

운영체제실습

assignment 3

학 과: 컴퓨터정보공학부

담당교수: 김태석 교수님

학 번: 2019202050

성 명: 이강현

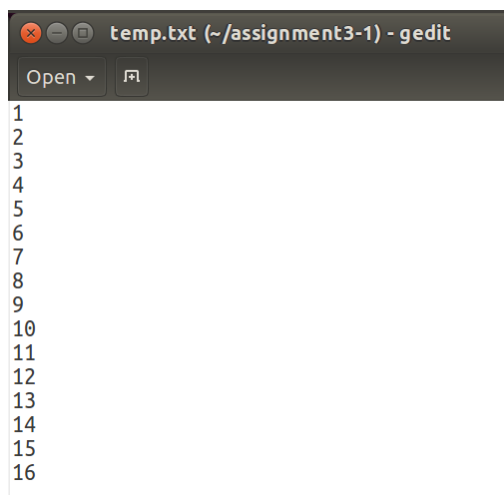
1. Introduction

fork 와 thread 로 이루어진 코드를 직접 구현해보면서 둘의 동작의 차이를 이해하고 성능을 비교해본다. cpu scheduling policy 에 대해 알아보고 각 policy 와 priority 가 다를 때 각각의 성능을 비교해보면서 scheduling 에 대한 학습을 진행한다. task_struct 를 이용해 프로세스 정보를 출력해보면서 PCB 에 대한 학습을 진행한다.

2. Result

<assignment 3-1>

메모장에 numgen 이라는 파일 생성 프로그램을 만들고 각각 MAX_PROCESSES 값을 8 그리고 64 일때로 나누어 성능을 테스트 해 보았다.

A screenshot of a gedit text editor window. The title bar shows 'temp.txt (~/.assignment3-1) - gedit'. Below the title bar is a toolbar with an 'Open' button and a file icon. The main text area shows line numbers from 1 to 16, with no text visible on the lines themselves.

<MAX_PROCESSES = 8>

위와 같이 정확히 프로세스 개수의 2 배만큼이 파일에 기록되어 있는 것을 확인할 수 있다.

fork 와 thread 둘 다 n 개의 프로세스 개수가 주어졌다면 반씩 쪼개어 merge-sort 알고리즘과 같이 $\log n$ 번 실행되게끔 for 문을 구성하고 그 안에 연산한 값을 다시 원래의 배열에 덮어쓰기 하게끔 구현하였다.

```
os2019202050@ubuntu:~/assignment3-1$ make clean
rm -rf tmp*
rm -f numgen fork thread
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2019202050:
3
os2019202050@ubuntu:~/assignment3-1$ make
gcc -Wall -pthread -lrt -o numgen numgen.c
gcc -Wall -pthread -lrt -o fork fork.c
gcc -Wall -pthread -lrt -o thread thread.c
os2019202050@ubuntu:~/assignment3-1$ ./numgen
os2019202050@ubuntu:~/assignment3-1$ ./fork
value of fork : 136
0.008786
os2019202050@ubuntu:~/assignment3-1$ ./thread
value of thread : 136
0.002726
os2019202050@ubuntu:~/assignment3-1$
```

프로세스 수를 8 로 하였을 때 각각 fork 와 thread 의 결과이다. fork 방식이 thread 방식보다 약 3 배정도 느린 것을 확인할 수 있고 이는 새로운 프로세스를 부모로부터 모두 복제하여 만들어내야 하는 fork 방식의 특성상 thread 방식보다 실행 시간에서 더 느릴 수 밖에 없음을 보여주는 결과라고 생각한다.

```

os2019202050@ubuntu:~/assignment3-1$ make clean
rm -rf tmp*
rm -f numgen fork thread
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2019202050@ubuntu:~/assignment3-1$ make
gcc -Wall -pthread -lrt -o numgen numgen.c
gcc -Wall -pthread -lrt -o fork fork.c
gcc -Wall -pthread -lrt -o thread thread.c
os2019202050@ubuntu:~/assignment3-1$ ./numgen
os2019202050@ubuntu:~/assignment3-1$ ./fork
value of fork : 64
0.024042
os2019202050@ubuntu:~/assignment3-1$ ./thread
value of thread : 8256
0.018266

