운영체제 2차 과제

학과: 노어노문학과

학번: 2018131321

이름: 김현우

제출일: 2022.12.11

Freeday 사용일수: 3일

목차

개발환경

Window11, VirtualBox, Linux 18.04.02 LTS, Kernel Version 4.20.11

Ⅰ. 과제 개요와 Round Robin 스케줄링에서 time-slice의 영향 설명

1. 과제 개요

과제2번의 목표는 유저 어플리케이션을 구현하여 time-slice에 따른 문맥전환(Context switching) 오버헤드를 측정하는 것이다. 다음과 같은 목적을 달성하기 위해 과제를 3단계에 거쳐서 진행했다. 우선 성능 평가에 사용할 유저 어플리케이션(cpu.c)을 구현하였다. 이 프로그램을 프로세스 2개, 수행 시간 30초로 설정하여 수행하는 것을 목표로 하였다. 두 번째 단계에서는 유저 어플리케이션의 스케줄링 클래스를 RT-RR로 변경하였다. 자식 프로세스를 생성하는 fork()를 구현하기 전에 미리 스케줄링 클래스를 변형하는 과정이 필요하였다. 마지막 단계에서는 커널 레벨에서 CPU burst를 측정하였다. 이때 성능 영향을 정확하게 관찰하기 위해 코어 수를 1개로 설정하는 과정이 선행되었다. 커널 레벨에서 CPU burst를 측정하는 목적은 더 정확한 CPU 수행 시간을 측정하기 위함이다.

2. Round Robin에서 time-slice의 영향

CPU 스케줄링은 메모리 내에서 실행이 준비된 프로세스 중 하나를 선택하여 CPU에 할당하는 작업을 말한다. 스케줄링을 구현할 때 시스템의 용도에 따라 요구사항이 달라진다. 이번 과제에서는 개인용 컴퓨터에서의 스케줄링을 구현하는 것이 목적이므로 빠른 응답시간으로 스케줄링을 디자인하는 것이 중점적인 과제이다. 따라서 빠른 응답시간을 요구하는 Interactive 프로세스에 적합한 Round Robin 스케줄링 기법을 사용하였다.

Round Robin 기법은 선점형 스케줄링 방식으로 CPU를 시간 단위(time quantum)로 할당한다. 시간 단위를 모두 수행한 프로세스는 Ready 큐의 끝으로 들어가 다시 할당을 기다린다. 각 프로세스의 다음 time quantum이 돌아오기까지의 시간은 n개의 프로세스가 큐에 존재한다고 가정할 때 최대 (n-1) * q(Time quantum)이다. 리눅스에서는 Round Robin 기법을 사용하기 위해서는 'SCHED_RR'이라는 스케줄링 정책을 할당해줘야 한다. 이 정책은 슈퍼 유저(root)만이 생성할 수 있다.

Round Robin 스케줄링 방식은 작업을 전환할 때마다 Context switching이 발생하고, 이러한 과정에서 발생하는 오버헤드가 달라지기 때문에 time-slice를 어떻게 할당하는지에 따라서 성능에

차이가 발생한다. 만약 time-slice가 작다면 작업이 빈번하게 전환된다. 이때 많은 Context switching이 발생하고 그에 따라 오버헤드가 상당량 증가한다. 더불어 만약 문맥 전환에 필요한 시간보다 time-slice가 낮다면 효율이 매우 떨어지게 된다. 반면, time-slice가 크다면 응답시간이 느려지기 때문에 Interactive 프로세스를 Round Robin 기법으로 사용하는 이유가 없어진다. 만약 time-slice가 극단적으로 크다고 가정했을 때, 프로세스의 크기만큼 CPU가 처리를 진행하다가 해당 프로세스가 끝나야 다른 프로세스가 처리될 것이다. 따라서 이 경우는 FCFS와 같은 성능을 보이게 된다. 따라서 적절한 크기의 time-slice를 설정하는 것이 중요한 과제로 부각된다.

