로고이(가) 표시된 사진

자동 생성된 설명

운영체제

Assignment 3 과제

수업 명 : 운영체제

과제 이름 : assignment3

담당 교수님 : 최상호 교수님

학 번 : 2019202005

이 름 : 남종식

**Introduction**

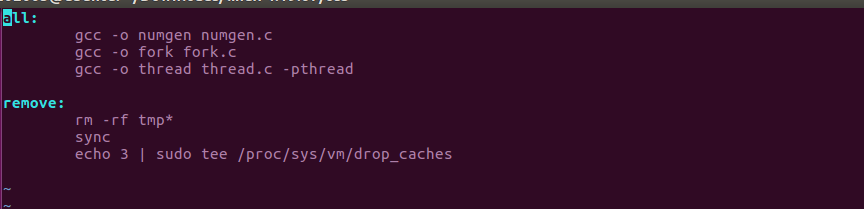
이번 과제는 총 3가지의 과제로 이루어져 있으며 먼저 3-1 과제는 numgen.c를 통해 temp.txt에 1부터 하나씩 증가하여 프로세스 수 2배까지 기록을 하는 것입니다. MAX\_PROCESS의 두배만큼의 숫자를 생성해 fork.c와 thread.c를 통해 최상단부터 process/thread마다 2개의 숫자의 값을 읽고 덧셈 연산을 진행합니다. 최종적으로 나온 값과 전체 프로그램의 수행시간도 측정합니다.

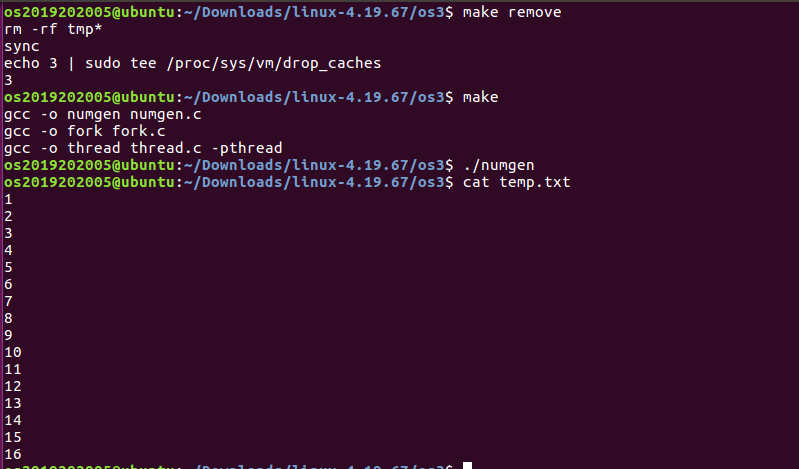
다음은 3-2과제입니다. 각 프로세스에서 CPU스케쥴링 정책을 변경하는 과제입니다. 일단 temp.txt를 MAX\_PROCESSES의 수만큼 생성하여 무작위로 9보다 작거나 같은 interger형 양수를 저장합니다. 이를 filegen.c에서 실행하고 schedtest.c에서는 일단 fork를 통해 MAX\_PROCESSES만큼의 프로세스를 생성합니다. 그리고 cpu scheduling 정책을 변경합니다. 그 후 각 i번째 프로세스에 만들어져 있는 파일에서 저장되어 있는 값을 읽어옵니다. 이때 테스트할 cpu scheduling으로는 The standard round-robin time-sharing policy, first-in first-out policy, round robin등이 있습니다. 각 스케쥴링 정책과 priority, nice값을 설정한 후 이에 대한 수행시간 차이를 비교하는 과제입니다.

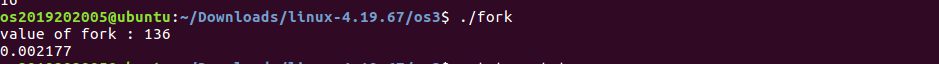
마지막으로 3-3과제입니다. PID를 바탕으로 8가지의 프로세스 정보를 출력하는 module을 작성하는 과제입니다. 프로세스 이름, 현재 프로세스 상태, 프로세스 다음은 출력해야 할 8가지 정보입니다. 그룹 정보, 해당 프로세스를 실행하기 위해 수행된 context switching횟수, fork()호출한 횟수, 부모 프로세스 정보, 형제자매 프로세스 정보, 자식 프로세스 정보가 있습니다. 2차 과제에서 작성한 ftrace 시스템 콜(336번)을 다음 함수로 wrapping 하여 사용해야합니다.

**Result**

**3-1과제**

 먼저 매 실험 전에 캐시 및 버퍼를 비워서 실험에 영향을 주는 요소를 제거해줘야 하기 때문에 명령어 make remove를 통해 지울 수 있도록 Makefile에 추가하였습니다. 그리고 numgen.c, fork.c , thread.c 을 모두 컴파일 하도록 작성하였습니다.

