

운영체제실습

assignment 3

담당교수 : 김태석 교수님

학 번 : 2021202058

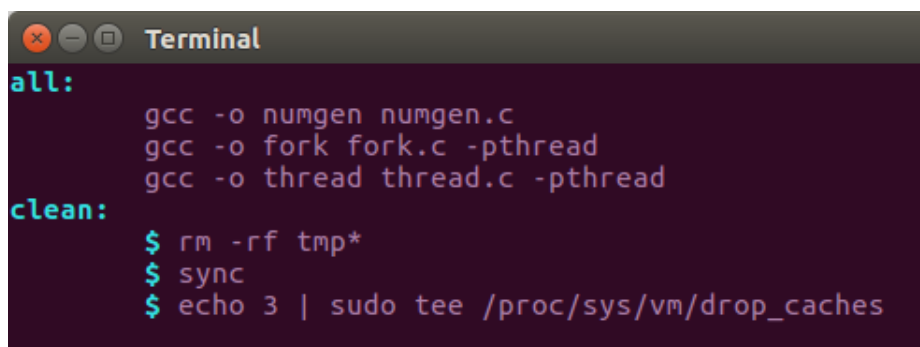
성 명 : 송채영

1. Introduction

우선 3-1 과제에서는 fork 와 thread 를 통해 부모 프로세스/스레드에게 값을 전달하는 과정을 반복한 후 같은 기능을 구현하였을 때의 성능차이와 결과의 차이를 비교해본다. 3-2 과제에서는 CPU 스케줄링 정책을 변경해보고 nice, priority 값을 변경해보며 나타나는 성능차이를 비교해본다. 마지막으로 3-3 과제에서는 2 차 과제에서 진행했던 코드와 task_struct 를 바탕으로 pid 를 입력 받아 각 프로세스의 정보를 출력해보는 것이 목표이다.

2. Result

3-1



```
all:
    gcc -o numgen numgen.c
    gcc -o fork fork.c -pthread
    gcc -o thread thread.c -pthread

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

Makefile 을 다음과 같이 설정하여 캐시 및 버퍼를 비워 실험에 영향을 주는 요소를 제거하도록 하였다.

fork.c 구현에 대해 설명하겠다. 우선 파일에서 데이터를 읽어 배열에 저장한 후 프로세스를 생성한다. 각 자식 프로세스는 연산을 진행한 뒤 exit 를 통해 처리결과를 반환하며 부모 프로세스는 자식 프로세스의 실행을 wait 를 통해 기다린다.

thread.c 구현에 대해 설명하겠다. 동일하게 파일에서 데이터를 읽어 배열에 저장한 후 스레드를 생성해 각 스레드는 배열의 연산을 진행하며, 이때 pthread_create 와 pthread_join 을 활용해 스레드를 생성하고 기다린다.

두 코드 모두 배열 요소를 병렬로 처리하는 방식을 사용하며, 하나는 프로세스를 사용하고 다른 하나는 스레드를 사용한다.

우선 MAX_PROCESS = 8 일때의 결과이다.

```

os2021202058@ubuntu:~/Assignment3/3-1$ make clean
rm -rf tmp*
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202058@ubuntu:~/Assignment3/3-1$ make
gcc -o numgen numgen.c
gcc -o fork fork.c -pthread
gcc -o thread thread.c -pthread

```

과제 요구사항에 맞게 make clean 을 매 실험 전 진행하였다.

```

os2021202058@ubuntu:~/Assignment3/3-1$ ./numgen
os2021202058@ubuntu:~/Assignment3/3-1$ cat temp.txt
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

```

numgen.c 파일에서는 특정 파일 temp.txt 를 생성한 후 1 부터 프로세스 수의 2 배까지 기록하도록 구현하였다. 실행파일을 실행한 후, temp.txt 파일을 출력해보았을 때 16 까지 잘 저장되어 있는 것을 확인할 수 있다.

