
Operating Systems

Assignment #2



Professor	김태석 교수님
Department	컴퓨터공학과
Student ID	2012722028
Name	장 한 별
Date	2018. 11. 12.

Introduction.

이번 과제는 thread 의 특징을 알고, 다중 thread 로 수행하는 프로그램을 작성한다.
2-1은 여러 개의 thread를 생성하여, 각각 file에 쓰여진 수를 읽고 곱셈 연산을 수행한다.
그 결과값을 다시 file의 마지막 줄에 써넣고, 기록된 모든 값을 연산하도록 구현한다.
2-2는 다중 thread 를 생성하여 linux cpu scheduling을 변경하여 수행 시간을
확인하는 것을 목표로 한다.

Reference.

[Assignment 2-1]

- 파일 입출력 <http://prosto.tistory.com/81>
- 난수: <https://edu.goorm.io/learn/lecture/201/%EB%B0%94%EB%A1%9C-%EC%8B%A4%ED%96%89%ED%95%B4%EB%B3%B4%EB%A9%B4%EC%84%9C-%EB%B0%B0%EC%9A%B0%EB%8A%94-c%EC%96%B8%EC%96%B4/lesson/12382/%EB%82%9C%EC%88%98-%EB%9E%9C%EB%8D%A4-%EB%A7%8C%EB%93%A4%EA%B8%B0>
- 파일 읽기 <https://shaeod.tistory.com/278>
- 시간 측정 http://tewda.blogspot.com/2015/02/clock_gettime.html
- thread, semaphore: 2018년 1학기 시스템프로그래밍 강의자료
- 13학번 최용락 학우에게 clock_gettime() 함수 활용 관련 도움을 받음

[Assignment 2-2]

- 스케줄링 <http://palpit.tistory.com/622>

Conclusion.

- Assignment 2-1

결과 확인

```
hanbyeol@hanbyeol:~/2-2$ make clean
rm -rf gen_file
rm -rf schedtest
rm -rf ./tmp1
rm -rf tmp*
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
3
hanbyeol@hanbyeol:~/2-2$
```

실험 전, 캐시 및 버퍼를 모두 비워서 항상 같은 조건으로 시작을 해야한다.

```
rm -rf tmp*
```

```
sync
```

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

위 3명령어는 Makefile에 넣어, make clean시 작동하도록 저장했다.

```
hanbyeol@hanbyeol:~/2-1$ make
gcc -o gen_file gen_file.c
gcc -o thread thread.c -pthread -lrt
```

Thread 생성을 위해 compile시 -pthread, 성능 측정을 위한 구조체나 함수를 이용하기 위해 -lrt를 추가해야한다.

```
hanbyeol@hanbyeol:~/2-1$ ls
gen_file gen_file.c Makefile thread thread.c
hanbyeol@hanbyeol:~/2-1$ ./gen_file
hanbyeol@hanbyeol:~/2-1$ ./thread
```

이렇게 생성된 실행파일 중 gen_file을 먼저 실행 시킨 후, thread를 실행시킨다.

```
thread id[4697]: mul: 20 * 16 = 320
thread id[4698]: mul: 4 * 9 = 36
thread id[4699]: mul: 13 * 19 = 247
thread id[4700]: mul: 4 * 8 = 32
thread id[4701]: mul: 2 * 3 = 6
thread id[4702]: mul: 13 * 13 = 169
thread id[4703]: mul: 20 * 15 = 300
thread id[4704]: mul: 15 * 21 = 315
```

그 후, 결과가 terminal에 출력된 것을 확인할 수 있다. 이때, 곱셈 연산 후 나온 result를 확인해보면, 320, 36, 247, 32... 순으로 나오는 것을 확인할 수 있다. 이는 새롭게 연산된 결과 이므로 기존의 file에 마지막 줄에 추가되어 다시 곱셈 연산이 된다.

```
-----
thread id[4825]: mul: 320 * 36 = 11520
thread id[4826]: mul: 247 * 32 = 7904
thread id[4827]: mul: 6 * 169 = 1014
thread id[4828]: mul: 300 * 315 = 94500
thread id[4829]: mul: 22 * 39 = 858
thread id[4830]: mul: 60 * 40 = 2400
```

위 사진을 통해 그 사실을 확인할 수 있다. 320 * 36, 247 * 32... 순으로 출력되는 것을 확인할 수 있다. 위와 마찬가지로 그 결과 값인 11520, 7904 는 다시 file에 마지막 줄에 추가된다.