 먼저 MAX\_PROCESS의 수 가 8일 때 상황입니다. Numgen,c를 통해 temp.txt를 생성하여 이 파일에 1부터 MAX\_PROCESS\*2인 16까지의 숫자를 순서대로 저장하였습니다. 파일을 실행 후 cat temp.txt를 통해 숫자가 잘 저장된 모습을 확인할 수 있습니다.

 다음은 fork.c를 실행한 결과입니다. 일단 최종적으로 나온 값과 전체 프로그램 수행시간이 출력된 모습을 확인할 수 있습니다. fork.c와 thread.c에서 덧셈을 진행할 때 각 숫자들을 읽은 후 덧셈을 진행하여 이를 부모 프로세스와 스레드에게 값을 전달합니다. 그 후 최종적으로 하나의 값을 가지게 됩니다.

그리고 수행시간을 계산하기 위해서는 clock\_gettime()함수를 사용하여 끝난 시간에서 처음 시작한 시간을 뺀 값으로 구할 수 있었습니다.

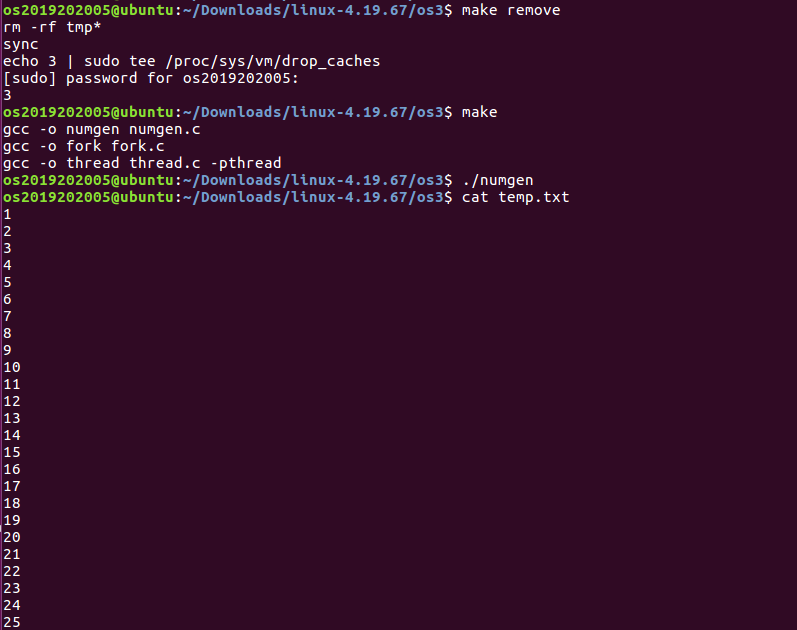


다음은 thread.c를 실행한 결과입니다. 덧셈을 통해 최종적으로 나온 값은 fork.c를 실행했을 때와 같은 값이 나온 것을 확인할 수 있으며 수행시간은 더 짧게 측정된 것을 확인할 수 있습니다. 이를 통해 thread로 실행했을 때 더 빠른 것을 알 수 있으며 이유를 생각해 보았을 때 thread를 생성해 실행했을 때는 code, data file영역을 등을 공유하기 때문에 모든 영역을 복사하는 fork보다 더 시간이 적게 걸린 것이라고 생각합니다. 따라서, thread는 메모리를 공유하여 실행하기 때문에 실행시간이 더 빠르다는 것이라고 생각합니다.



다음은 2개씩 숫자를 덧셈한 결과입니다. temp.txt에 저장된 모습을 확인할 수 있습니다.

다음은 MAX\_PROCESS가 64일 때 실행한 결과입니다.





실행과정은 MAX\_PROCESS가 8일 때와 똑같은 방식으로 진행했습니다. 먼저 캐시 및 버퍼를 비워서 실험에 영향을 주는 요소를 제거하고 make후 numgen.c를 실행시켜 temp.txt에 MAX\_PROCESS\*2의 값인 128까지 숫자가 잘 저장되었는 지 확인하였습니다.



위 두 사진을 비교했을 때 fork.c를 실행했을 때와 thread.c를 실행했을 때 최종적으로 나온 값이 서로 다른 것을 확인할 수 있으며 thread.c를 실행했을 때 정확한 값이 나온 것을 알 수 있습니다. 그리고 수행시간은 역시 thread로 실행했을 때 더 빠른 것을 확인할 수 있습니다.

값이 잘못 출력된 이유를 생각해보자면, 부모 프로세스가 자식 프로세스를 fork() 함수를 사용하여 생성하면, 부모와 자식 프로세스 간에 상태 및 반환 값을 전달하기 위한 프로세스 종료 상태를 관리하는데 16비트가 사용됩니다. 이 16비트를 나누어서 사용하는데, 상위 8비트는 자식 프로세스가 exit() 함수로 반환한 값이 저장되고, 하위 8비트는 부모 프로세스가 자식 프로세스의 종료 상태 status값을 저장합니다. status 값은 자식 프로세스가 종료될 때 exit() 함수를 통해 전달됩니다. 반환 값은 해당 프로세스가 종료될 때 exit() 함수에 전달한 값입니다.