```

os2021202058@ubuntu:~/Assignment3/3-1$ ./fork
value of fork : 136
0.002231
os2021202058@ubuntu:~/Assignment3/3-1$ ./thread
value of thread: 136
0.000844
os2021202058@ubuntu:~/Assignment3/3-1$ █

```

결과를 확인해보면 MAX_PROCESSES 가 8 일 때 136 이 정상적으로 출력되며, thread 의 실행결과가 더 빠른 것을 확인할 수 있다.

```

os2021202058@ubuntu:~/Assignment3/3-1$ make clean
rm -rf tmp*
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202058@ubuntu:~/Assignment3/3-1$ make
gcc -o numgen numgen.c
gcc -o fork fork.c -pthread
gcc -o thread thread.c -pthread
os2021202058@ubuntu:~/Assignment3/3-1$ ./numgen
os2021202058@ubuntu:~/Assignment3/3-1$ cat temp.txt

```

이어서 MAX_PROCESS = 64 일때의 결과이다.

```

116
117
118
119
120
121
122
123
124
125
126
127
128
os2021202058@ubuntu:~/Assignment3/3-1$ ./fork
value of fork : 64
0.012989
os2021202058@ubuntu:~/Assignment3/3-1$ ./thread
value of thread: 8256
0.004640
os2021202058@ubuntu:~/Assignment3/3-1$ █

```

numgen 실행파일을 실행한 후, temp.txt 파일을 출력해보았을 때 64 의 2 배인 128 까지 잘 저장되어 있는 것을 확인할 수 있다. 이어서 결과를 확인해보면 MAX_PROCESSES 가 64 일 때 thread 의 결과는 8256 이 정상적으로 출력되지만, fork 의 결과는 64 가 출력된 것을 확인할 수 있으며 thread 의 실행결과가 더 빠른 것 역시 확인할 수 있다.

우선 MAX_PROCESSES 가 8, 64 일 때 모두 fork 의 실행시간보다 thread 의 실행시간이 빠른 것을 확인할 수 있었다. fork 방식은 프로세스를 복제하여 별도의 메모리 공간을 사용하는 반면, thread 방식은 단일 프로세스 내에서 여러 스레드가 공유하는 메모리 공간을 사용하기 때문에, 이러한 차이로 인해 fork 방식에서의 실행시간이 더 느릴 수 있다.

이어서 자식 프로세스에서 부모 프로세스로 값을 넘겨줄 때 반환 값이 2^8 이상이면 안 되며 8bit 만큼 right shift 해주어야 정상적인 값이 반환될 수 있다. 그 이유는 다음과 같다. 16bits 인 exit code 중 상위 8 bits 는 exit()에서의 exit code 이며 하위 8bits 는 0 일 때 정상 종료임을 뜻한다. 상위 8bits 를 얻기 위해서 right 8bits shift 를 해주어야 하며 이때 2^8 이상의 정보는 담을 수 없다. 따라서 MAX_PROCESSES 가 64 일 때 범위를 초과하기 때문에 64 만 출력된 것을 알 수 있다.

3-2

```
/**
 * sched_setscheduler - change the scheduling policy and/or RT priority of a thread.
 * @p: the task in question.
 * @policy: new policy.
 * @param: structure containing the new RT priority.
 *
 * Return: 0 on success. An error code otherwise.
 * NOTE that the task may be already dead.
 */
int sched_setscheduler(struct task_struct *p, int policy,
                      const struct sched_param *param)
{
    return _sched_setscheduler(p, policy, param, true);
}
```

각 프로세스에서 CPU 스케줄링 정책을 변경할 때 sched_setscheduler() 함수를 사용하기 때문에 원형을 확인해보았다.

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

struct sched_param {
    int sched_priority;
};
```

위 sched_setscheduler() 함수의 파라미터 중 하나로 우선 순위나 관련된 정보를 저장하는데 사용하는 구조체의 원형이다.

