운영체제 [COSE341] 2차과제

학과 컴퓨터학과

학번 2020320026

이름 권도혁

제출일 2023. 05. 30

Freeday 사용 일수 0일

목차

1. 과제 개요

이번 과제에서는 Round Robin 스케줄러의 time slice가 성능에 미치는 영향에 대해 알아본다. 주어진 행렬 연산 코드를 기반으로 time slice가 각각 1ms, 10ms, 100ms로 변할 때 마다 수행하는 연산 횟수로 성능 변화를 관찰하는 방법으로 진행하며, 모든 작업은 Ubuntu에서 Linux-4.20.11 커널 버전으로 진행하였다.

2. Round Robin 스케줄링에서 Time Slice의 영향 설명

Round Robin 스케줄링은 ready queue에 있는 프로세스들을 FCFS 방식으로 특정 time slice동안 실행한 후 다음 프로세스를 실행하는 선점형 스케줄링 알고리즘이다. 따라서 RR 스케줄링에서 time slice는 하나의 프로세스가 실행되는 시간을 결정하고, 이에 따른 전체 실행시간에서의 context switch 횟수와 context switch에 의한 overhead를 결정한다. 그러므로 RR 스케줄링에서 적합한 time slice를 결정하는 것은 전체 성능을 좌우하는 중요한 요소이다. time slice가 매우 작다면 전체 실행 시간에서 context switch에 의한 overhead가 굉장히 커져 context switch를 수행하는 시간이 프로세스 실행 시간보다 더 커지는 상황이 발생하고, time slice가 매우 크다면 비선점형 스케줄링인 FCFS 방식과 유사한 동작으로 하게 되며 이는 convoy effect를 발생시킬 것이다.

3. 소스코드 및 작업 내용 설명

cpu.c 파일은 주어진 행렬 연산 코드를 기반으로 작성된 calc 함수와 fork와 waitpid를 수행하는 main 함수, RR 스케줄링으로 설정하기 위한 sched_setattr 함수, RR 스케줄링으로 변경하기 위한 sched_setattr에 필요한 sched_attr 구초제, 시그널을 처리하기 위한 전역변수들과 시그널 핸들러 함수 handler로 이루어져 있다.

A. main()

우선 main 함수는 프로그램을 실행할 때 입력하는 프로세스 개수와 프로그램 실행 시간을 정수 자료형으로 변환하기 위해 atoi 함수를 사용하고 이를 전역 변수 num과 dur에 할당한다. 다음으로 sched_setattr 함수의 반환값을 받기 위한 정수형 변수 sched, 자식 프로세스의 pid를 저장하기 위한 정수형 배열 pidarr을 선언하고, RR 스케줄링으로 변경하기 위해 sched_attr 구조체 변수 attr을 선언하고 memset으로 초기화한 다음, 프로

세스 우선순위와 스케줄링 정책을 RR로 설정한다.

다음으로 프로세스 개수만큼 반복하는 for문으로 진입하고 fork를 수행하기 전에 RR 스케줄링으로 변경하는 과정을 거친다. 부모 프로세스에서 fork를 수행하면 자식의 pid가 반환되기 때문에 부모 프로세스에서 이를 pidarr에 저장하고 자식 프로세스는 calc 함수를 수행한다. 부모 프로세스에서 프로세스 개수만큼 fork를 수행하면 for문을 탈출하고 다시 waitpid를 수행하기 위한 for문에 진입한다. 이 과정에서 부모 프로세스가 생성한 각 자식 프로세스들이 좀비 프로세스가 되는 것을 방지한다.

B. calc()

부모 프로세스가 자식이 terminate 되는 것을 wait하는 동안 자식 프로세스는 calc를 수행한다. calc에서는 주어진 행렬 연산 코드를 수행하기 전에 시간을 측정하기 위한 timespec 구조체 변수 begin과 end를 선언한다. 이는 calc 수행 중 100ms 마다 수행한 연산 횟수를 출력하기 위한 100ms 타이머 변수이고, 전체 수행 시간 측정을 위한 totalbegin 변수는 시그널 처리를 위해 전역변수로 선언하였다. 행렬 연산을 수행하기 직전 100ms 타이머와 전체 시간 타이머를 시작하고 행렬 연산을 시작한다.