이러한 구조 때문에, 반환 값은 하위 8비트의 제한을 받으므로 0부터 255까지의 값을 가질 수 있습니다. 만약 반환 값이 256 이상이라면 하위 8비트만 사용되므로, 결과값이 잘릴 수 있습니다. MAX\_PROCESS가 8일 때는 결과 값이 255 이하이기 때문에 정상적으로 값이 나온 것이고, 64일 때는 8256으로 255을 넘기 때문에 정확한 값이 나오지 않은 것입니다.

**3-2과제**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3-1과제와 동일하게 명령어 make remove를 통해 캐시 및 버퍼를 지울 수 있게 하였고 filegen.c와 schedtest.c를 모두 컴파일할 수 있도록 하였습니다.

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

해당 디렉토리 안에 temp디렉토리가 생긴 것을 확인할 수 있습니다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명 MAX\_PROCESSES가 10000이므로 0부터 9999까지 총 10000개의 txt파일이 생긴 것을 확인할 수 있습니다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명   텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

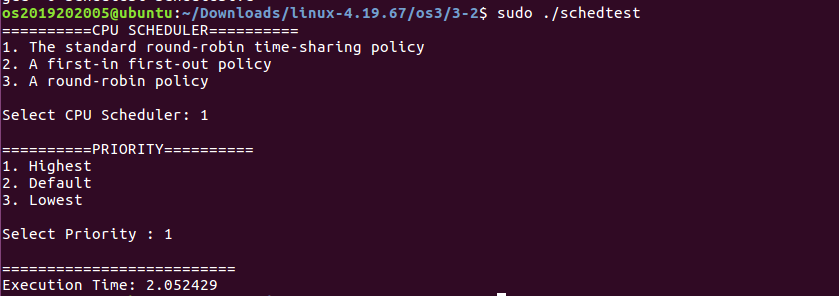
텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명스크린샷, 텍스트, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

각 파일 안에 9이하의 값이 잘 저장된 모습을 확인할 수 있습니다.

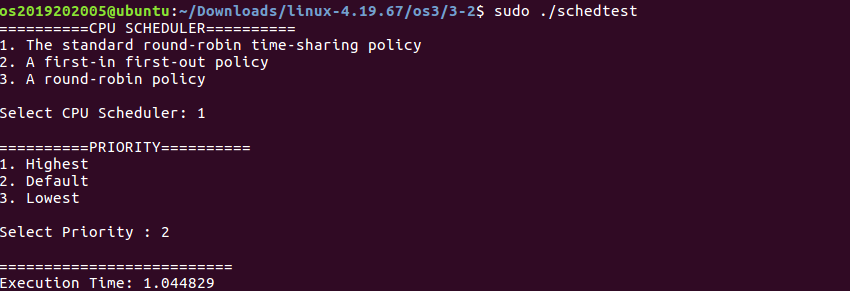
**The standard round-robin time-sharing policy & Highest**



일단 schedtest.c를 실행시키면 CPU스케줄러 3개를 선택할 수 있게 했습니다. 3개의 알고리즘 중 하나를 선택하면 우선순위를 선택할 수 있게 했습니다. The standard round-robin time-sharing policy는 nice값을 통해 우선순위를 매겼으며 Highest는 -20, Default는 0그리고 Lowest는 19로 설정하였습니다.

위 사진에서 우선순위가 Highest일 때 실행시간을 확인할 수 있습니다.

**The standard round-robin time-sharing policy & Default**



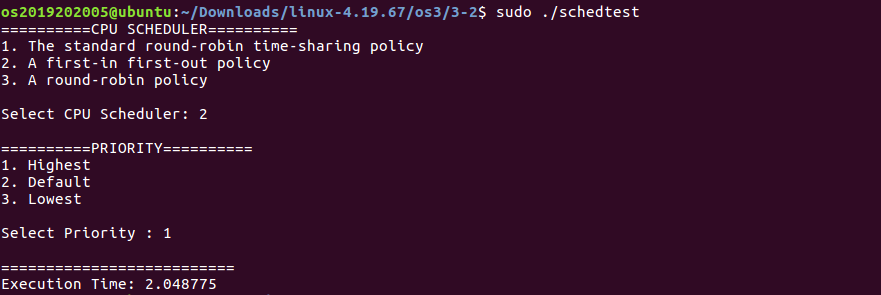
다음은 우선순위가 Default일 때 실행시간을 확인할 수 있습니다.

**The standard round-robin time-sharing policy & Lowest**



다음은 우선순위가 Lowest일 때 실행시간을 확인할 수 있습니다.

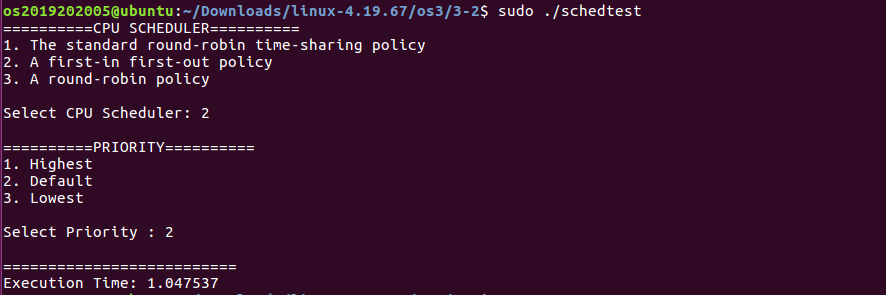
**A first-in first-out policy & Highest**



A first-in first-out policy는 Real time priority에 따라 우선순위를 나눴으며 Highest는 99, Default는 50, Lowest는 1로 설정했습니다.