```

두번째로 프로세스 수를 64 로 두었을 때의 결과이다. 마찬가지로 thread 가 fork 보다 더 빠른 실행시간을 가졌고 thread 로 실행하였을 때는 올바른 결과를 얻었지만 fork 방식으로 연산을 진행한 결과 이상한 값이 나왔음을 확인할 수 있다. 이는 반환값과 연관성이 있다. 부모 프로세스가 자식프로세스에서 연산한 값을 `exit()`함수를 통해 부모에게 반환하고 부모는 `wait()`를 이용해 반환 값을 얻는다. 그러나 `wait` 를 이용하여 받아오는 값은 다음과 같다.

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

따라서 정상종료에 해당하는 값은 8 비트 즉, 2 의 8 승을 넘어서는 안되고 그렇기에 fork 시 반환값을 이용하여 값을 전달하는 방식으로는 이 범위를 벗어나는 연산을 진행하게 되므로 프로세스 수가 64 일때 올바른 값을 얻어낼 수 없었다. 또한 위와 같은 구조로 값이 반환되기 때문에 얻어낸 값을 8 비트만큼 right shift 하여 실제값을 얻어낼 수 있다.

<assignment 3-2>

생성된 프로세스들이 열람할 파일들을 저장할 곳을 filegen 에서 mkdir 함수를 통해 생성할 수 있게끔 하고 이 곳에 프로세스 개수만큼 파일을 쓰게끔 구현하였다.

다음은 schedtest 에 대한 설명이다.

child process 를 10000 개 생성하여 각각 스케줄링을 변경하고 파일을 읽는 코드를 작성했다. 성능을 비교할 스케줄링 정책은 총 세가지이다.

1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR

SCHED_OTHER 는 비 실시간 정책으로 nice 값을 이용하여 우선순위를 결정하며 따라서 priority 는 0 으로 설정하여 sched_setscheduler 를 이용해 정책을 적용하였고 그 이후에 setpriority 함수를 이용해 nice 값을 highest 일때는 -20, default 일때는 0, lowest 일때는 19 으로 설정하여 우선순위를 부여했다.

그 외 SCHED_FIFO 와 SCHED_RR 방식은 실시간 정책으로 priority 값이 높을수록 더 높은 우선순위를 나타내므로 sched_setscheduler 를 이용해 highest 일 때 99, default 일 때 50, lowest 일때는 1 로 설정했다.

결과는 다음과 같다.

```
Standard Round Robin Time Sharing(Nice:-20): 0.263488
Standard Round Robin Time Sharing(Nice:0): 0.275685
Standard Round Robin Time Sharing(Nice:19): 0.248499
FIFO(Priority:1): 0.280687
FIFO(Priority:50): 0.281696
FIFO(Priority:99): 0.278496
Round Robin(Priority:1): 0.282225
Round Robin(Priority:50): 0.307516
Round Robin(Priority:99): 0.297599
```

아무래도 여러 변수에 따라 실행시간이 달라질 수 있기에 여러 번 실행을 해본 결과 SCHED_FIFO 와 SCHED_RR 방식은 우선순위를 설정하는 것에 대한 큰 차이가 발생하지 않았고 우열을 가리기 힘들었다. FIFO 는 들어온 순서에 따라 차례로 실행하기 때문에 큰 영향을 받지 않은 것이라고 예상되고 RR 또한 FIFO 와 마찬가지로 생각되었다. 게다가 I/O bound 라는 작업 특성상 CPU 를 많이 사용하지 않기에 CPU 선점시간이 그렇게 성능에 크게 작용하지 않았을 것으로 추측된다. 유일하게 SCHED_OTHER 방식에서의 우선순위 선정은 의미가 있었다. 반복적으로 실행한 결과 같은 결과를 보였기 때문이다. 하지만 예상과는 다르게 우선순위를 제일 낮춘 19 일 때 가장 실행시간이 짧은 즉, 가장 성능이 좋았다. 이러한 이유가 무엇인지에 대한 예상은 I/O bound 의 작업이므로 CPU 를 빠르게 더 많이 선점하게 되는 것의 의미가 없기에 오히려 우선순위를 낮추는 것이 CPU 를 더 많이 사용해야 하는 다른 프로세스들에게 적절히 분배될 수 있었고 이것으로부터 전체적인 성능과 안정성을 확보할 수 있지 않았나라는 생각이 들었다. 하지만 위에서 설명한 바와 마찬가지로 이러한 실행시간은 너무 많은 변수의 영향을 받고 또한 작업 케이스가 너무 한정적이라는 점에서 세가지 정책에 대한 우열을 가리기 힘들지 않나라는 결론을 가지게 되었다.