filegen.c 파일에서는 temp 디렉토리를 생성한 후 랜덤한 정수 데이터가 적혀있는 파일을 MAX_PROCESSES 개수, 즉 10000 만큼 생성하고 1~9 사이의 랜덤 정수가 각 파일 안에 저장되어 있다. 이때 MAX_PROCESSES 의 개수는 10000 으로 설정하였는데, 작게하면 성능 비교가 안 되기 때문에 10000 이상으로 설정해주어야 한다.

```
os2021202058@ubuntu:~/Assignment3/3-2$ make clean
rm -rf tmp*
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202058@ubuntu:~/Assignment3/3-2$ make
gcc -o filegen filegen.c
gcc -o schedtest schedtest.c
```

```

os2021202058@ubuntu:~/Assignment3/3-2$ ./filegen
os2021202058@ubuntu:~/Assignment3/3-2$ ls temp
0      1263 1529 1795 206 2325 2591 2857 3121 3388 3653 3919 4184
1      1264 153 1796 2060 2326 2592 2858 3122 3389 3654 392 4185
10     1265 1530 1797 2061 2327 2593 2859 3123 339 3655 3920 4186
100    1266 1531 1798 2062 2328 2594 286 3124 3390 3656 3921 4187
1000   1267 1532 1799 2063 2329 2595 2860 3125 3391 3657 3922 4188
1001   1268 1533 18 2064 233 2596 2861 3126 3392 3658 3923 4189
1002   1269 1534 180 2065 2330 2597 2862 3127 3393 3659 3924 419
1003   127 1535 1800 2066 2331 2598 2863 3128 3394 366 3925 4190
1004   1270 1536 1801 2067 2332 2599 2864 3129 3395 3660 3926 4191
1005   1271 1537 1802 2068 2333 26 2865 313 3396 3661 3927 4192
1006   1272 1538 1803 2069 2334 260 2866 3130 3397 3662 3928 4193
1007   1273 1539 1804 207 2335 2600 2867 3131 3398 3663 3929 4194
1008   1274 154 1805 2070 2336 2601 2868 3132 3399 3664 393 4195
1009   1275 1540 1806 2071 2337 2602 2869 3133 34 3665 3930 4196
101    1276 1541 1807 2072 2338 2603 287 3134 340 3666 3931 4197
1010   1277 1542 1808 2073 2339 2604 2870 3135 3400 3667 3932 4198
1011   1278 1543 1809 2074 234 2605 2871 3136 3401 3668 3933 4199
1012   1279 1544 181 2075 2340 2606 2872 3137 3402 3669 3934 42
1013   128 1545 1810 2076 2341 2607 2873 3138 3403 367 3935 420
1014   1280 1546 1811 2077 2342 2608 2874 3139 3404 3670 3936 4200
1015   1281 1547 1812 2078 2343 2609 2875 314 3405 3671 3937 4201
1016   1282 1548 1813 2079 2344 261 2876 3140 3406 3672 3938 4202
1017   1283 1549 1814 208 2345 2610 2877 3141 3407 3673 3939 4203
1018   1284 155 1815 2080 2346 2611 2878 3142 3408 3674 394 4204
1019   1285 1550 1816 2081 2347 2612 2879 3143 3409 3675 3940 4205

```

Temp 파일이 제대로 생성되었는지 확인해보았다. 사진상에선 잘렸지만 0~9999 까지 총 10000 개가 정상적으로 생성되었음을 확인했다.

schedtest.c 파일에서는 CPU 스케줄링 방식(SCHED_OTHER, SCHED_FIFO, SCHED_RR)을 테스트하고 입력된 nice 또는 priority 값에 따라 프로세스의 스케줄링 동작을 비교하도록 구현하였다.

우선 SCHED_OTHER 은 The standard round-robin time-sharing policy 이며 이 경우에는 nice 값을 설정해주어야 한다. -20~19의 값을 가질 수 있으며 19 일 때 lowest 값을, -20 일 때 highest 값을, 0 일 때 default 값을 가진다.

다음으로 SCHED_FIFO 는 A first-in first-out policy 고, SCHED_RR 은 A round-robin policy 를 말하며, 이 경우는 priority를 설정해야 한다. 1~99의 값을 가질 수 있으며 1 일 때 lowest 값, 99 일 때 highest 값을 가지며 default 값으로 50 을 설정하였다. 강의 자료에선 0 부터라고 나와있지만 리눅스 공식 홈페이지를 참고하였을 때 1 부터 99 로 나와있어 1~99 로 진행하였다.