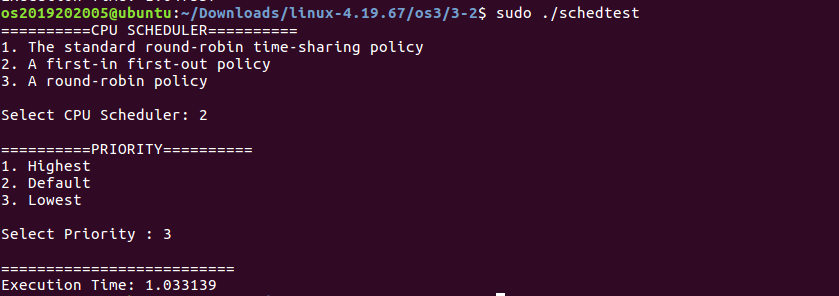
위 사진에서 우선순위가 Highest일 때 실행시간을 확인할 수 있습니다.

**A first-in first-out policy & Default**



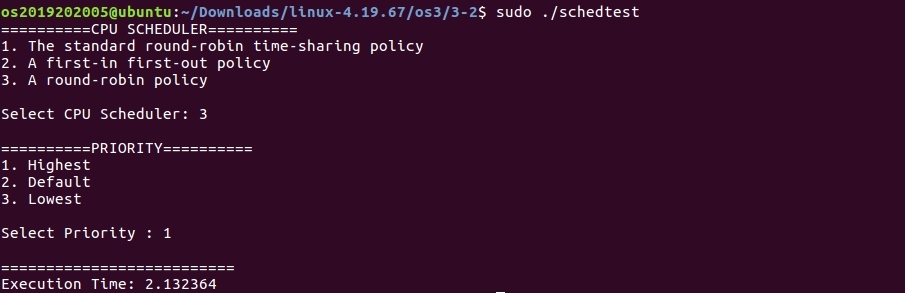
다음은 우선순위가 Default일 때 실행시간을 확인할 수 있습니다.

**A first-in first-out policy & Lowest**



다음은 우선순위가 Lowest일 때 실행시간을 확인할 수 있습니다.

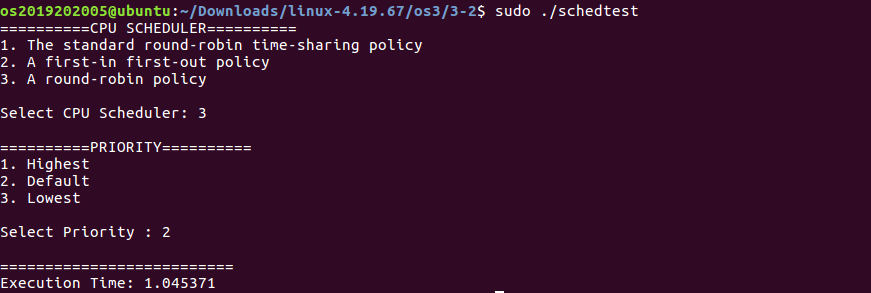
**A round-robin policy & Highest**



A round-robin policy또한 fifo와 동일하게 Real time priority에 따라 우선순위를 나눴으며 Highest는 99, Default는 50, Lowest는 1로 설정했습니다.

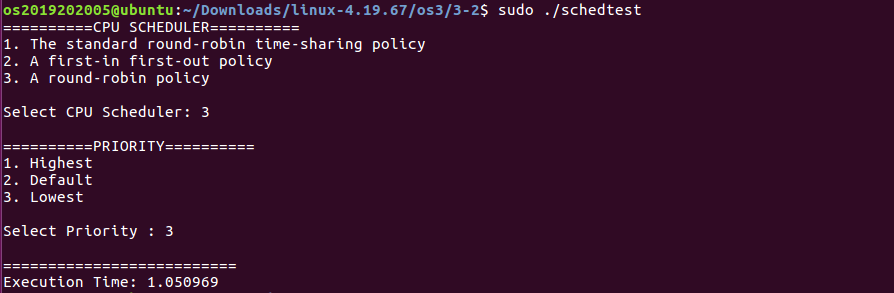
위 사진에서 우선순위가 Highest일 때 실행시간을 확인할 수 있습니다.

