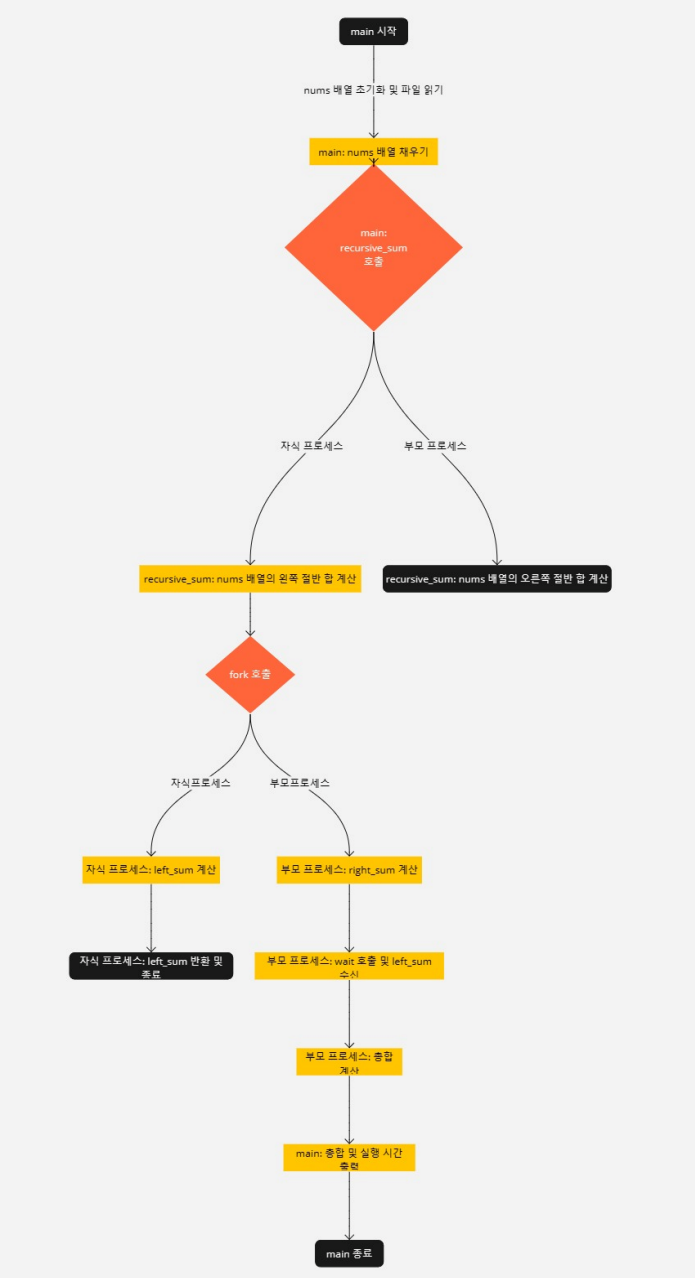
Assignment 3

과 목 명	운영체재
학 과	컴퓨터정보공학부
학 번	2020202096
성 명	우성원
교 수 님	김태석교수님
제 출 일	2023.10.31

- # Introduction

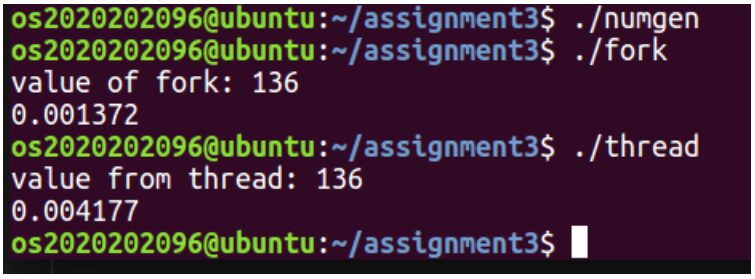
  이번 과제는 3개의 세부 과제로 이루어져있는데 먼저 과제 3_1은 max_process *2개의 순차적인 숫자를 fork를 통해 각 프로세스들이 읽어서 합산값을 반환하는 과제이다.max_process의 개수가 증가됨에 따라 프로세스 계층의 depth또한 늘어나며 이를 재귀적으로 구현하였다.또한 분할&정복을 통해 각각 배열의 왼쪽 오른쪽으로 나누어서 합산값을 부모프로세스와 자식프로세스가 계산하도록 하였다.과제 3_2는 각 스케쥴러의 정책과 우선순위에 따른 실행시간을 파악하는 과제이다. 먼저 filegen을 통해 temp파일에 랜덤한 숫자를 적고 이를 읽는 시간을 각 스케쥴러와 우선순위를 통해 파악하였다.스케쥴러 설정은 sched_setscheduler함수를 이용하였으며 우선순위는 스케쥴러에 맞게 nice와 priorty값을 조정하였다 마지막 3_3 커널의 sched.h에서 task_struct 원형에 새로운 fokr_count변수를 만들고 fork.c에서 task_struct에 새로 만든 fork_count를 초기화 및 사용하게끔하였다.또한 ftrace함수를 wrapping하여 인자로 받은 pid의 task_struct의 정보(fork횟수,pid 상태,부모프로세스,자식프로세스 등)를 출력하는 코드를 구현하였다.

- # Assignment 3-1

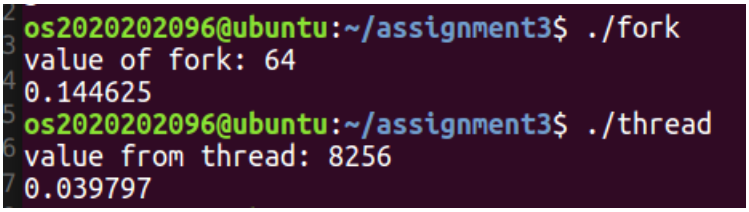


먼저 구현에 대해 설명하면 재귀적으로 fork를 하면서 부모프로세스,자식프로세스 모두 연산을 수행도록 구현하였다. 연산 수행 범위는 분할 정복을 통해 각각 왼쪽 오른쪽씩 숫자 배열을 분할하고 하나의 프로세서가