Time quantum의 값을 정하는 규칙인 'rule of thumb'에 따르면 time-slice는 CPU burst의 80% 정도를 포함할 수 있어야 한다고 명시되어 있다. Context switching 자체가 다른 작업을 수행하는 과정에서 오버헤드를 발생시키기 때문에 문맥 전환을 줄이는 게 좋다. 따라서 time-slice를 너무 작지도 않고, 너무 크지도 않게 조절하려는 과정에서 도출된 최선의 값이 CPU burst의 80% 정도이다.

Ⅱ. time-slice의 변화에 따른 성능을 관찰한 표 및 결과 분석

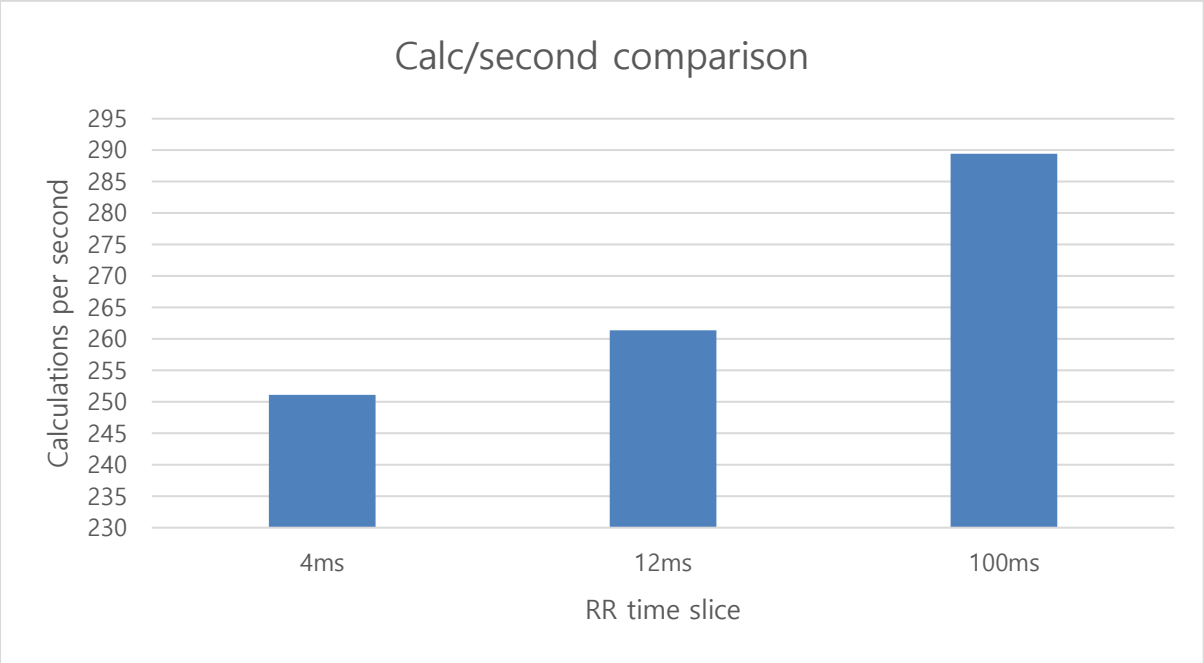
1. Time-slice변화에 따른 성능 표 및 그래프

RR Time slice	4ms		12ms		100ms	
	# of calc.	Time(s)	# of calc.	Time(s)	# of calc.	Time(s)
Process #0	3773	15.224861775	3875	15.026435981	4296	15.000708427
Process #1	3876	15.231390332	3978	15.020034949	4386	14.997249124
Total calc and Time	7649	30.456252107	7853	30.046470930	8682	29.997957551

<표1>

RR Time slice	4ms	12ms	100ms
Calculations per sec (total calc/Time)	251.147120	261.361809	289.419704
Baseline=4ms	100.00	104.07	115.24
Baseline=12ms	96.09	100.00	110.74