<assignment 3-3>

먼저 프로세스의 fork 호출횟수를 task_struct 에 그리고 fork 를 했을 때 값의 증가를 적절하게 수행하기 위해서 cscope 를 이용해 task_struct 와 fork 의 원형을 찾았다.

```
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  */
```

fork_count 라는 변수로 task_struct 에 새로운 필드를 추가한 모습이다.

```
2202 p = copy_process(clone_flags, stack_start, stack_size,
2203                 child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
2204 p->fork_count = 0;
2205 current->fork_count++;
2206 add_latent_entropy();
2207
```

fork.c 파일에서 do_fork 라는 부분을 찾아 부모 프로세스에서 자식 프로세스의 task_struct 를 복제하는 부분을 찾아 복제된 task_struct 의 추가한 fork_count 변수를 수정하는 부분이다. p 는 자식 프로세스의 task_struct 이니 0 으로 초기화 해주었고 current 는 현재 fork 를 호출한 대상 즉, 부모 프로세스의 task_struct 이니 1 증가해주는 모습이다.

커널 코드를 수정하였으니 커널을 컴파일을 진행하였다.

다음은 모듈에서 구조체 정보를 출력할 때 어떤 것을 참조해야 하는지 직접 찾아보았다.

```
69 /* Used in tsk->state: */
70 #define TASK_RUNNING                0x0000
71 #define TASK_INTERRUPTIBLE          0x0001
72 #define TASK_UNINTERRUPTIBLE        0x0002
73 #define __TASK_STOPPED              0x0004
74 #define __TASK_TRACED               0x0008
75 /* Used in tsk->exit_state: */
76 #define EXIT_DEAD                   0x0010
77 #define EXIT_ZOMBIE                 0x0020
78 #define EXIT_TRACE                   (EXIT_ZOMBIE | EXIT_DEAD)
79 /* Used in tsk->state again: */
80 #define TASK_PARKED                  0x0040
81 #define TASK_DEAD                   0x0080
82 #define TASK_WAKEKILL                0x0100
83 #define TASK_WAKING                  0x0200
84 #define TASK_NOLoad                  0x0400
85 #define TASK_NEW                     0x0800
86 #define TASK_STATE_MAX              0x1000
87
```

state 에 대한 정의가 되어있는 부분이다. state 를 읽어 위와 같은 정보를 가졌을 때 각각 상태를 출력해준다.

```
601      /* -1 unrunnable, 0 runnable, >0 stopped: */
602      volatile long          state;
603
```

구조체에는 위와 같이 변수가 정의되어 있기에 task->state 와 같이 불러와서 사용할 수 있었다.

```
841      * executable name, excluding path.
842      *
843      * - normally initialized setup_new_exec()
844      * - access it with [gs]et_task_comm()
845      * - lock it with task_lock()
846      */
847      char                    comm[TASK_COMM_LEN];
848
```

프로세스 정보는 comm 변수를 참조한다.

```
770      struct task_struct      *group_leader;
771
```

그룹 리더에 대한 정보는 구조체 포인터로 접근이 가능했고 이를 통해 pid, 프로세스 이름등을 출력할 수 있었다.

```
810      /* Context switch counts: */
811      unsigned long           nvcs;
812      unsigned long           nivcs;
813
```

context switch 의 횟수는 위와 같은 변수를 참조할 수 있다.

```
759      /* Real parent process: */
760      struct task_struct __rcu *real_parent;
761
762      /* Recipient of SIGCHLD, wait4() reports: */
763      struct task_struct __rcu *parent;
764
```