2개의 숫자를 읽을때까지 depth를 증가시키며 2개의 숫자를 읽을 때 재귀를 끝내 해당 숫자의 합산값을 호출한 프로세서에서 받아서 다시 합산시키도록 구현하였다.



먼저 maxprocess가 8일때의 결과이다. 총 1~16까지의 모든 수를 합산한 결과를 fork 와 thread둘다 잘 반환하는 것을 볼수있다.



Maxprocess가 64일 때 결과이다. 총 1~128의 모든 수를 합산하는 결과를 표출한다
Fork의 경우 exit()에 들어가는 인자가 255,즉 연산중 부모 프로세서에 넘기는 값이 8비트를 초과하기 때문에 modular연산에 의해 8비트 right shift되어 나머지를 합산한 결과가 나왔다.
Thread의 경우 그러한 과정이 없기 때문에 결과가 제대로 나오는 것을 확인할수있다.

-반환 값은 255 이상이면 안되며, 8-bit 만큼 right shift 해주어야 하는 이유

자식 프로세스에서 부모 프로세스로 값을 넘겨줄 때, exit() 함수를 사용해 반환 값을 지정한다. 이때 반환 값은 8비트를 초과하면 안 된다. 왜냐하면, 유닉스 기반 시스템에서 exit() 함수로 전달된 값은 부모 프로세스에게 8비트 정수로만 전달되기 때문이다.
exit() 함수로 전달된 값은 부모 프로세스에서 wait() 또는 waitpid() 함수를 통해 회수된다. 이 함수들은 자식 프로세스의 종료 상태를 나타내는 16비트 정수를 반환한다. 이 16비트 중 상위 8비트는 자식 프로세스의 종료 코드를 나타내고, 하위 8비트는 자식 프로세스가 비정상적으로 종료되었을 때의 신호 번호를 나타낸다.
따라서, 부모 프로세스가 자식 프로세스의 exit() 함수에서 설정한 반환 값을 올바르게 읽으려면, 반환된 16비트 정수를 8비트만큼 오른쪽으로 시프트(>> 8)해야 한다. 이렇게 하면 상위 8비트가 하위 8비트 위치로 이동하고, 부모 프로세스는 자식 프로세스가 exit() 함수를 통해 전달한 실제 값을 얻을 수 있다.