<pre>----- thread id[4889]: mul: 11520 * 7904 = 91054080 thread id[4890]: mul: 1014 * 94500 = 95823000 thread id[4891]: mul: 858 * 2880 = 2471040 thread id[4892]: mul: 57222 * 1248 = 71412056 -----</pre>	
올바르게 진행되고 있음을 확인할 수 있다.	
<pre>thread id[4934]: mul: 64022400 * 37537920 = 2403267729408000 thread id[4935]: mul: 221205600 * 20487168 = 4531876289740800 thread id[4936]: mul: 2822400 * 15113952 = 42657618124800 ----- thread id[4937]: mul: 2147483647 * 2147483647 = 4611686014132420609 thread id[4938]: mul: 2147483647 * 2147483647 = 4611686014132420609 thread id[4939]: mul: 2147483647 * 2147483647 = 4611686014132420609 -----</pre>	
연산자, 피연산자, 결과값의 변수를 모두 unsigned long long int로 선언했다. 하지만, 위 사진을 보면 표현 한계에 도달하여 overflow가 발생했음을 확인할 수 있다.	
<pre>----- thread id[6065]: mul: 2147483647 * 2147483647 = 4611686014132420609 Run time : 0.78397645 s hanbyeol@hanbyeol:~/2-1\$ █</pre>	
필자는 #define MAX_PROCESSES 128 로 설정했다. 128 의 2배인 256 개의 난수가 temp.txt 에 write 되었고, 각 thread가 2개의 숫자를 read 하여, 결과 값을 다시 temp.txt 로 write 한다. 그럼 256개의 절반인 128개로 줄어 들었고, 다시 이 결과값을 연산하면 1/2 가 된다. 이런식으로 결과값 이 하나가 될 때 까지 수행하고, 위 그림을 통해 끝까지 모두 완료되었음을 확인할 수 있다.	
Running time	
<pre>thread id[6055]: mul: 2147483647 * 2147483647 = 4611686014132420609 Run time : 0.56434602 s hanbyeol@hanbyeol:~/2-1\$ █</pre>	<pre>Run time : 0.79149050 s hanbyeol@hanbyeol:~/2-1\$ █</pre>
<pre>Run time : 0.64425756 s hanbyeol@hanbyeol:~/2-1\$ █</pre>	<pre>Run time : 0.78287790 s hanbyeol@hanbyeol:~/2-1\$ █</pre>
<pre>Run time : 0.78035882 s hanbyeol@hanbyeol:~/2-1\$ █</pre>	<p>5번의 연산 결과 :</p> <p>0.56s, 0.79s, 0.64s, 0.78s, 0.78s</p> <p>평균 :</p> <p>0.71 s</p>
temp.txt	

14	28
2	63
9	45
7	49
15	55
3	84
7	49
7	252
5	210
11	81
21	

위 그림은 cat 명령어로 temp.txt 를 출력한 화면의 일부분이다. 14, 2, 9, 7... 순으로 난수가 저장되었음을 확인할 수 있고, 그 결과값인 28, 63.. 이 올바르게 저장되었음을 확인할 수 있다.

● Assignment 2-2

결과 화면

```
hanbyeol@hanbyeol:~/2-2$ ls
gen_file gen_file.c Makefile schedtest schedtest.c
hanbyeol@hanbyeol:~/2-2$ ./gen_file
hanbyeol@hanbyeol:~/2-2$ ls
gen_file gen_file.c Makefile schedtest schedtest.c tmp1
hanbyeol@hanbyeol:~/2-2$ cd tmp1
```