결과를 살펴보겠다.

```

os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 1
Enter a nice value(-20, 0, 19) : -20
0.500104
os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 1
Enter a nice value(-20, 0, 19) : 0
0.308011
os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 1
Enter a nice value(-20, 0, 19) : 19
0.287722

```

우선 SCHED_OTHER 의 결과이다. Nice 값이 더 작은 프로세스, -20 일 때 높은 우선 순위를 가지게 되며 CPU 자원을 더 많이 확보하게 되어 실행시간이 짧아야 하지만 결과는 정 반대로 나왔다. 그 이유를 추측해보았는데, 우선 I/O bound workload 는 입출력 작업이 완료될 때까지 소요되는 시간이다. 이때 nice 값이 19 이면, 가장 낮은 우선순위를 할당 받아 CPU 를 적극적으로 활용하는 다른 프로세스를 선점할 가능성이 줄어들며, I/O 작업을 수행할 때 더 유리하게 되므로 대기시간 역시 줄어들게 된다. 따라서 I/O 작업 시 효율성이 향상되는 것 같다. 또한 우선순위가 낮은 프로세스는 CPU 경쟁 시간이 덜하기 때문에 이러한 이유들로 인해 가장 짧은 실행시간을 얻게 되었다고 생각한다.

```

os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 2
Enter a priority value(1, 50, 99) : 1
0.303845
os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 2
Enter a priority value(1, 50, 99) : 50
0.310067
os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 2
Enter a priority value(1, 50, 99) : 99
0.301632

```

다음으로 SCHED_FIFO 의 결과이다. Priority 가 각각 1, 50, 99 일 때의 결과는 매 실행 마다 다르게 나온다.

```

os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 3
Enter a priority value(1, 50, 99) : 1
0.313456
os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 3
Enter a priority value(1, 50, 99) : 50
0.306342
os2021202058@ubuntu:~/Assignment3/3-2$ ./schedtest
CPU Scheduling
1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR
Enter scheduling number 3
Enter a priority value(1, 50, 99) : 99
0.309318
os2021202058@ubuntu:~/Assignment3/3-2$ █

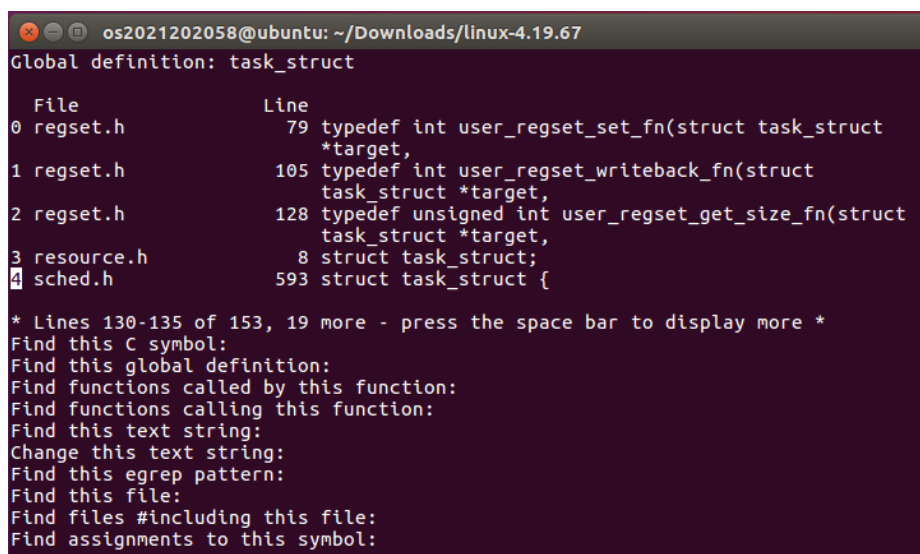
```


다음으로 SCHED_RR 의 결과이다. Priority 가 각각 1, 50, 99 일 때의 결과는 매 실행 마다 다르게 나온다.