<표2>



<그래프>

2. 표 및 그래프 결과 분석

프로세스를 time-slice 1ms, 10ms, 100ms로 설정하여 진행하려고 하였다. 그러나 sched_rr_timeslice_ms를 1ms, 10ms, 100ms로 설정하고 fork()로 각 child 프로세스를 생성할 때, sched_rr_get_interval 함수를 이용해 time-slice를 확인하면 다른 값이 나온다. Time-slice는 4ms, 12ms, 100ms로 측정된다. 따라서 실제 burst time인 4ms, 12ms, 100ms로 실험 결과 값을 분석하였다.

Time-slice가 4ms일 때, 총 행렬 연산 횟수는 7649번으로 측정되었다. 또한, 두 프로세스의 행렬 연산 CPU burst는 30.456252107(sec)으로 측정되었다. 'dmesg'명령어를 통해 측정한 PID별 CPU burst는 ns 단위로 측정되므로 초 단위로 변경하였다. 같은 방법으로 12ms, 100ms일때의 값도 측정하였다. 12ms의 행렬 연산 횟수는 7853회이고 CPU burst는 30.046470930(sec)이다. 100ms의 행렬 연산 횟수는 8682회이고 CPU burst는 29.997957551(sec)이다. 이를 기록하여 정리한 값들은 <표1> 에 나타나 있다.

표를 통해서 Time-slice의 값이 클수록 연산 횟수가 증가하는 결과를 볼 수 있다. 이는 time-slice가 클수록 문맥 전환에서 발생하는 오버헤드 값이 적기 때문이라고 유추할 수 있을 것이다. 100ms는 4ms의 time-slice를 가질 때보다 문맥 전환이 자주 발생하지 않으므로 행렬 연산에 더 많은 시간을 투자할 수 있을 것이다. 따라서 7649의 연산 횟수보다 많은 8682번의 결과를 발생시킨 것으로 생각한다.

<표2> 에는 <표1> 의 값을 활용해 3가지 time-slice에 따른 값들의 변화를 비교하였다. <표1>의 행렬 연산 횟수(# of calc)를 초 단위 수행 시간(Time)으로 나눈 값을 Calculations per sec 항목에 기록하였다. 다음 값들을 4ms, 12ms time-slice를 기준으로 비교해보았다. 4ms를 기준으로 할 때, 12ms는 약 4.07% 성능이 상승하였다. 또한, 100ms일 때 약 15.24% 성능이 상승하였다. 12ms를 기준으로도 같은 방식으로 값을 비교해보았다.

결과를 정리하면 다음과 같다.

- **4ms의 경우, 10ms와 비교 시 약 4.07%의 성능 하락이 발생**
- **12ms의 경우, 100ms와 비교 시 약 10.74%의 성능 하락이 발생**
- **즉, context switching이 10배 정도 많아지는 경우, 약 3~11%의 성능 하락이 발생**

성능에 있어서 결과의 차이가 발생하는 이유는 앞서 말했듯 time-slice 값이 작을수록 문맥 전환이 많이 일어나게 되고, 그에 따라 오버헤드가 많이 발생하기 때문이다. 4ms는 12ms보다 더 많은 context switching이 발생하기 때문에 12ms보다 대략 4.07%의 성능 하락이 발생했다. 12ms를 100ms와 비교했을 때 비슷한 성능 하락이 발생한다. 즉, time-slice의 값이 대략 10배 작아질 때마다 3~11% 정도의 성능하락이 발생한 것을 표를 통해 도출할 수 있다. 이러한 값들을 바탕으로 그래프를 만들었을 때, 각 time-slice마다의 calculations per second 값들의 차이를 확연히 알 수 있다.

