운영체제실습 보고서

Assignment 3

학 과 : 컴퓨터정보공학부

담당교수 : 김태석 교수님

학 번 : 2018202018

이 름 : 유승재

1. Introduction

이번 과제는 프로세스에 대해서 자세히 알아보고 프로세스와 쓰레드의 연관성 또는 차이점 등의 특징, 여러 프로세스를 한 번에 수행하게 될 때 프로세스 스케줄링에 대한 동작 원리와 이해, 그리고 직접 스케줄링을 바꿔가며 실행한 결과값의 차이를 비교하는 과정을 거칩니다. 마지막으로 프로세스가 각각 가지고 있는 task_struct에 대한 이해와 이를 직접 사용해 출력하는 것이 이번 과제입니다.

2. Result

1) Assignment 3-1

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$ ./fork
Value of fork: 136
0.007251
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$ ./thread
Value of thread : 136
0.001173
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$
```

Numgen을 통해 1~16까지의 정수를 입력한 temp.txt를 생성하고 각각 fork와 thread를 통해서 덧셈을 진행한 결과 화면입니다. Thread를 이용한 프로그램이 fork를 이용해 자식 프로세스에서 덧셈을 진행한 결과보다 더욱 빠른 결과를 보여주는 것을 알 수 있습니다.

이의 결과는 fork를 통한 자식 프로세스 연산은 프로세스 간의 overhead가 자주 발생하게 되는데 이는 thread간의 overhead보다 더 많은 cost를 발생하게 됩니다. 따라서 위의 결과는 올바른 결과라고 생각합니다.

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$ ./fork
Value of fork: 64
0.054313
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$ ./thread
Value of thread : 8256
0.013778
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$
```

위와 동일한 조건으로 numgen을 통해 temp.txt에 1~128의 정수를 입력했을 때의 결과입니다. 위와 동일하게 thread가 fork보다 더 빠른 처리를 하는 것을 확인할 수 있습니다.

하지만 fork로 출력한 결과값에 값의 손실이 있었다는 것을 확인할 수 있습니다.

이는 자식 프로세스가 종료 시에 값을 반환하는 방식에 있어 값의 손실을 야기할 수 있기 때문입니다.

자식프로세스에서 부모프로세스로 값을 넘겨줄 때 (exit ()) 반환 값은 2 8 이상이면 안되며, 8-bit 만큼 right shift 해주어야 child process가 반환된 값을 정상적으로 확인할 수 있음. (e.g. a >> 8) 이 이유는??

매크로	설명
WIFEXITED(status)	자식 프로세스가 정상적으로 종료되었다면 TRUE
WIFSIGNALED(status)	자식 프로세스가 시그널에 의해 종료되었다면 TRUE
WIFSTOPPED(status)	자식 프로세스가 중단되었다면 TRUE
WEXITSTATUS(status)	자식 프로세스가 정상 종료되었을 때 반환한 값

	8비트	8비트
정상 종료	프로세스 반환 값	0
비정상 종료	0	종료 시킨 시그널 번호

자식 프로세스가 종료될 경우에는 8비트(1바이트)로 이루어진 값을 반환하게 됩니다.

자식 프로세스는 정상적으로 종료되었을 경우 프로세스 반환 값을 16비트중 상위 8비트에 저장하고, 하위 8비트를 0으로 채워 넣습니다. 즉 프로세스 반환 값은 8비트의 정수로 이루어져 있습니다. 또한 이 값은 16비트중 상위 8비트에 저장이 되기 때문에 right shift >> 8 을 수행해주어야 위의 반환 값을 정확하게 가져올 수 있습니다.

WEXITSTATUS(status)는 상위 8비트의 값을 읽어오도록 하는 매크로입니다.