위 그림은 실행파일 gen_file을 실행시킨 후 화면이다. 그 결과 tmp1 directory가 생성되었음을 확인할 수 있다.

```
12101.txt 14209.txt 16316.txt 3678.txt 5785.txt 7892.txt 999.txt
12102.txt 1420.txt 16317.txt 3679.txt 5786.txt 7893.txt 99.txt
12103.txt 14210.txt 16318.txt 367.txt 5787.txt 7894.txt 9.txt
12104.txt 14211.txt 16319.txt 3680.txt 5788.txt 7895.txt
12105.txt 14212.txt 1631.txt 3681.txt 5789.txt 7896.txt
12106.txt 14213.txt 16320.txt 3682.txt 578.txt 7897.txt
hanbyeol@hanbyeol:~/2-2/tmp1$
```

tmp1 directory는 MAX_PROCESSES 만큼 생성한 text 파일들이 담겨있다. 이 text 파일들에는 난수가 적혀있다. 필자는 #define MAX_PROCESSES 16384 로 설정했다.

SCHED_OTHER

hanbyeol@hanbyeol:~\$./gen_f	hanbyeol@hanbyeol:~/2-2\$./gen_f	hanbyeol@hanbyeol:~/2-2\$./gen_f
hanbyeol@hanbyeol:~\$./sched	hanbyeol@hanbyeol:~/2-2\$./sched	hanbyeol@hanbyeol:~/2-2\$./sched
Input priority:	Input priority:	Input priority:
1	1	1
[SCHED_OTHER]	[SCHED_OTHER]	[SCHED_OTHER]
Run time : 3.382479	Run time : 3.4288255900 s	Run time : 2.593330803
hanbyeol@hanbyeol:~\$	hanbyeol@hanbyeol:~/2-2\$	hanbyeol@hanbyeol:~/2-2\$

3번의 연산 결과 :

3.38s, 3.42s, 2.59s,

평균 :

3.13 s

hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ Input priority: 5 [SCHED_OTHER] Run time : 3.412596 s hanbyeol@hanbyeol:~/2-2\$	hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ Input priority: 5 [SCHED_OTHER] Run time : 3.4235417863 s hanbyeol@hanbyeol:~/2-2\$	hanbyeol@hanbyeol:~/2-2\$./gen ./hanbyeol@hanbyeol:~/2-2\$./: Input priority: 5 [SCHED_OTHER] Run time : 3.4145218811 s hanbyeol@hanbyeol:~/2-2\$
---	---	---

3번의 연산 결과 :

3.41s, 3.42s, 3.41s,

평균 :

3.41 s

SCHED_FIFO

hanbyeol@hanbyeol:~/2-2\$ Input priority: 1 [SCHED_FIFO] Run time : 6.97668220 s hanbyeol@hanbyeol:~/2-2\$ ls	hanbyeol@hanbyeol:~/2-2\$ Input priority: 1 [SCHED_FIFO] Run time : 11.71745786 s hanbyeol@hanbyeol:~/2-2\$	hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ Input priority: 1 [SCHED_FIFO] Run time : 4.40043144 s hanbyeol@hanbyeol:~/2-2\$
--	--	--

3번의 연산 결과 :

6.97s, 11.7s, 4.4s,

평균 :

7.69 s

hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ Input priority: 5 [SCHED_FIFO] Run time : 4.3881917131 s hanbyeol@hanbyeol:~/2-2\$	hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ s Input priority: 5 [SCHED_FIFO] Run time : 4.4050676995 s hanbyeol@hanbyeol:~/2-2\$	hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ Input priority: 5 [SCHED_FIFO] Run time : 3.57921974 s hanbyeol@hanbyeol:~/2-2\$
--	--	--

3번의 연산 결과 :

4.38s, 4.4s, 3.5s,

평균 :