두 결과 모두 우선순위를 설정하는 것에 큰 차이가 발생하지 않았다. FIFO 는 들어온 값을 차례로 실행하기 때문에 우선순위에 영향을 크게 받지 않게 되어 위와 같은 결과가 나타났다고 생각한다. 또한 RR 의 경우 time slice 를 가져 시간적인 차이를 보일 것이라고 예상했지만, 시간적인 면에서 큰 차이를 느끼지 못했다. 실험을 진행하면서 버퍼를 비워주는 clean 을 매번 진행해 영향을 주는 요소를 줄였음에도 실험 결과가 일정하게 나오지 않는 것 같다. 또한 더 복잡한 프로그램을 실행해 실험을 해본다면 결과를 더 확실하게 알 수 있을 것이라고 예상된다.

3-3

우선 fork()가 호출되는 횟수를 저장하기 위해 cscope 로 sched.h 를 찾았다.



```
os2021202058@ubuntu: ~/Downloads/linux-4.19.67
Global definition: task_struct

  File          Line
0 regset.h       79 typedef int user_regset_set_fn(struct task_struct
   *target,
1 regset.h       105 typedef int user_regset_writeback_fn(struct
   task_struct *target,
2 regset.h       128 typedef unsigned int user_regset_get_size_fn(struct
   task_struct *target,
3 resource.h      8 struct task_struct;
4 sched.h        593 struct task_struct {

* Lines 130-135 of 153, 19 more - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

```

os2021202058@ubuntu: ~/Downloads/linux-4.19.67
1199 #ifdef CONFIG_SECURITY
1200     /* Used by LSM modules for access restriction: */
1201     void                                *security;
1202 #endif
1203     int fork_count;
1204     /*
1205      * New fields for task_struct should be added above here, so that
1206      * they are included in the randomized portion of task_struct.
1207      */
1208     randomized_struct_fields_end
1209
1210     /* CPU-specific state of this task: */
1211     struct thread_struct                thread;
1212
1213     /*
1214      * WARNING: on x86, 'thread_struct' contains a variable-sized
1215      * structure. It *MUST* be at the end of 'task_struct'.
1216      *
1217      * Do not put anything below here!
1218      */
1219 };
1220
1221 static inline struct pid *task_pid(struct task_struct *task)
-- INSERT --
1203,16-23    63%

```

Sched.h 파일에 fork 의 호출횟수를 count 할 변수를 추가해주었다. 주석을 살펴보면, 새로운 변수는 해당 주석 위에 선언하라고 되어 있어 해당 주석 위에 선언하였다. 이 변수를 통해 각 프로세스가 fork 를 호출한 횟수를 저장할 수 있다.

```

os2021202058@ubuntu: ~/Downloads/linux-4.19.67
File: fork.c
File
0 drivers/pinctrl/intel/pinctrl-cedarfork.c
1 fs/xfs/libxfs/xfs_inode_fork.c
2 kernel/fork.c
3 tools/testing/selftests/powerpc/benchmarks/fork.c
4 tools/testing/selftests/powerpc/pmu/ebb/fork_cleanup_test.c
5 tools/testing/selftests/powerpc/tm/tm-fork.c

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:

```

이어서 fork count 를 초기화 하거나 증가하기 위해 cscope 로 fork.c 를 찾았다.

```

os2021202058@ubuntu: ~/Downloads/linux-4.19.67
return task;
}

/*
 * Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long _do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr,
              unsigned long tls)
{
    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 ((clone_flags & CSIGNAL) != SIGCHLD)
            trace = PTRACE_EVENT_CLONE;
        else
            trace = PTRACE_EVENT_FORK;

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

    p = copy_process(clone_flags, stack_start, stack_size,
                    child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
    p->fork_count = 0;
    current->fork_count++;

    add_latent_entropy();

    if (IS_ERR(p))
        return PTR_ERR(p);
}

```

Fork 가 동작하는 부분인 do_fork 부분을 보면 p = copy_process 부분을 확인할 수 있다. p->fork_count = 0 을 통해 새로 생성된 자식 프로세스의 fork 카운트를 0 으로 초기화 하며, current->fork_count++;을 통해 fork 를 호출한 부모 프로세스의 count 개수를 증가시키는 코드를 추가해주었다.

커널 코드를 수정하였으므로 make -j12, make modules_install, make install, reboot 과정을 거쳐주었다.

3-3 에서 요구하는 프로세스 정보를 출력하기 위해 각각의 원형을 찾아보았다.