행렬 연산 도중 count 값이 증가할 때 마다 100ms가 지났는지 확인한다. 100ms를 측정하기 위해 long long 타입의 t를 선언하여 end와 begin의 초(sec)의 차에 1000을 곱하고 나노초(ns)의 차에 1000000을 나누어 더한 값으로 밀리초(ms) 값을 구했고 이 값이 100이 넘는다면 지금까지의 count 값을 출력하도록 했다. 같은 방법으로 totaltime을 end와 totalbegin의 차가 dur을 넘는다면 총 연산횟수를 출력하고 break 하도록 했다.

C. handler() & signal()

행렬 연산 도중 시그널을 처리하는 방법은 행렬 연산을 수행하기 위해 while문으로 진입할 때와 count 값이 증가하여 다시 처음부터 행렬 연산을 수행할 때 signal() 함수로 SIGINT(ctrl + c)가 발생했을 때 handler 함수로 넘어갈 수 있도록 구현하였다. signal() 함수는 2개의 파라미터를 가지는데, 첫번째 파라미터는 시그널 종류, 두번째 파라미터는 시그널을 처리하는 handler 함수의 주소이다.

구현한 handler 함수는 SIGINT가 발생하면 수행되고 이때 지금까지 각 프로세스가 수행한 연산 횟수를 출력하고 exit 하도록 했다.

D. sched_info_depart()

CPU burst time을 커널 로그에 출력하기 위한 kernel/sched에 위치한 stats.h 파일의 sched_info_depart 함수는 우선순위가 10인 프로세스에 대해 커널 로그에 CPU burst time을 pid, priority와 함께 출력하도록 수정했다.

4. 성능 관찰 및 분석

A. 표와 차트의 값

우선 표와 차트에서 각 time slice마다 관찰한 연산 횟수는 각 time slice마다 5회 실행하여 산술 평균값을 도출한 후, 소수점 이하를 버린 값이다. 또한 각 프로세스의 CPU burst time과 초당 연산횟수, 단위 시간당(Baseline=1ms, Baseline=10ms) 연산 횟수 대비 각 time slice 당 연산 횟수 증가율 및 감소율은 소수점 3번째자리에서 반올림한 값이다.

B. time slice와 연산 횟수의 관계

아래의 표 1과 차트 1은 단위 시간 동안 행렬 연산 횟수를 나타낸 것으로, 30초 동안 연산을 수행한 후 time slice가 1ms의 경우 합계 5433회, 10ms의 경우 합계 5750회, 100ms의 경우 합계 5938회로 측정되었다. 이는 time slice가 1ms일 때를 기준으로 연산 횟수가 10ms일 때 5.83%, 100ms일 때 9.3% 증가했다. 10ms일 때를 기준으로 100ms일 때 연산 횟수가 3.46% 증가했다. 이를 통해 time slice와 연산 횟수는 비례 관계에 있음을 알 수 있다.

RR Time slice	1ms	10ms	100ms
	# of calc.	# of calc.	# of calc.
Process #0	2728	2869	2972
Process #1	2705	2881	2966
Total calc.	**5433**	**5750**	**5938**

표 1 [time slice 별 연산횟수]