**A round-robin policy & Default**



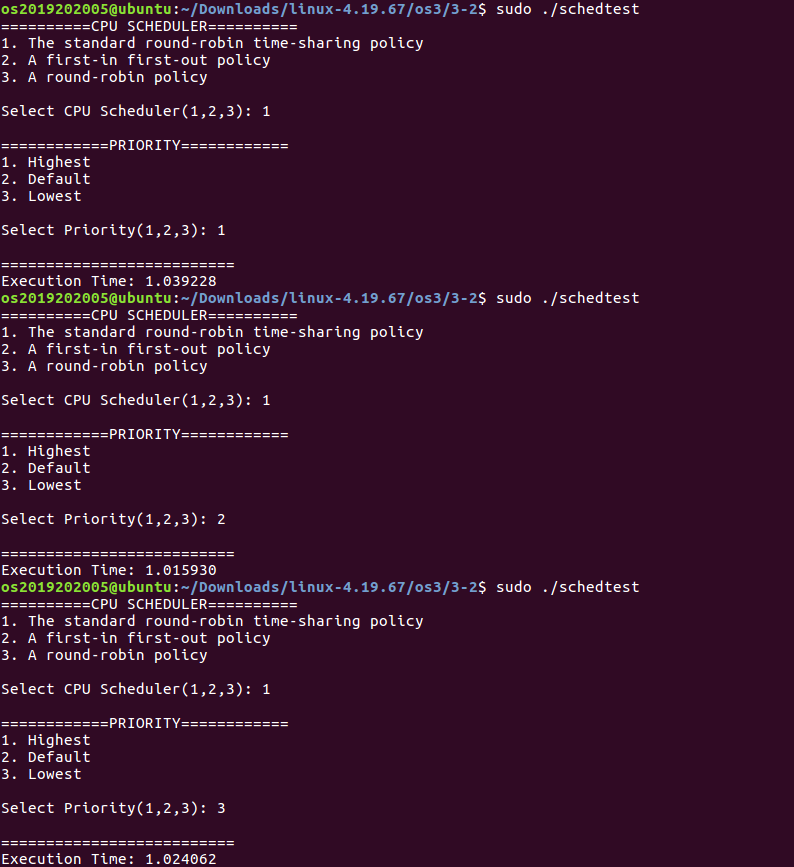
다음은 우선순위가 Default일 때 실행시간을 확인할 수 있습니다.