Ⅲ. 소스코드 및 작업 내용에 대한 설명

1. User application 작성

앞서 설명했듯 과제를 총 3단계를 거쳐 진행했다. 우선 첫 번째로 성능 평가에 사용할 유저 어플리케이션 cpu.c를 작성했다. 유저 어플리케이션은 행렬 연산을 수행하는 부분과 입력한 값만큼 자식 프로세스를 생성하는 부분으로 구분되어 있다.

```
int forknum = atoi(argv[1]);
int time = atoi(argv[2]);
pid_t pid[forknum];
```

forknum 변수는 생성할 자식 프로세스의 값을 입력하여 받는 부분이다. Time 변수는 수행할 시간(sec)이 들어갈 부분이다. Pid 배열은 생성한 자식 프로세스들의 값을 받는 부분이다. 배열로 선언한 이유는 for 함수를 통해 자식 프로세스를 생성할 계획이기 때문이다.

```
for(i=0 ; i<forknum ; i++)
        pid[i] = fork();
        if(pid[i] == 0)
                calc_result = calc(i, time, ts.tv_nsec);
                exit(0);
while(wait(&status) != -1);
```

이후 for을 통해서 입력받은 자식 프로세스만큼 생성해 주었다. 이때 앞서 말한 pid[i] 배열에 fork() 시스템콜을 통해 프로세스를 생성해 주었다. 이때 pid[i]가 0일 때는 자식 프로세스에 들어간 것이므로 자식 프로세스일 때 진입하는 조건문을 만들었다. 자식 프로세스로 진입한 후 행렬 연산을 수행하는 calc함수를 호출하였다. 이후 wait()를 통해 자식 프로세스가 종료될 때까지 기다리는 작업을 추가해주었다.

```
        clock_gettime(CLOCK_MONOTONIC, &begin);
        //행렬 연산 부분
        clock_gettime(CLOCK_MONOTONIC, &end);
```

이 부분에서는 행렬 연산이 시작하고 끝난 시간을 계속해서 측정하였다. 시스템 부팅 시간을 기

준으로 시간을 측정하는 CLOCK_MONOTONIC으로 clock_gettime() 시스템콜을 호출하였다. 측정을 시작한 시간은 begin 변수로, 측정을 끝낸 시간은 end 변수로 들어가게 구현하였다. 이후 begin과 end값으로 나노초 시간을 구하였다. 구한 값은 time_nano 변수에 들어간다.

```
temp += time_nano;
time_milli = temp/1000000;
time_second = temp/1000000000;
```

time_nano 변수는 temp변수에 누적으로 더해진다. 이때 temp변수에 누적된 시간의 값은 나노초이므로 밀리초와 초로 바꿔줄 필요가 있다. 따라서 temp변수를 적절한 값으로 나눠줘 time_milli에는 누적된 밀리초 단위의 시간이, time_second에는 누적된 초 단위의 시간이 들어가게 해주었다.

```
if(time_second >= time)
        break;
```

마지막으로 어플리케이션을 실행할 때 프로세스가 실행할 시간을 입력하므로 그 부분에 대한 구현 역시 필요했다. 따라서 time(입력한 시간)과 누적된 시간 값이 같거나 초과하면 행렬 연산 함수를 멈추는 부분을 구현했다. time과 time_second는 초 단위의 시간이므로 비교할 수 있다. 이때 누적된 시간 값보다 초과하는 부분을 추가한 이유는 행렬 연산 과정이 계속되면서 마지막 연산이 정확히 time 값과 같지 않고 살짝 초과하는 경우를 상정한 것이다.

2. 유저 어플리케이션의 스케줄링 클래스를 RT-RR로 변경

```
struct sched_attr attr;
attr.sched_priority = 10;
attr.sched_policy = SCHED_RR;
```

두 번째 작업은 유저 어플리케이션의 스케줄링 클래스를 RT-RR로 변경하는 것이었다. 이러한 과정을 수행하기 위해서 먼저 sched_attr 구조체를 만들었다. 구조체에는 여러 변수가 있지만 이 작업에서는 스케줄링 정책을 변경할 수 있는 sched_policy, 추후에 사용할 priority의 값들을 변경해주었다. 메인 함수에서 sched_attr 구조체를 attr로 선언하고 초기화해주었다. 또한, policy 값 역시 SCHED_RR로 변형해주어 Round-Robin 스케줄링 기법으로 구동되도록 했다. Priority 값은 10으로 설정했다.

```
setattr_result = sched_setattr(getpid(), &attr, 0);
interval = sched_rr_get_interval(0, &ts);
```