● Assignment 3-2

-Sched_FIFO 스케쥴러

Sched_FIFO 는 리눅스의 실시간 스케줄링 정책 중 하나이다. FIFO는 First-In-First-Out의 약자로, 먼저 들어온 프로세스가 먼저 처리된다는 원칙을 가진다. 이 정책은 특정 프로세스에게 CPU를 할당할 때, 그 프로세스가 종료되거나 블로킹 상태가 될 때까지 CPU를 계속 점유한다.

Sched_FIFO 정책을 사용하는 프로세스는 우선 순위(priority)를 가진다. 낮은 숫자의 우선 순위가 높은 숫자의 우선 순위보다 높다. 따라서 우선 순위가 높은 프로세스가 실행을 요청하면 우선 순위가 낮은 프로세스는 CPU를 내어주어야 한다. 하지만 Sched_FIFO 프로세스 내에서 같은 우선 순위를 가진 프로세스들 사이에서는, 먼저 CPU를 점유한 프로세스가 CPU를 계속 점유하며, 다른 프로세스는 기다리게 된다.

Sched_FIFO 스케쥴러는 Nice값을 사용하지않고 Priority값만 사용한다.아래 결과에 Nice값은 사용되지않는다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_FIFO highest
Policy: SCHED_FIFO, Priority: 99, Nice: -20
Elapsed time: 0.499460 seconds
```

먼저 가장 우선순위는 낮지만 우선순위 값이 큰 99를 설정했을때의 결과이다. 이는 우선순위가 가장 낮음을 의미한다. 0.49second정도로 나오는 것을 확인할수있다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_FIFO default
Policy: SCHED_FIFO, Priority: 50, Nice: 0
Elapsed time: 0.487780 seconds
```

그 다음 default 우선순위인 중간값인 50을 사용하였다. 0.487780 second로 우선순위가 높아지자 조금 더 실행시간이 빨라진 것을 확인할수있다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_FIFO lowest
Policy: SCHED_FIFO, Priority: 1, Nice: 19
Elapsed time: 0.481032 seconds
```

그 다음 가장 높은 우선순위인 1을 사용하였다. 0.481032 second로 셋 중 가장 빠른 실행시간을 보여준다.

-Sched_RR 스케쥴러

SCHED_RR은 실시간 스케줄링 정책 중 하나로, SCHED_FIFO의 라운드 로빈 버전이다. 이 정책은 여러 스레드가 동일한 우선 순위를 가질 때 효과적으로 사용된다. SCHED_RR에서도 SCHED_FIFO처럼 각 스레드에는 고정된 우선 순위가 주어진다. 스케줄러는 우선 순위가 높은 순서대로 스레드를 확인하고, 실행할 준비가 된 스레드 중 가장 우선 순위가 높은 스레드부터 실행한다. 하지만 SCHED_FIFO와는 다르게, 같은 우선 순위를 가진 스레드들은 정해진 시간 동안 순서대로, 즉 라운드 로빈 방식으로 실행된다.이런 방식은 동일 우선 순위의 스레드들 사이에서 공정한 CPU 시간 분배를 가능하게 하며, 한 스레드가 계속해서 CPU 자원을 독점하는 것을 방지한다. 따라서 SCHED_RR은 실시간 작업이지만, 여러 작업이 공평하게 처리될 필요가 있을 때 적합하다.

Sched_RR 스케쥴러는 Nice값을 사용하지않고 Priority값만 사용한다.아래 결과에 Nice값은 사용되지않는다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_RR highest
Policy: SCHED_RR, Priority: 99, Nice: -20
Elapsed time: 0.995170 seconds
```