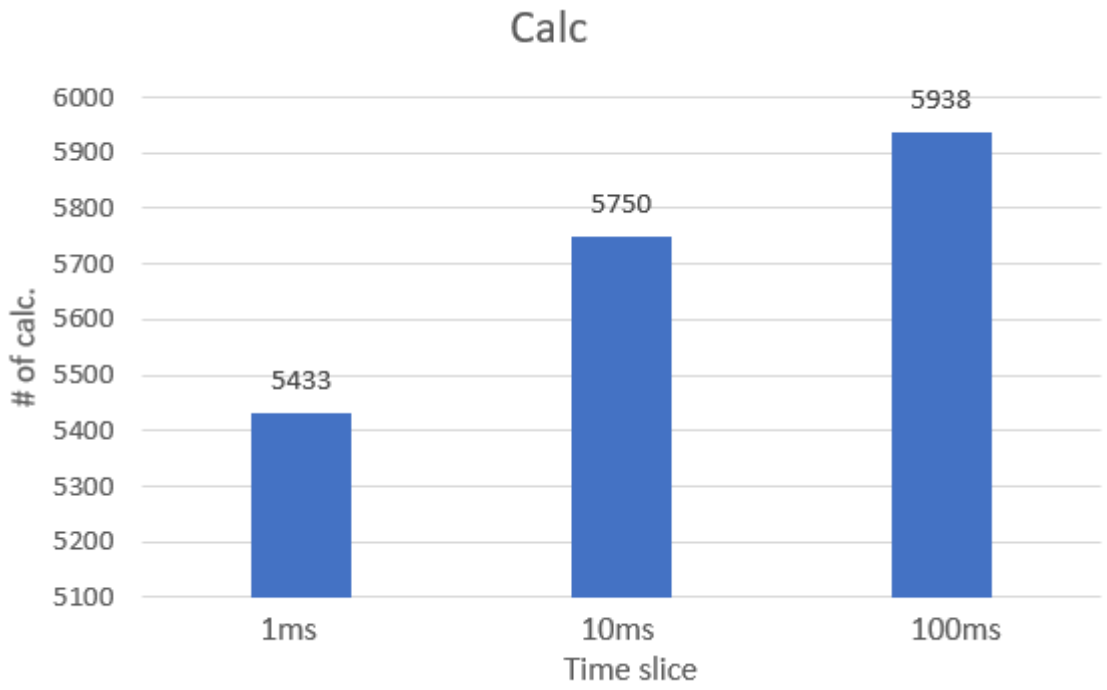

차트 1 [time slice 별 연산횟수]

위의 결과로 알 수 있듯 time slice가 증가하면 성능이 대체적으로 개선된다는 결론을 추론할 수 있다. 하지만, 이러한 분석은 30초 동안의 연산 횟수 데이터만을 가지고 분석한 결과이기 때문에 각각의 time slice 동안 연산 횟수의 정확한 증감을 관찰하는 것에 한계가 존재한다. 이는 30초 동안 실행되는 행렬 연산 프로세스를 제외한 다른 프로세스의 존재를 고려하지 않은 분석이기 때문이다. 따라서 단위 시간 동안의 CPU burst time을 도출하여 CPU burst time 동안의 연산 횟수를 비교하여 분석하는 방법을 통해 이러한 문제점을 해결할 수 있다.

C. CPU burst time을 고려한 time slice와 연산 횟수의 관계

아래의 표 2는 각 프로세스 별 CPU burst time과 프로세스가 수행한 행렬 연산 값이다. 표 3은 CPU burst time 동안 초당 연산횟수와 이를 고려하여 도출한 연산횟수의 증감을 구한 값이다. CPU burst time을 고려하지 않은 분석과는 달리, CPU burst time을 고려한 분석에서 단위 시간 당 연산 횟수 증감은 줄어든 것을 알 수 있다. 1ms를 기준으로, 10ms일 때 5.79% 연산 횟수가 증가하였고, 100ms일 때 8.93% 증가했고 10ms를 기준으로, 100ms일 때 2.96% 증가했음을 알 수 있다.

RR Time slice	1ms		10ms		100ms	
	# of calc.	Time(s)	# of calc.	Time(s)	# of calc.	Time(s)
Process #0	2728	14.95	2869	15.04	2972	15.12
Process #1	2705	15.04	2881	14.96	2966	14.97
Total calc. and Time	**5433**	**29.99**	**5750**	**30**	**5938**	**30.09**

표 2 [CPU burst time과 time slice 별 연산횟수]

RR Time slice	1ms	10ms	100ms
Calculations per second	181.16	191.66	197.34
Baseline=1ms	100	105.79	108.93
Baseline=10ms	**94.52**	**100**	**102.96**

표 3 [CPU burst time동안 초당 연산횟수와 time slice 별 연산횟수 증감]

실제 CPU burst time을 고려한 경우와 그렇지 않은 경우 모두 time slice가 증가함에 따라 단위 시간 당 연산 횟수가 증가하는 추세를 보여준다. time slice가 증가함에 따라 context switch의 횟수가 줄어들기 때문에 프로세스가 CPU를 점유하는 시간이 증가하여 연산횟수가 증가했음을 알 수 있다.