이후 sched_setattr() 함수를 fork() 이전에 선언하여 자식 프로세스가 모두 RT-RR로 수행하도록 해주었다. Sched_setattr() 함수는 호출되어 sched_setattr() 시스템 콜을 사용할 수 있게 해준다. 이때 getpid()를 사용하는 부분이 자식 프로세스로 들어간 if문이므로 자식 프로세스의 값들에 영향을 준다. 또한 interval 변수에 time-slice 값을 받는 함수를 선언해준다. Interval 값은 다시 calc 함수에서 사용된다.

```
tv_millis = tv_nsec / 1000000;
if(time_milli % tv_millis==0)
```

calc 함수에서 interval 값을 tv_nsec로 받아준다. 이 값은 나노초 단위이기 때문에 값을 밀리초 단위로 변형시켜주었다. 이 값은 프로세스가 행렬 연산을 수행할 때의 중간값을 나타내는 데 사

용된다. 수행한 시간의 누적값인 time_milli를 time-slice로 나눠주면서 tv_millis 값이 지날 때마다 중간값을 출력해주었다.

3. 커널 레벨에서 CPU burst 출력

이후 커널 레벨에서 CPU burst를 출력하였다. 정확한 CPU burst 시간을 출력하기 위해서 커널 레벨에서 측정하였다. 이때, time-slice는 1ms, 10ms, 100ms 순서대로 변형하면서 값을 출력하였다. 또한 여러 개의 코어를 사용하면 문맥 전환 횟수 파악이 안 되기 때문에 정확한 값을 판단하기가 어려워 1개의 cpu만을 사용하도록 설정을 변경해주었다.

if(t->rt_priority == 10)

CPU burst 출력은 stats.h의 sched_info_depart 함수에 출력하는 부분을 추가해서 구현하였다. 앞서 말했듯 sched_priority를 10으로 유저 어플리케이션에서 변경하였고 그에 따라 priority가 10인 자식 프로세스들이 출력되었다.

IV. 과제 수행 시의 문제점과 해결 과정 또는 의문 사항

과제 수행을 할 때 sched_info_depart 함수를 찾아 dmesg를 출력하기 위한 코드를 추가했었지만 dmesg에 sched: RT throttling activated라는 로그만 나오고 의도한 출력이 나오지 않는 문제점이 발생했다. 이러한 문제를 해결하기 위해 virtual machine에 새로운 가상 머신을 1개 더 추가해서 새롭게 부팅을 진행하였다. 새로운 가상 머신에서는 dmesg에서 앞서 말한 문제점이 생기지 않았고 과제를 해결할 수 있게 되었다. 하지만 의문 사항이 하나 있었다. 문제를 해결하기 위해 기존의 가상 머신에서 커널의 버전을 확인해보았지만, 과제 0에서 설치했던 커널 버전으로 동일하게 나타났다. 따라서 왜 동작이 진행되지 않았고 새롭게 만든 가상 머신에서는 동작했는지가 의문점으로 남아있다.

두 번째 문제는 fork 이전에 RT-RR 스케줄링을 설정할 때, 자식 프로세스들이 Round Robin 기법으로 적절히 동작하는지 확인할 방법이 없었다. 이 의문점을 해소하기 위해 sched_getattr 함수를 구현하는 코드를 추가하고, 구조체를 하나 더 추가해서 getattr 함수로 현재의 정보를 받는 과정을 만들었다. 그 정보를 통해 sched policy를 출력했을 때 RT-RR 값으로 지정되어있음을 확인하게 되었고 다음 단계로 계속해서 진행할 수 있었다.