```

/*
 * executable name, excluding path.
 *
 * - normally initialized setup_new_exec()
 * - access it with [gs]et_task_comm()
 * - lock it with task_lock()
 */
char                                comm[TASK_COMM_LEN];

```

첫번째로 프로세스 이름과 관련된 부분이다. comm 에 접근하여 받아올 수 있음을 확인할 수 있다.

첫번째 출력 문장을 보면 pid 도 같이 출력을 하는데, next_state 를 활용하여 모든 프로세스를 돌며 찾도록 구현하였다.

```
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */

/* 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
```

다음으로 현재 프로세스의 상태와 관련된 부분이다. 주석에 나와있는 것처럼, Running or ready, Wait with ignoring all signals, Wait, Stopped 의 경우 tsk->state 를 활용하며, , Zombie process, Dead 의 경우 exit_state 를 사용해 구현하였다.

```
/*
 * Children/sibling form the list of natural children:
 */
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;
```

다음으로 프로세스 그룹 정보와 관련된 부분이다. 그룹리더를 통해 group 에 대한 id 와 이름을 얻을 수 있다.

```
#endif

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

다음으로 해당 프로세스를 실행하기 위해 수행된 context switch 횟수이다.

```
/*
 * Pointers to the (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->real_parent->pid)
 */

/* Real parent process: */
struct task_struct __rcu    *real_parent;

/* Recipient of SIGCHLD, wait4() reports: */
struct task_struct __rcu    *parent;

/*
 * Children/sibling form the list of natural children:
 */
struct list_head            children;
struct list_head            sibling;
struct task_struct          *group_leader;
```

다음으로 부모 프로세스, 형제자매 프로세스, 자식 프로세스와 관련된 부분이다. 부모 프로세스에서는 parent 를 사용하였다. 형제자매 프로세스는 부모의 자식 프로세스를 모두 출력하면 되므로, list_for_each 와 list_entry 를 사용하였는데, 이때 자기자신을 제외하고 출력하도록 하였다. 자식 프로세스도 형제자매 프로세스와 동일하게 구현하였다.

```
os2021202058@ubuntu:~/Assignment3/3-3$ sudo make
make -C /lib/modules/4.19.67-2021202058/build SUBDIRS=/home/os2021202058/Assignment3/3-3 modules
make[1]: Entering directory '/home/os2021202058/Downloads/linux-4.19.67'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory '/home/os2021202058/Downloads/linux-4.19.67'
os2021202058@ubuntu:~/Assignment3/3-3$ sudo insmod process_tracer.ko
os2021202058@ubuntu:~/Assignment3/3-3$ ./test
os2021202058@ubuntu:~/Assignment3/3-3$
```