4.09 s

SCHED_RR

<pre>hanbyeol@hanbyeol:~/2- hanbyeol@hanbyeol:~/2- Input priority: 1 [SCHED_RR] Run time : 3.401832051 hanbyeol@hanbyeol:~/2-</pre>	<pre>hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ Input priority: 1 [SCHED_RR] Run time : 3.184507314 s hanbyeol@hanbyeol:~/2-2\$</pre>	<pre>hanbyeol@hanbyeol:~/2-2\$ Input priority: 1 [SCHED_OTHER] Run time : 3.390911284 s hanbyeol@hanbyeol:~/2-2\$</pre>
<p>3번의 연산 결과 :</p> <p>3.40s, 3.18s, 3.3s,</p> <p>평균 :</p> <p>3.29 s</p>		
<pre>hanbyeol@hanbyeol:~/2- hanbyeol@hanbyeol:~/2- Input priority: 5 [SCHED_RR] Run time : 9.380755788 hanbyeol@hanbyeol:~/2-</pre>	<pre>hanbyeol@hanbyeol:~/2-2\$ hanbyeol@hanbyeol:~/2-2\$ Input priority: 5 [SCHED_RR] Run time : 4.3938815448 s hanbyeol@hanbyeol:~/2-2\$</pre>	<pre>hanbyeol@hanbyeol:~/2-2\$. hanbyeol@hanbyeol:~/2-2\$ s Input priority: 5 [SCHED_RR] Run time : 6.4115521193 s hanbyeol@hanbyeol:~/2-2\$</pre>
<p>3번의 연산 결과 :</p> <p>9.38s, 4.39s, 6.41s,</p> <p>평균 :</p> <p>6.72 s</p>		
<p>Priority 수가 커질수록 우선순위는 낮아 진다. 우선 각각의 스케줄링 기법에 따른 실행시간을 비교하면, other 방식의 성능이 가장 뛰어났다.</p> <p>이론적으로, time-slice를 사용하는 SCHED_OTHER와 SCHED_RR이 time sharing을 사용하지 않는 SCHED_FIFO 보다 전체 실행 시간이 더 길어야 한다. 이유는 Time sharing 방식을 사용하기 때문에 I/O 요청 후 time slice 시간이 만료 되었을 때 문맥 교환이 이뤄진 다. 이 문맥 교환 시간은 순수 overhead 시간이기 때문에 FIFO 방식보다 시간이 더 걸리게 된다.</p> <p>< 성능 비교 시 MAX_PROCESSES 수는 16384개로 설정. ></p> <p>CPU Time이 SCHED_OTHER 방식일 때 가장 적게 나왔다. 이 의미는 가장 CPU를 비효율적으로 사용하였다는 의미이다. FIFO 방식과 RR 방식을 효율적으로 사용하지 못하여, 위의 전체 실행 시간도 더 오래 걸림을 알 수 있었다.</p> <p>[Priority 설정] 또한, RR 방식과 FIFO 방식은 Policy 변경 전, priority를 설정을 해주어야 하는데 이 경우 RR 방식과 FIFO 방식의 priority 값을 같게 설정해 줌으로써 Policy 만의 비교를 진행하였다.</p>		

[고찰]

이번 과제는 멀티 thread 의 생성 및 처리방법과 다양한 CPU 스케줄링의 성능을 비교했다.

먼저 2-1 gen_file.c과 thread.c 을 코딩하는 것이 가장 오랜 시간이 걸린 것 같다. 파일 입출력에 대한 문법과 시스템 프로그래밍 강의 때 배운 thread와 동기화를 위한 semaphore의 개념과 문법들을 모두 잊어버린 바람에 이 모든 것들을 다시 처음부터 공부한 후에 시작해야 했기 때문이다. 참고자료들을 참고하면서, 차분하고 확실하게 습득해 나갔다. 알고리즘상에 문제는 크게 없었다. Thread의 수가 file에 쓰여지는 수에 영향을 끼쳤기 때문에 이 수를 thread가 생성될 때 호출하는 함수인 thr_fn에 넘겨주는 것이 중요했는데 이는 전역 변수로 선언하여 해결했다.

2-1 에서 파일 입출력에 대한 복습을 미리 했기 때문에 2-2 의 gen_file.c 는 큰 문제없이 해결했다. 다만 경로를 설정하는 변수를 새롭게 추가하여 sprintf() 함수에 넣어 한 단계를 더 거쳐 file 을 생성하는 방법은 참고자료를 참고하여 해결했다.

이번 과제를 통해 멀티 thread의 중요성을 파악했고, 중간고사에도 출제되었던 다양한 CPU 스케줄링의 중요성에 대해 다시 한번 짚고 넘어가는 계기가 되었다.