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$ ./fork
Value of fork: 64
0.054313
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$ ./thread
Value of thread : 8256
0.013778
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment31$
```

따라서 fork에서는 8비트로 표현할 수 있는 정수인 0~255 사이의 정수만을 출력할 수 있습니다. 위는 numgen를 통해 temp.txt에 1부터 128까지의 정수를 순서대로 입력되도록 만들었을 때의 fork와 thread의 실행 결과입니다.

Thread는 1부터 128의 합인 (128 * (128 + 1)) / 2 = 8256을 정확하게 출력한 것을 확인할 수

있었습니다. 반면 fork는 64라는 출력 결과를 보여주게 됩니다.

8256은 16진수로 나타내게 될 경우 0x2040으로 나타낼 수 있고, 이의 상위 8비트는 0x20으로 10진수로 나타낼 경우 32에 해당합니다. 하지만 64의 값을 반환하는 이유는 1부터 128의 덧셈을 각각 하나씩 진행하며 합이 255를 초과하는 결과를 반환하게 되는 경우 status % 255의 연산을 통해 값에 손실이 일어나게 되고 따라서 그 결과 8256의 상위 8비트인 32가 반환되는 것이 아닌 64가 반환된 것을 확인할 수 있었습니다.

2) Assignment 3-2

The standard round-robin time-sharing policy : nice 값을 사용해서 우선순위 변경

- Highest : -20
- Default : 0
- Lowest : 19

A first-in-first-out policy : 실시간 스케줄링 정책, 따라서 nice값을 사용하지 않고 priority를 변경하여 우선순위를 변경함

- Highest : sched_get_priority_max(SCHED_FIFO) == 99
- Default : sched_get_priority_max(SCHED_FIFO) + sched_get_priority_min(SCHED_FIFO) / 2
- Lowest : sched_get_priority_min(SCHED_FIFO) == 1

A round-robin policy : 실시간 스케줄링 정책, 따라서 nice값을 사용하지 않고 priority를 변경하여 우선순위를 변경함

- Highest : sched_get_priority_max(SCHED_RR) == 99
- Default : sched_get_priority_max(SCHED_RR) + sched_get_priority_min(SCHED_RR) / 2
- Lowest : sched_get_priority_min(SCHED_RR) == 1

The Standard round-robin time-sharing policy (SCHED_OTHER)

Nice : -20(max)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.429653
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

Nice : 0(default)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.235713
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

Nice : 19(min)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.364288
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

A first-in first-out policy (SCHED_FIFO)

Priority : 1(min)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.508112
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

Priority : 50(default)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.298533
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

Priority : 99(max)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.314370
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

A round-robin policy (SCHED_RR)

Priority : 1(min)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.582237
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

Priority : 50(default)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.384023
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

Priority : 99(max)

```
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$ sudo ./schedtest
3.707292
os2018202018@ubuntu:~/Downloads/linux-4.19.67/assignment32$
```

이번 테스트에서는 The Standard round-robin time-sharing policy (SCHED_OTHER)의 결과가 평균적인 결과로 가장 빠른 성능을 보여주는 것을 확인할 수 있었습니다. 하지만 많은 테스트를 거칠 경우 결과값의 변동이 크게 일어날 경우가 있고, 유효한 소요 시간의 속도 차이를 확인하기는 어려울 정도의 결과값이 출력되었습니다.

일반적으로 SCHED_FIFO는 프로세스의 cpu 점유 예상 시간을 예측하고 그 예측에 따라서 성능이 크게 변화할 수 있다고 생각합니다. 또한 SCHED_RR는 모든 프로세스를 공평하게 실행하는 구조입니다. 이는 공평한 방식이긴 하나 이번 테스트에 사용한 프로세스는 매우 간단한 작업을 수행하는 프로세스 10000개를 가지고 테스트를 하였습니다. 따라서 이번 테스트에는 SCHED_FIFO와 SCHED_RR이 크게 효율적이지 못한 테스트 방식이었다고 생각합니다. 또한 프로세스 하나의 처리 시간이 짧은 프로세스들을 묶어 테스트를 진행하였기 때문에 큰 차이를 확인하지 못하였던 것이라고 생각합니다.

3) Assignment 3-3