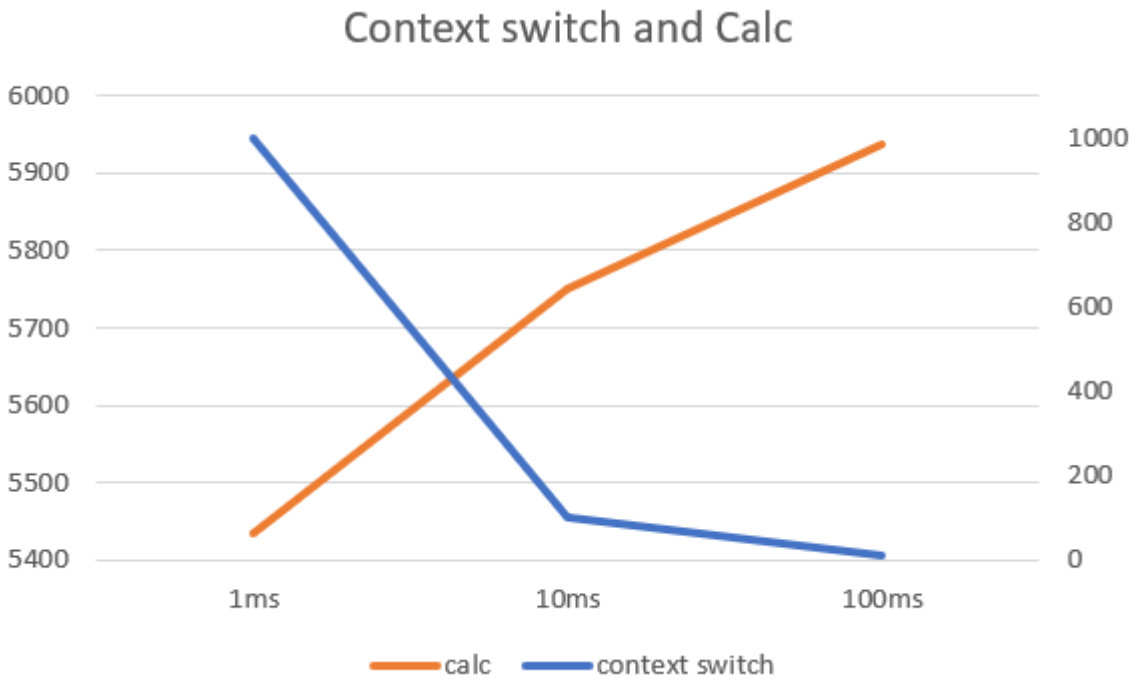

차트 2 [Context switch 횟수와 연산횟수]

위의 차트 2에서 time slice가 증가함에 따라 context switch 횟수는 감소하는 반비례 관계에, 행렬 연산 횟수는 증가하는 비례 관계에 있음을 알 수 있다. 이러한 결과에서 Round Robin 스케줄링에서 time

slice가 증가함에 따라 프로세스를 처리하는 성능이 증가함을 추론할 수 있다.

D. 결론

B와 C에서의 데이터를 기반으로, time slice가 증가하면 전체 성능이 좋아진다는 결론을 도출할 수 있다. 이는 time slice가 증가하면 프로세스가 CPU를 점유하는 시간이 증가하여 프로세스에 필요한 연산을 많이 수행할 수 있게 되어 나타나는 성능 개선이다.

하지만, 매우 큰 Time slice가 적용되는 경우 FCFS와 유사하게 동작할 수 있기 때문에 convoy effect를 야기할 수 있으며, 처리하려는 프로세스가 responsiveness-sensitive한 경우에는 큰 Time slice가 오히려 성능 저하를 불러올 수 있기 때문에 OS는 RR 스케줄링을 사용할 경우 최적의 성능을 낼 수 있는 적절한 Time slice를 채택하는 것이 중요하다.