먼저 가장 우선순위는 낮지만 우선순위 값이 큰 99를 설정했을때의 결과이다. 이는 우선순위가 가장 낮음을 의미한다. 0.995170second정도로 나오는 것을 확인할수있다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_RR default
Policy: SCHED_RR, Priority: 50, Nice: 0
Elapsed time: 0.732396 seconds
```

그 다음 default 우선순위인 중간값인 50을 사용하였다. 0.732396 second로 우선순위가 높아지자 조금 더 실행시간이 빨라진 것을 확인할수있다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_RR lowest
 Policy: SCHED_RR, Priority: 1, Nice: 19
Elapsed time: 0.543745 seconds
```

마지막으로 가장 높은 우선순위인 우선순위를 1로 설정하였을 때 결과이다.0.543745 seconds로 가장 빨라진 것을 확인할수있다.

-Sched_OTEHR 스케쥴러

SCHED_OTHER는 리눅스 운영체제의 기본 스케줄링 정책이다. 이 정책은 고정 우선순위가 아닌 타임쉐어링을 사용하여 프로세스들 사이에서 CPU 시간을 공유한다. 프로세스의 실행 우선순위는 nice 값에 따라 조정된다. 높은 nice 값은 낮은 우선순위를 의미하며, 낮은 nice 값은 높은 우선순위를 의미한다. SCHED_OTHER는 다른 실시간 스케줄링 정책에 비해 더 유연하며, 일반적인 사용자 레벨의 프로세스에 적합하다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_OTHER highest
Policy: SCHED_OTHER, Priority: 0, Nice: -20
Elapsed time: 0.268892 seconds
```

가장 높은 우선순위를 설정하게끔 nice값을 -20으로 설정하였을 때 결과이다.0.268892 seconds인 것을 확인할수있다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_OTHER default
Policy: SCHED_OTHER, Priority: 0, Nice: 0
Elapsed time: 0.388890 seconds
```

그 다음 default 우선순위인 중간값인 50을 사용하였다. 0.388890 second로 우선순위가 낮아지자 조금 느려진 것을 확인할수있다.

```
os2020202096@ubuntu:~/assignment3_2$ sudo ./schedtest SCHED_OTHER lowest
Policy: SCHED_OTHER, Priority: 0, Nice: 19
Elapsed time: 0.659675 seconds
```

먼저 가장 낮은 우선순위인 nice값을 19로 설정했을때의 결과이다. 0.659675 second로 우선순위가 가장 낮고 가장 느림을 확인할수있다.

Assignment 3-3

이 과제의 요구사항을 만족하기위해 다음과 같은 과정을 수행하였다.

먼저 sched.h에서 task_struct의 원형을 찾아 fork_count라는 int 변수를 하나 추가하였다.

```
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

    int fork_count;

    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long           state;

    /*
```

해당 변수를 사용하기위해 fork()를 호출시 작동되는 함수에 기능을 추가하였는데 fork.c의 copy_process() 함수에서 task_struct를 복사하여 새로운 프로세서(자식프로세서)에 할당하는 코드에 fork_count변수를 0으로 추가해주고 기존의 부모프로세서의 fork_count++해주는 방식으로 과제의 요구사항을 충족하였다.

```
    goto fork_out;

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

p->fork_count=0;

current->fork_count++;
```

이후 process_tracer는 기존의 ftrace를 wrapping하여 해당 ftrace의 인자로 넘겨받은 pid의 task_struct를 추적하여 해당 task_struct의 정보를 출력하는 코드를 작성하였다. 아래는 그 결과이다.

```
os2020202096@ubuntu:~/assignment3_3$ sudo insmod process_tracer.ko
[sudo] password for os2020202096:
os2020202096@ubuntu:~/assignment3_3$ lsmod | grep process
process_tracer         16384  0
os2020202096@ubuntu:~/assignment3_3$ ./test
Syscall invoked.
os2020202096@ubuntu:~/assignment3_3$ dmesg
```