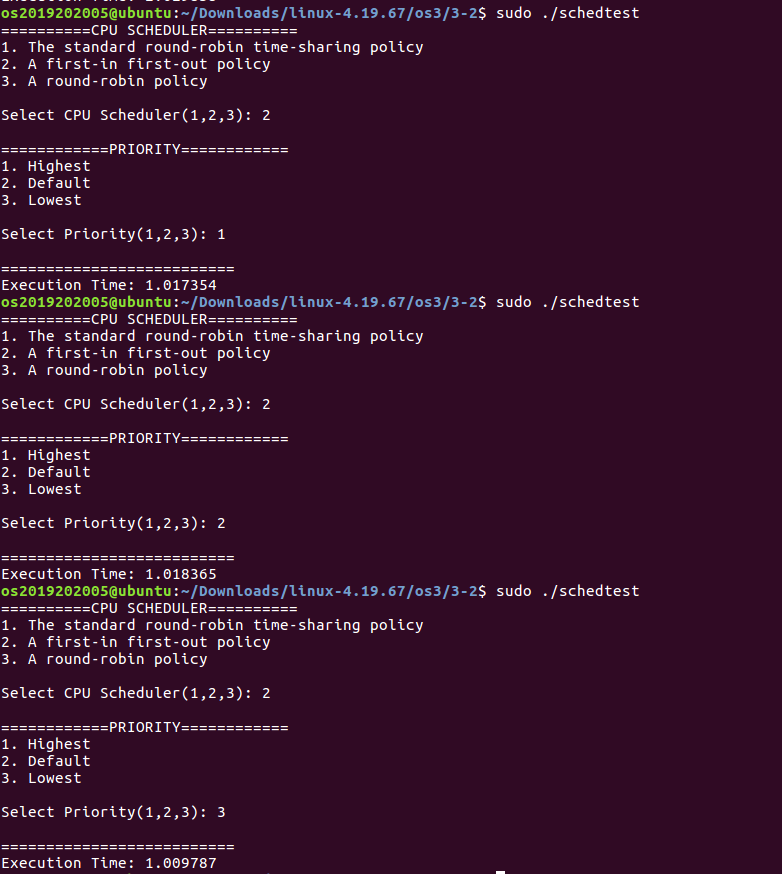
**A round-robin policy & Lowest**

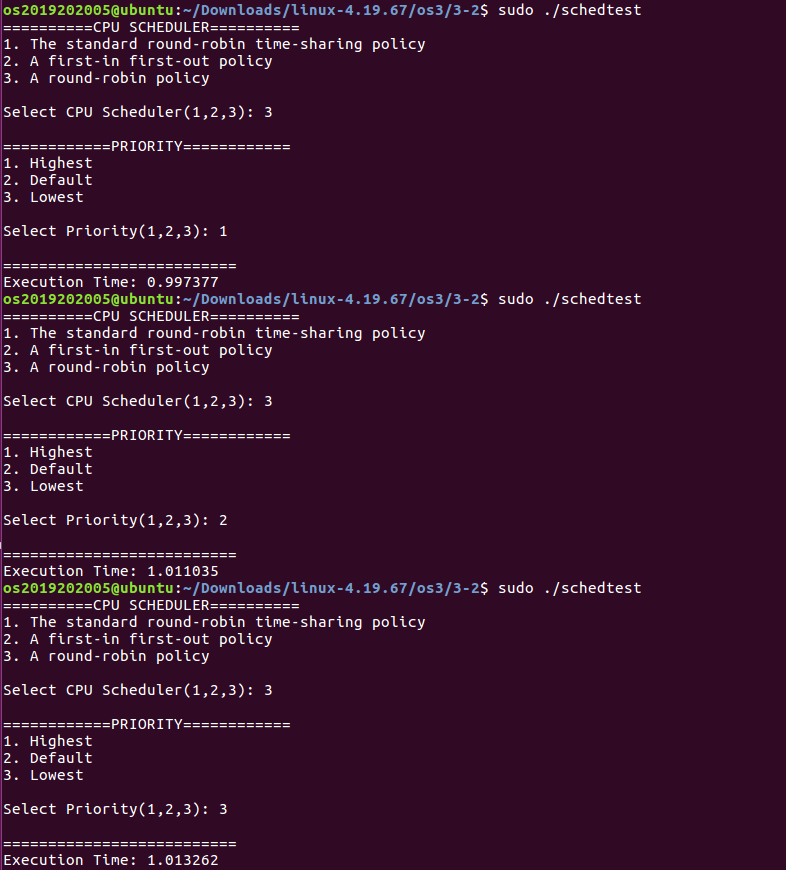
****

다음은 우선순위가 Lowest일 때 실행시간을 확인할 수 있습니다.

위 테스트 이외에도 더 많은 결과를 보고 싶어 테스트를 많이 진행했습니다.

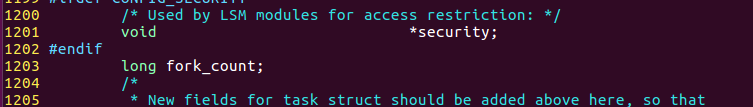




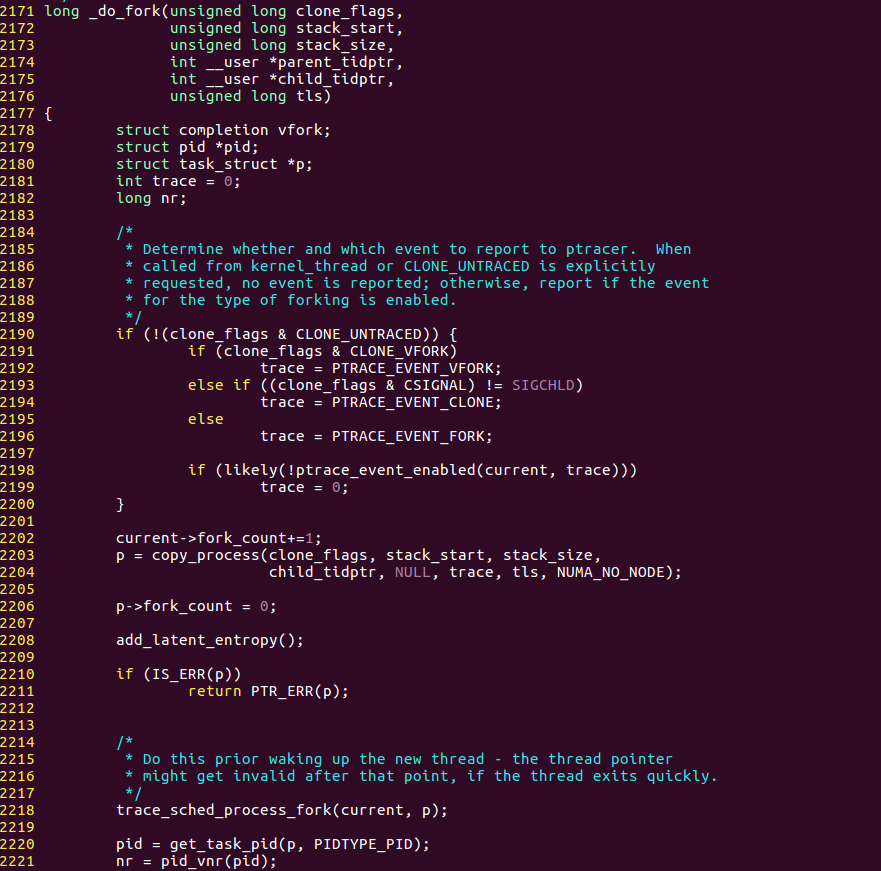


우선순위가 높게 부여되면 먼저 실행이 되는 것은 당연하다고 생각합니다. 하지만 실험을 여러 번 계속 하면서 우선순위가 높다고 해서 실행시간 이 더 짧은가 라는 의문점이 생겼습니다. 그리고 실험에서도 우선순위가 높다고 해서 실행시간이 더 짧고 우선순위가 낮다고 해서 실행시간이 더 길게 결과가 나오지 않았습니다. 또한, 결과를 확인했을 때 실행시간이 조금씩은 차이가 있지만 큰 차이는 나타나지 않았습니다.