5.과제 수행 과정 중 문제점과 해결 과정

A. fork()와 exit()

cpu.c 파일을 작성하기 전에 fork()에 대한 이해를 먼저 할 필요가 있다고 생각하여 forktest라는 파일을 만들어 fork()를 실험하는 과정을 거쳤다. 어떻게 하면 for문 안에서 fork()를 하면, child가 for문을 거치면서 새로운 child를 계속해서 생성하여 어떻게 하면 원하는 만큼만 fork를 수행할 수 있을지 고민하여 fork를 하고 원하는 작업을 하고 난 뒤 바로 exit(0)를 하면 새로운 child를 생성하지 않는다는 것을 보았다.

하지만 이 경우에 child가 좀비 프로세스가 되는 문제가 있었다. 이는 ctrl + c를 입력했을 때 프로세스가 종료되지 않는 문제를 발생했기 때문에 알게 되었다. 프로세스가 좀비 프로세스가 되었을 경우 ctrl + c를 입력해도 종료되지 않을 수 있다는 정보를 얻어 부모에서 wait 하는 과정이 필요함을 알게 되어 부모에서 입력한 프로세스 개수만큼 fork를 수행한 후 각 자식에 대해 wait을 하는 과정을 거치게 함으로 이를 해결할 수 있었다.

B. dmesg

커널 로그에 CPU burst time을 출력하고자 커널 소스를 변경하고 cpu.c를 실행한 후 dmesg로 로그를 봤는데 아무것도 출력되지 않는 문제가 있었다. 커널 소스를 변경할 때 코드에 오타가 있었는지 확인해봤지만 문제가 없었다. 하지만 uname -r을 통해 커널 버전을 확인해 보니 linux-4.20.11 버전이 아님을 확인했다. 정확하지는 않지만 커널을 수정하고 컴파일하고 나서 다시 시작할 때

커널 버전이 바뀌었지 않았나 추측하여 다시 시작할 때 linux-4.20.11 커널 버전으로 실행하여 dmesg를 해보니 CPU burst time이 잘 나온 것을 확인할 수 있었다.

C. 100ms

행렬 연산 중 100ms 마다 지금까지의 count 값을 출력해야하는 과정이 있었다. 이를 위해 시간을 측정하고 count가 증가할 때마다 시간을 확인해 100ms 인지 아닌지 확인했다. if문의 조건에 시간이 정확히 100ms일 경우를 조건으로 줄 경우 연산 수행 중 100ms 마다 출력되는 경우가 적었다. 이를 통해 연산 수행 중 100ms를 초과하는 경우가 많이 발생한다는 것을 알게 되었다.

이러한 문제를 해결하는 방법에 나노초(ns) 단위로 sleep하여 100ms를 비교적 정확히 측정하는 방법이 있었다. 하지만 이 방법은 sleep하는 과정 때문에 count 값에 꽤 큰 영향을 줄 것이라고 생각했다. 뿐만 아니라 이 방법도 예외가 존재하여 100ms가 초과하는 경우가 생기는 것은 분명하기 때문에 원래의 방법보다 그저 확률을 높이는 방법이라고 판단해서 채택하지 않았다.

결국 작게는 1~3ms, 크게는 10~20ms의 오차가 있더라도 시간이 100ms 이상일 때 출력하는 것으로 결정했다. 이것이 문제를 해결했다고 볼 수 없지만, 처음보다 내가 원하는 결과에 훨씬 가까운 결과를 낼 수 있었다.

D. 전체 시간 측정

처음 시간을 측정할 때 timespec 구조체 변수 2개만을 사용하여 전체 시간과 중간 100ms 체크포인트를 측정했다. 하지만 이 경우 100ms 체크포인트를 찍고 다시 타이머를 초기화할 때 전체 시간까지 초기화되어 프로그램이 무한하게 실행되는 문제가 있었다.

이를 해결하기 위해 전체 시간을 측정하는 변수를 하나 더 만들어서 100ms 체크포인트 마다 전체 시간 변수에 더해서 전체 시간을 측정하는 방법과 타이머를 두 개 만들어서 체크포인트용 타이머와 전체 시간용 타이머를 따로 운용하는 방법을 생각했다. 나는 두 가지 방법 중 타이머를 두 개로 운용하는 방법을 채택하여 전체 시간 문제를 해결할 수 있었다.