부모에 대한 정보는 실제 fork 를 한 부모를 나타내는 real_parent 와 SIGCHLD 를 받을 parent 가 있는데 real_parent 를 사용했다.

```
766      /* Children/sibling form the list of natural children:
767      */
768      struct list_head        children;
769      struct list_head        sibling;
770
```

children 과 같은 부모를 가진 sibling 은 list_head 형태로 존재하고 이를 list_for_each 매크로를 이용하여 순차접근하여 모두 출력하였다.


```

[20056.812543] ##### TASK INFORMATION of "[1] systemd" #####
[20056.812546] - task state : Wait
[20056.812546] - Process Group Leader : [1] systemd
[20056.812547] - Number of context switches : 4409
[20056.812548] - Number of calling fork() : 193
[20056.812549] - it's parent process : [0] swapper/0
[20056.812549] - it's sibling process(es) :
[20056.812550]   > [1] systemd
[20056.812551]   > [2] kthreadd
[20056.812551]   > This process has 2 sibling process(es)
[20056.812552] - it's child process(es) :
[20056.812553]   > [427] systemd-journal
[20056.812554]   > [442] systemd-udevd
[20056.812554]   > [469] vmware-vmblock-
[20056.812555]   > [508] vmtoolsd
[20056.812556]   > [517] systemd-timesyn
[20056.812557]   > [982] acpid
[20056.812558]   > [983] cron
[20056.812559]   > [990] cupsd
[20056.812559]   > [999] VGAuthService
[20056.812560]   > [1001] bluetoothd
[20056.812561]   > [1006] rsyslogd
[20056.812562]   > [1009] dbus-daemon
[20056.812563]   > [1048] cups-browsed
[20056.812564]   > [1051] NetworkManager
[20056.812565]   > [1057] accounts-daemon
[20056.812565]   > [1060] systemd-logind
[20056.812566]   > [1064] snapd
[20056.812567]   > [1085] agetty
[20056.812568]   > [1088] lightdm
[20056.812569]   > [1110] irqbalance
[20056.812570]   > [1116] polkitd
[20056.812571]   > [1304] avahi-daemon
[20056.812572]   > [1551] rtkit-daemon
[20056.812573]   > [1566] upowerd
[20056.812573]   > [1583] colord
[20056.812574]   > [1601] systemd
[20056.812575]   > [1608] gnome-keyring-d
[20056.812576]   > [1815] whoopsie
[20056.812577]   > [2087] udisksd
[20056.812578]   > [2170] fwupd
[20056.812579]   > This process has 30 child process(es)
[20056.812579] ##### END OF INFORMATION #####

```

pid 1 의 정보를 출력하니 위와 같이 올바르게 잘 출력하는 것을 확인할 수 있었다.

3. 고찰

먼저 fork 와 thread 의 코드를 짜면서 어려움이 가장 컸는데 그 이유는 이론은 기존에 배운 것을 바탕으로 하기에 쉽게 이해가 되었지만 구현 방식에 있어서 문제가 많았기 때문이었다. 먼저 재귀적으로 구현하려고 하였으나 그렇게 구현하니 두 코드간 구현의 차이가 많이 발생했다. 따라서 fork 방식과 thread 방식의 성능을 비교해야 한다는 과제 목적에 근거하면 fork 와 thread 를 제외한 나머지 코드가 동일해야 하는 변인 통제가 잘 이루어지지 않았고 이 때문에 오히려 fork 가 더 성능이

좋아지는 경우도 생겼다. 따라서 코드를 다시 수정하게 되었다. 3-2 에서도 어려움이 있었는데 단순히 스케줄링 정책을 변경하는 것은 단순했으나 각각의 성능차이가 나질 않았고 오히려 SCHED_OTHER 방식에서는 우선순위를 낮추는 것이 오히려 성능에 도움을 주어서 이를 어떻게 해석해야 하는지에 대한 어려움이 있었다. 3-3 에서는 task_struct 에 필드를 추가할 수 있고 함수를 적절히 수정해서 정보를 담을 수 있다는 점을 배웠다. 또 list_head 라는 구조체를 실습해볼 수 있어서 좋았다. 뭔가 확실하게 답이 나오지 않고 분석을 요하는 과제였어서 오히려 난이도에 비해 더 많은 시간을 쏟게 되었지만 그러한 과정에서 더 많이 찾아보고 더 많은 것을 이해할 수 있게 되었다.

4. Reference

운영체제실습 강의자료 참조