 나왔다. 하지만 context switch는 줄일 수 있어 overhead를 줄일 수 있어 cpu utilization부분에서는 강력한 효과를 보인다.

SRTF의 경우 SJF에서 OS가 CPU를 빼어 가져올 수 있는 권한이 있기 때문에 overhead는 더 걸려 cpu utilization에서는 SJF보다는 취약할 수 있지만, 가장 실행시간이 적은 프로세스를 계속해서 update해주면서 레디큐에 넣어주므로 turn around time과 waiting time에서 강점을 보인다. 하지만 response time의 경우 먼저 들어왔어도 shortest job이 아니면 실행 순서가 자꾸 뒤로 밀리기 때문에 좋은 성능을 보이진 않는다.

SJF의 경우 남은 CPU BURST가 가장 적은 JOB부터 먼저 레디큐에 넣어 실행하는 것인데 OS가 CPU를 빼지 않아서 유동적으로 update해줄 수는 없다. 따라서 process 1과 process 2가 동시에 도착했을 때 process 2가 cpu burst time이 더 적기 때문에 process 2가 먼저 실행되는 것을 확인할 수 있다. 따라서 FCFS보다는 더 효율적인 알고리즘이라 위의 모든 기준에서 FCFS보다 SJF가 더 효율적인 것을 확인할 수 있다.(CPU utilization은 동일 -> context switch가 같기 때문에)

RR의 경우 remaining time과는 상관없이 time slice를 2초씩 주어서 모든 process가 균등하게 실행될 수 있도록 한다. RR의 경우 I/o의 개입이 있을 시 효과적인 것을 더욱 뚜렷하게 확인할 수 있다. context switch가 빈번하게 일어나서 overhead가 많이 걸려 cpu utilization부분에서는 가장 취약한 것을 확인할 수 있다. 이 알고리즘을 구현할 때 만약 process1이 time slice가 만료되어 다시 레디 큐에 들어가야하고, process3이 그 타임에 딱 도착하여 레디큐에 들어가야할 때 막 도착한 프로세스를 먼저 레디큐에 넣는 알고리즘을 구현하는 것이 어려웠지만 레디큐에 넣는 알고리즘을 따

로 함수로 빼서 이를 활용하게끔하여 전체적인 코드의 중복도 줄이고, 구현도 할 수 있었다. RR의 경우 response time에서 가장 강력한 효율이 보이는데 2초씩 process가 최대로 실행되므로 도착하고부터 가장 처음으로 실행되기까지 가장 빠르다는 의미이다. 하지만 완료되기까지 다른 프로세스들을 많이 기다려야하므로 waiting time이 가장 크다.

test case 2에 대한 결과화면은 다음과 같다.

```
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.2 FCFS
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6
Average Waiting Time= 5.83
Average Turnaround Time=9.33
Average Response Time=5.83
CPU Utilization=97.22%
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.2 SJF
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6
Average Waiting Time= 5.83
Average Turnaround Time=9.33
Average Response Time=5.83
CPU Utilization=97.22%
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.2 RR 2
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P4 | P4 | P5 | P5 | P6 | P6 | P3 | P4 | P4 | P5 | P5 | P6 | P6 | P5 | P6 | P6
Average Waiting Time= 8.17
Average Turnaround Time=11.67
Average Response Time=4.17
CPU Utilization=94.59%
os2021202045@ubuntu:~/ASS3-3$ ./cpu_scheduler input.2 SRTF
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6
Average Waiting Time= 5.83
Average Turnaround Time=9.33
Average Response Time=5.83
CPU Utilization=97.22%
```

Average waiting time: RR>FCFS=SJF=SRTF
Average turnaround time: RR>FCFS=SJF=SRTF
Average response time: FCFS=SJF=SRTF>RR
CPU utilization: FCFS=SJF=SRTF>RR
성능의 비교가 위의 case와 비슷한 순서로 결과가 나온다.

고찰

3-1을 짜면서 context switch 수를 구할 때 task_struct에서 두 개의 인자에 접근해야한다는 점이 중요했다. nvcsw의 경우 voluntary로 cpu가 스스로 사용을 포기할 때 발생하는 켄텍스트 스위치인데 예를 들어, 입출력 작업을 기다리거나 sleep()호출로 대기 상태로 전환할 때 발생한다. nivcsw의 경우 프로세스가 cpu시간을 초과하여 강제로 중단될 때 발생되는 컨텍스트 스위치로 이 둘을 합산해야 원하는 전체적인 컨텍스트 스위치 횟수를 구할 수 있었다. 또한, 이번에 thread는 process와 달리 부모와 자원을 공유해서 각 쓰레드마다 자신의 정보를 가지고 다녀야 하기 때문에 구조체 개념을 사용해야한다는 것까지 생각이 오래걸렸다. 3-3의 경우 위에 썼다 시피 RR의 레디큐에 PROCESS를 넣을 때 두 PROCESS가 동시에 충돌하면 어느 PROCESS부터 레디큐에 넣어줘야하는지 그리고 이를 구현하는 부분이 까다로웠지만 함수를 따로 빼서 해결하였다.