Makefile 을 통해 커널 모듈을 컴파일 한 후 명령어를 활용하여 모듈을 커널에 적재하고 간단한 테스트를 진행하였다.

```

[ 6244.917307] ##### TASK INFORMATION of '[1] systemd' #####
[ 6244.917308] - task state : Wait with ignoring all signals
[ 6244.917309] - Process Group Leader : [1] systemd
[ 6244.917310] - Number of context switches : 3553
[ 6244.917310] - Number of calling fork() : 140
[ 6244.917311] - its parent process : [0] swapper/0
[ 6244.917311] - its sibling process(es) :
[ 6244.917312]   > [2] kthreadd
[ 6244.917312]   > This process has 1 sibling process(es)
[ 6244.917313] - its child process(es) :
[ 6244.917313]   > [396] systemd-journal
[ 6244.917314]   > [420] systemd-udevd
[ 6244.917315]   > [452] vmware-vmblock-
[ 6244.917315]   > [455] vmtoolsd
[ 6244.917316]   > [543] systemd-timesyn
[ 6244.917316]   > [939] bluetoothd
[ 6244.917317]   > [941] cupsd
[ 6244.917317]   > [942] VGAuthService
[ 6244.917318]   > [954] acpid
[ 6244.917318]   > [955] systemd-logind
[ 6244.917319]   > [956] cron
[ 6244.917320]   > [961] dbus-daemon
[ 6244.917320]   > [971] NetworkManager
[ 6244.917321]   > [973] accounts-daemon
[ 6244.917321]   > [975] rsyslogd
[ 6244.917322]   > [997] agetty
[ 6244.917322]   > [1023] irqbalance
[ 6244.917323]   > [1024] cups-browsed
[ 6244.917323]   > [1075] lightdm
[ 6244.917324]   > [1079] polkitd
[ 6244.917324]   > [1272] upowerd
[ 6244.917325]   > [1289] colord
[ 6244.917325]   > [1294] rtkit-daemon
[ 6244.917326]   > [1458] avahi-daemon
[ 6244.917327]   > [1507] whoopsie
[ 6244.917327]   > [1650] systemd
[ 6244.917328]   > [1657] gnome-keyring-d
[ 6244.917329]   > [2216] udisksd
[ 6244.917329]   > [2307] fwupd
[ 6244.917329]   > This process has 29 child process(es)
[ 6244.917330] ##### END of INFORMATION #####
os20211202058@ubuntu:~/Assignment3/3-3$

```

dmseg 를 통해 커널 메시지를 출력한 결과이다. 제안서의 출력 예시처럼 pid1 프로세스의 pid 와 이름이 제대로 출력되는 것을 확인할 수 있으며, 프로세스의 상태, 그룹 정보, context_swich 횟수, fork 호출 횟수, 부모 프로세스 정보, 형제자매 프로세스의 pid, 프로세스 이름, 프로세스 수, 자식 프로세스의 pid, 프로세스 이름, 프로세스 수가 정상적으로 출력되는 것을 확인할 수 있다.

3. 고찰

이번 과제를 통해 시스템프로그래밍 과제 때 사용하였던 fork, thread 의 개념을 상기시킬 수 있었으며 운영체제 시간에 배웠던 개념을 직접 코드로 구현해 보는 과정도 경험할 수 있었다. 우선 3-1 과제에서 당연히 thread 의 소요시간이 적게 나올 것이라고 예상했는데, 주변 친구들은 fork 의 실행시간이 더 적게 나온 경우도 적지 않아 신기했다. 코드의 구현 면에서 struct 나 동적 할당 등의 부분에서 시간을 더 소요하는 것 같았다. 또한 3-2

과제에서는 직접 스케줄러를 변경해보고 우선순위를 변경해보았지만, 결과가 확실하게 나오지 않아 아쉬웠다. 3-1 과 3-2 과제는 구현하는 것 보다 결과를 분석하고 이유에 대해 찾아보고 알아보는 것에 시간이 더 많이 소요됐던 것 같다. 하지만 그 과정에서 얻은 점도 많았다. 개념적으로만 알고 있던 부분도 헛갈리는 요소가 많았는데, 그러한 부분을 확실하게 할 수 있는 기회였다. 마지막 과제에서는 cscope 로 원형을 찾아보고 ;/검색어 기능과 다음 검색어로 넘기는 shift n 을 활용하여 하나하나 찾아보느라 시간이 많이 소요됐던 것 같다. 대부분의 것들이 task_struct 안에 있었으며, 찾은 것들로만 코드를 구현하여 결과를 뽑아낼 수 있었다는 점이 신기했다. 또한 주석을 잘 읽어보면 이곳에 새로운 것을 추가 해야한다. 이러한 것을 사용해야 한다 등과 같이 대부분의 주의사항과 사용법이 써져 있어 원형을 찾은 이후에는 더 수월했던 것 같다.

4. Reference

- 운영체제실습 강의자료 참조
- Status 값 / <https://80000coding.oopy.io/bd2fe002-d165-4bcb-b0c7-0efcfcec5949>
- Status 값 / <https://codetravel.tistory.com/30>
- 스케줄링 알고리즘 / https://blog.naver.com/alice_k106/221149061940