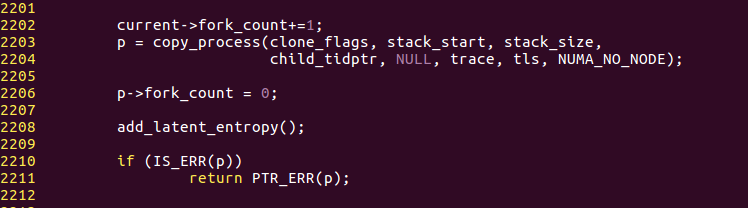
**3-3과제**

****

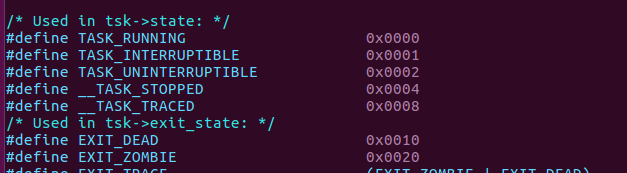
먼저 cscope를 통해 sched.h를 찾아서 이를 수정했습니다. 과제에서 fork() 호출 횟수를 출력하는 부분이 있어 이를 세기 위해 fork\_count라는 변수를 하나 추가했습니다.



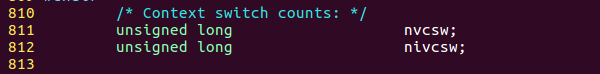
다음으로 cscope를 통해 fork.c를 찾아서 수정했습니다. 기존의 코드를 보았을 때 copy\_process함수를 사용하고 있는 것을 확인할 수 있는데 이는 fork시스템 호출에서 호출되며 새로운 자식 프로세스를 생성하는 데 사용됩니다.

****

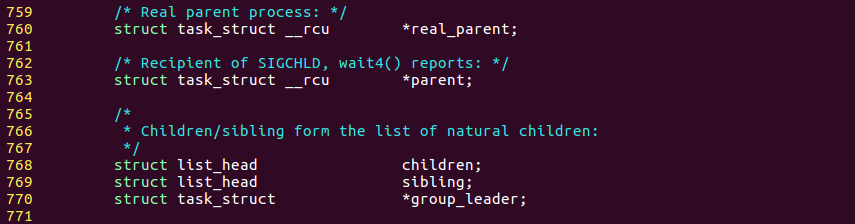
현재 프로세스의 task\_struct를 의미하는 current를 통해 부모 프로세스는 fork()호출 횟수를 1증가하고 자식 프로세스의 fork()호출 횟수는 0으로 초기화 시켜줍니다.

****

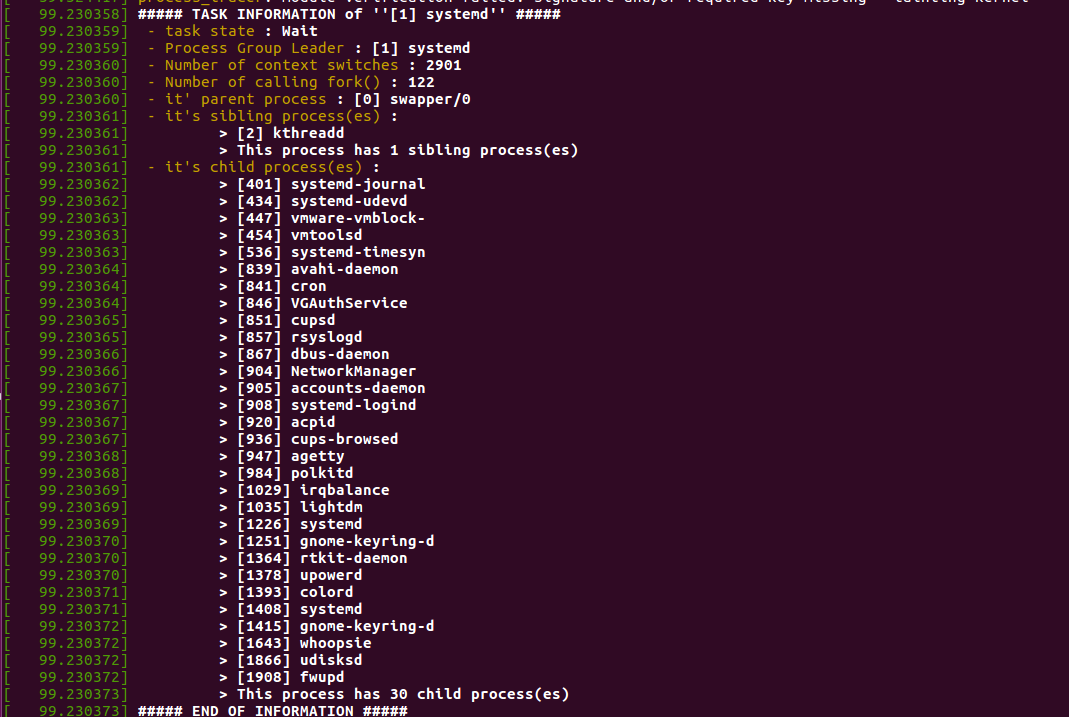
각각의 프로세스 상태에 따라 define된 값을 확인할 수 있습니다.