모듈을 커널에 적재후 lsmod를 통해 적재가 되었음을 확인하였다.

```
[ 6653.054355] ##### START OF INFORMATION #####
[ 6653.054357] ##### TASK INFORMATION OF [1] systemd #####
[ 6653.054357] - task state : Wait
[ 6653.054358] - Process Group Leader : [1] systemd
[ 6653.054358] - Number of context switches : 6143
[ 6653.054359] - Number of calling fork() : 264
[ 6653.054359] - it's parent process : [0] swapper/0
[ 6653.054359] - it's sibling process(es) :
[ 6653.054360]   [2] kthreadd
[ 6653.054360] > This process has 1 sibling process(es)
[ 6653.054360] - Child process(es) :
[ 6653.054361]   [421] systemd-journal
[ 6653.054361]   [435] systemd-udevd
[ 6653.054362]   [494] vmware-vmblock-
[ 6653.054362]   [497] vmtoolsd
[ 6653.054363]   [524] systemd-timesyn
[ 6653.054363]   [951] accounts-daemon
[ 6653.054364]   [965] cron
[ 6653.054364]   [966] rsyslogd
[ 6653.054364]   [967] acpid
[ 6653.054365]   [971] dbus-daemon
[ 6653.054365]   [978] NetworkManager
[ 6653.054366]   [982] systemd-logind
[ 6653.054366]   [983] VGAuthService
[ 6653.054366]   [987] cupsd
[ 6653.054367]   [994] cups-browsed
[ 6653.054367]   [1008] polkitd
[ 6653.054368]   [1020] agetty
[ 6653.054368]   [1043] bluetoothd
[ 6653.054368]   [1057] irqbalance
[ 6653.054369]   [1086] lightdm
[ 6653.054369]   [1256] avahi-daemon
[ 6653.054370]   [1451] upowerd
[ 6653.054370]   [1459] rtkit-daemon
[ 6653.054370]   [1475] colord
[ 6653.054371]   [1504] whoopsie
[ 6653.054371]   [1525] systemd
[ 6653.054371]   [1532] gnome-keyring-d
[ 6653.054372]   [1984] udisksd
[ 6653.054372]   [2051] fwupd
[ 6653.054373] > This process has 29 child process(es)
[ 6653.054373] ##### END OF INFORMATION #####
os2020202096@ubuntu:~/assignment3_3$
```

이제 dmesg로 커널 메시지의 결과를 확인해보면 1번 프로세스 즉 systemd의 정보가 나오는 것을 확인할수있다.현재 상태는 wait이며 그룹리더는 자기자신,컨텍스트 스위칭횟수는 6143,fork 호출횟수는 264,부모 프로세스는 0번이며 하나의 형제프로세스,29개의 자식프로세스들이 있음을 확인할수있다.

```
os2020202096@ubuntu:~/assignment3_3$ sudo rmmod process_tracer.ko
```

이제 모듈을 언로드하였다.

```
os2020202096@ubuntu:~/assignment3_3$ lsmod | grep process
```

잘 제거가 된 모습이다.

고찰

먼저 3-1과제의 경우 재귀를 이용해야한다는 것을 알기가 쉽지않았다.아직 코딩테스트를 많이 풀어보지 않았기에 감이 안잡힌듯하다.하지만 depth가 증가해야한다는 사실을 깨달았고 이를 구현하기위한 최적의 방법은 재귀라는 생각이 들었다.이를 통해서 3-1과제를 해결할수있었다.3-3과제의 경우 커널의 어디에 fork.c에서 fork()함수가 실행될 때 호출되는 함수가 있는지 몰라서 이를 찾는데 참고자료의 도움을 받았다. 그리고 task_struct의 원형을 찾는 것 또한 쉽지않았다.하지만 copy_process()에서 tast_struct를 생성한다는 것을 알았고 이를 이용해서 새로운 변수인 fork_count를 task_struct에 추가하여 유저모드에서도 이용할수있도록 하였다.추후 많은 공부를 통해 커널 컨트리뷰트까지 하면 좋을것같다는 생각이 들었다.

참조

Process 생성 : 네이버 블로그 (naver.com)

[Linux Kernel] task_struct structure — Aiden (41d3n.xyz)