```
[ 6201.953323] ##### TASK INFORMATION of ''[1] systemd'' #####
[ 6201.953325] - task state : Wait
[ 6201.953325] - Process Group Leader : [1] systemd
[ 6201.953326] - Number of context switches: 4564
[ 6201.953327] - Number of calling fork() : 135
[ 6201.953327] - it's parent process: [0] swapper/0
[ 6201.953328] - it's sibling process(es) :
[ 6201.953329]         > [2] kthreadd
[ 6201.953329]         > This process has 1 sibling process(es)
[ 6201.953330] - it's child process(es) :
[ 6201.953330]         > [354] systemd-journal
[ 6201.953331]         > [385] vmware-vmblock-
[ 6201.953332]         > [390] systemd-udevd
[ 6201.953333]         > [410] vmtoolsd
[ 6201.953334]         > [511] systemd-timesyn
[ 6201.953335]         > [859] systemd-logind
[ 6201.953336]         > [861] bluetoothd
[ 6201.953337]         > [862] cron
[ 6201.953338]         > [864] rsyslogd
[ 6201.953338]         > [868] accounts-daemon
[ 6201.953339]         > [869] dbus-daemon
[ 6201.953340]         > [877] NetworkManager
[ 6201.953341]         > [880] VGAuthService
[ 6201.953342]         > [881] cupsd
[ 6201.953342]         > [885] acpid
[ 6201.953343]         > [888] avahi-daemon
[ 6201.953344]         > [892] snapd
[ 6201.953345]         > [932] irqbalance
[ 6201.953346]         > [959] cups-browsed
[ 6201.953347]         > [966] polkitd
[ 6201.953348]         > [973] lightdm
[ 6201.953348]         > [977] agetty
[ 6201.953349]         > [1161] upowerd
[ 6201.953350]         > [1182] rtkit-daemon
[ 6201.953351]         > [1223] colord
[ 6201.953352]         > [1365] whoopsie
[ 6201.953353]         > [1379] systemd
[ 6201.953354]         > [1429] gnome-keyring-d
[ 6201.953355]         > [1941] fwupd
[ 6201.953355]         > [1966] udisksd
[ 6201.953356]         > This process has 30 child process(es)
[ 6201.953356] ##### END OF INFORMATION #####
```

위의 결과는 test 프로그램에서 syscall(__NR_ftrace, 1)을 실행했을 때의 결과입니다.

인자로 들어온 pid에 해당하는 프로세스의 task_struct를 pid_task를 사용하여 찾았습니다. 그 후 해당 task_struct에서 원하는 정보들을 가져오는 방식으로 코드를 구현했습니다.

Context switch의 경우에는 비자발적인 문맥 교환과 자발적인 문맥 교환의 변수가 각각 저장되어 있습니다. 이번 과제에서 context switch가 의미하는 것은 모든 context switch의 횟수라

고 생각하여 이 두 변수를 합산한 결과를 출력하였습니다.

Fork의 횟수는 include/linux/sched.h의 파일에 정의되어 있는 task_struct에 fork_count 변수를 선언하였고, kernel/fork.c 파일에 선언되어 있는 _do_fork() 함수에 프로세스가 생성될 때 fork_count = 0으로 초기화하며 현재 프로세스가 새로운 프로세스를 fork할 때마다 fork_count를 증가시키는 방식으로 구현하였습니다. 그 후 인자로 받아온 프로세스의 pid에 해당하는 task_struct 구조체의 fork_count를 출력하여 fork의 횟수를 확인하는 방식으로 구현했습니다.

또한 위의 결과화면에서는 확인할 수 없지만 context switch와 fork의 횟수는 실행 시 마다 조금씩 변화하는 것을 확인했습니다. Context Switch의 경우에는 시스템에 실행 중인 다른 프로세스와 쓰레드의 현재 실행 상태, 그리고 스케줄링이 되는 방식에 따라서 조금씩 변화할 수 있기 때문에 큰 문제점이 아니라고 생각합니다. Fork 또한 마찬가지로 같은 프로세스가 동일하게 fork를 수행하더라도 스케줄러의 동작, 메모리 상태 등에 따라서 자식 프로세스가 생성되는 시점이나 순서가 달라질 수 있기 때문에 횟수가 변화할 수 있다고 보아 큰 문제점이 아니라고 생각합니다.

3. Consideration

이번 과제는 프로세스와 쓰레드의 성능 비교, 프로세스 스케줄링 정책을 바꾸며 프로세스 스케줄링 정책 처리 속도 비교, 프로세스의 task_struct 구조체를 이용하여 이 안에 저장되어 있는 해당하는 프로세스의 정보들을 출력해보는 세 가지의 과제를 진행하였습니다.

Assignment 3-1 : fork를 통해 자식 프로세스를 생성하고 코드를 수행하고 exit를 통해 결과값을 반환하는 과정과 thread에서 구조체를 이용하여 덧셈을 진행하고 결과값을 출력하는 과정을 구현하는 것에 어려움을 겪었습니다. 이전 시스템프로그래밍에서 공부했었던 과정이기 때문에 큰 어려움을 없었으나 자주 사용하지 않는 thread의 기능을 구현하는 과정에서 약간의 어려움을 겪었습니다. 또한 fork의 반환 값에 대해서 깊게 생각해본 적이 없었던 터라 8비트 이상의 결과값을 반환하게 될 때 발생하는 문제점에 대해서 알게 되었습니다.

Assignment 3-2 : 이번 과제를 진행하면서 각각의 Scheduling에 따라 어떠한 방식으로 Scheduling을 진행하는지에 대해서 명확하게 알게 되었지만 process 10000개의 개수가 큰 편차를 보여주기에는 너무 작은 숫자였다는 아쉬움이 남습니다. 이로 인해 process의 개수를 늘려 테스트를 진행해보려 하였으나 제 pc에서는 프로세스의 개수 제한이 13625로 제한이 걸려 있어 더 유의미한 수를 넣고 진행해보는 것이 불가능했습니다.

Assignment 3-3 : 각각의 프로세스가 가지고 있는 task_struct 구조체의 구조를 직접 코드를 보고 파악하고 task_struct를 직접 수정하여 새로운 값을 만들 수 있도록 하는 과정이 처음에는 쉽지 않았습니다. 커널 모드에서 동작하는 코드를 직접 수정한다는 것은 잘못 수정하는 과정을 거칠 경우에 큰 오류를 발생시킬 수 있었기 때문에 코드를 천천히 살펴본 후 수정을 하며 task_struct

에 대해서 더 자세히 알 수 있었습니다. Fork 코드 또한 직접 수정하면서 어떠한 방식으로 fork의 코드가 구현되었고 동작하는 지 알 수 있게 되었습니다.

4. Reference

https://badayak.com/entry/C%EC%96%B8%EC%96%B4-%EC%9E%90%EC%8B%9D-%ED%94%84%EB%A1%9C%EC%84%B8%EC%8A%A4-%EC%A2%85%EB%A3%8C-%ED%99%95%EC%9D%B8%EC%9D%84-%EC%9C%84%ED%95%9C-%EB%A7%A4%ED%81%AC%EB%A1%9C – 자식 프로세스 종료 확인 관련 문서