Context switch count를 출력하기 위해 필요한 정보도 찾을 수 있습니다.



Parent & siblings & child를 출력하기 위한 정보를 위 사진에서 확인할 수 있습니다. 이때 sibling과 child 프로세스는 모두 출력해야 하는데 list\_for\_each와 list\_entry를 사용해서 출력하였습니다.



모듈을 적재하는 과정과 컴파일 하는 과정을 통해 출력문을 확인한 결과입니다.

Pid의 값이 1인 프로세스를 통해 결과를 얻었습니다. 첫째줄에 프로세스의 이름을 나오고 다음 줄에 현재 상태를 확인할 수 있습니다. 다음으로 프로세스의 그룹 리더가 누구인지 출력하며 context switch횟수와 fork()를 호출한 횟수를 출력합니다. 다음으로 부모 프로세스를 출력하고 sibling프로세스와 child프로세스를 모두 출력해줍니다. 총 1개의 sibling프로세스와 29개의 child프로세스가 출력된 모습을 확인할 수 있습니다.

**고찰**

먼저 3-1과제를 진행하면서 저번 학기 시스템프로그래밍 수업시간에 배운 내용들이 생각도 났고 도움이 많이 되었습니다. 그리고 thread를 사용하여 프로그램을 실행할 때가 fork를 통해 실행하는 것보다 더 빠르게 실행시킬 수 있다는 점을 알게 되었습니다. 3-1과제는 다른 과제들 보다 수월하게 할 수 있었던 것 같습니다.

솔직히 지금까지의 운영체제 과제를 진행하면서 운영체제 수업에서 배우는 내용과 과제를 진행하는 부분에 있어서 조금 이질감이 있다고 생각했었습니다. 이론에서 배웠던 내용을 과제에서 거의 보지 못했던 것 같아 그랬었는데 이번 3-2과제를 진행하면서는 이론에서 배웠던 내용들이 과제를 이해하는 데 있어서 굉장히 많은 도움이 되었으며 과제를 하면서 일반 task와 특수 task를 처리하는 cpu스케줄러, nice와 real time priority등 이런 내용들은 추가로 알 수 있어서 굉장히 좋았습니다. 수업시간에 배운 내용들과 이번 시험기간에 공부했던 내용들이라서 이해도 빨랐고 좀 더 찾아보기도 했던 것 같습니다.

3-3과제에서는 저번 2차 과제에서 모듈 관련에서 과제를 진행했던 부분이 많이 도움이 되었습니다. 하지만 과제를 하면서 아직 모듈 프로그래밍이 익숙하지 않다는 점을 많이 느꼈고 그래서 유독 3-3과제를 하면서는 많이 찾아보면서 했던 것 같습니다. 커널을 수정하는 데에 있어서 특히 cscope를 많이 사용하였는데 이를 통해 쉽게 원하는 파일을 찾을 수 있었고 수정할 수 있었습니다. 그리고 출력해야 하는 프로세스 정보들을 모두 task\_struct에서 가져올 수 있어서 수월하게 출력할 수 있었습니다. 그리고 sibling과 child프로세스를 모두 출력해야 하는데 어떻게 출력할 지 고민하던 중 list\_for\_each와 list\_entry함수를 알게 되어 이를 사용하여 출력할 수 있었습니다.

그리고 fork호출 횟수를 출력할 때 커널 파일을 건드렸기 때문에 커널 컴파일을 진행했어야 하는데 처음에 이 과정을 진행하지 않고 다음 과정들을 진행해서 다른 것들은 다 잘 출력되는데 fork호출 횟수만 계속 0이 출력되어서 이 부분이 의심이 갔습니다. 다행히 이유를 찾아 커널 컴파일을 진행하였고 그 후 fork호출 횟수까지 잘 출력할 수 있었습니다.

**Reference**

강의자료 2023-2\_OSLab\_09\_CPU\_Scheduling

강의자료 2023-2\_OSLab\_Assignment\_3

Real-time 개념 및 리눅스에서의 RT Scheduling 이해하기 <https://blog.naver.com/alice_k106/221149061940>

[모듈프로그래밍] - task\_struct 이용하여 프로세스 정보 확인하기

<https://riucc.tistory.com/208>

모듈프로그램을 이용하여 리눅스 커널 분석하기(1)

https://url.kr/tq9biv

모듈프로그램을 이용하여 리눅스 커널 분석하기(2)

https://url.kr/vzldbk

모듈프로그램을 이용하여 리눅스 커널 분석하기(3)

<https://url.kr/hnbgt4>

C/C++ list\_for\_each\_entry() 함수

<https://tear94fall.github.io/lecture/2020/01/30/list-for-each.html>