[2023] 운영체제 2차 과제 보고서

학과 : 컴퓨터학과

학번 : 2021320126

이름 : 장지윤

제출 날짜 : 2023-06-05

Freeday 사용 일수 : 4일

1. 개발환경

- Host OS : Window 11
- Virtual Box 7.0.6
- Ubuntu 18.04.02 (64bit)
- Linux Kernel 4.20.11

2. 과제 개요

OS는 하나의 process 만 사용하는 것이 아닌, 굉장히 많은 process가 메모리에 올라가 있을 때, CPU가 idle한 상태를 최소화하며 가장 최대로 활용하도록 CPU를 scheduling 해야 한다. Scheduling을 함에 있어서 고려해야 되는 기준들로는 CPU 활용률, 처리량, 응답 시간, 대기 시간, Turnaround time으로 크게 5가지 조건들이 있다. 모든 조건을 만족하며 스케줄링 하는 것은 현실적으로 불가능하며, 이용하는 용도에 따라서 특정한 조건에 더 초점을 맞추게 된다.

대부분의 사람들이 사용하는 개인용 컴퓨터에서는 응답시간에 가장 큰 초점에 맞춰 스케줄링을 하게 된다. 이를 위해서는 I/O 처리시간과 더불어 process들이 동시에 돌아가는 것과 같은 느낌을 주는 것이 가장 중요하기 때문에 모든 process에서 starvation이 발생하지 않게 하는 것이 가장 중요하다. Starvation을 막기 위해 제안된 것이 Round Robin scheduling으로 프로세스들 사이에 우선순위를 두지 않고, 순서대로 Time Slice 만큼 자원을 할당하는 방식의 알고리즘이다. Response Time이 짧고, 모든 프로세스가 공평하게 반드시 실행된다는 장점이 있다. 또한, Time Slice마다 context switching이 발생하므로 이로 인한 오버헤드가 크다는 단점이 있다.

이에, 이번 과제는 행렬 연산을 수행하는 프로그램을 통해 Round Robin scheduling에서 Time slice에 따른 행렬연산 성능변화를 분석한다. multi core로 실험을 진행하면 정확한 context switching 오버헤드 결과를 얻기 힘들어, single core로 제한하여 실험을 진행한다. 또한, 단위 시간당 프로세스가 수행한 행렬 연산량을 정확히 구해내기 위해 Kernel을 수정하여 각 프로세스별 CPU Burst Time을 구하고 이를 기준으로 성능을 분석한다.

3. 성능 분석을 위한 프로그램 구현 (cpu.c)

- cpu.c

  생성할 프로세스의 개수와 각 프로세스의 수행 시간(ms)를 입력 받는 함수로 생성된 프로세스마다 행렬 연산을 수행하는 함수이다. 각 프로세스 별로 행렬 연산을 시작한 시점부터 100ms 마다 메시지를 출력하게 된다. 프로세스 실행이 종료될 때, 프로세스가 연산한 총 횟수와 시간을 출력한다. 첫 프로그램이 실행될 때, 자식 프로세스들을 fork() 함수로 생성하고 clock_gettime 함수를 통해 행렬 연산 시작 시간을 잰다.

이때 rc 값을 체크해 자식 프로세스는 fork()를 수행하지 않으며, 부모 프로세스 내 배열에 저장하여 이후 waitpid()에 이용한다. begin과 start라는 변수에 시작 시간을 넣어 begin은 행렬 연산 시작 후 100ms 마다 시간을 체크하기 위해 사용되며 start는 행렬 연산 수행 시간을 입력받은 프로세스의 수행 시간에 맞추기 위해 사용된다. 행렬 연산이 끝나면 rc 값으로 부모 프로세스라면 waitpid를 이용해 자식 프로세스를 생성 순서대로 수행시킨 뒤, 모든 자식 프로세스가 끝나면 부모 프로세스도 프로그램을 종료하게 된다.

- SIGINT 시그널에 의한 종료 (추가 과제)

프로세스가 SIGINT 시그널에 의해서 종료될 때, 프로세스를 바로 종료하지 않고 지금까지 수행했던 행렬 연산의 총합을 출력한 뒤 종료되도록 구현했다. 컴파일러에게 변수가 변경될 수 있음을 알려주는 volatile 한정자와 시그널 핸들러에서 액세스할 수 있는 타입인 sig_atomic_t 타입을 이용해 interrupted라는 변수를 정의했다. signal(SIGINT, handle_signal)함수를 이용해 SIGINT 시그널이 들어오면 handle_signal 함수가 수행되도록 하였고 handle_signal 함수가 수행되면 interrupted에 1을 저장하도록 설정하였다. 또한, if 문을 통해 interrupted가 1(True)이 되면 진행하던 행렬 연산을 멈추고 지금까지의 연산 총합을 출력하도록 하였다.

4. Time Slice에 따른 행렬연산 성능변화 분석

- 실험 방법

Single 코어만을 사용하도록 설정하고 스케줄링 방식을 Round Robin으로 변경하여, Time Slice에 따른 작업 성능을 분석한다. 위에서 구현한 cpu.c 내에 스케줄링 정책을 RR로 변경하여 실험하고 이를 통해 컨텍스트 스위칭에 따른 오버헤드를 분석한다. Proc을 통해 time slice를 1ms, 10ms, 100ms로 변경해가며 실행하여 변화하는 행렬 연산 횟수를 분석하였으며 각각 Time Slice 별로 5회 반복해 나온 횟수의 평균치로 분석하였다. 모든 실험은 프로세스 개수 2, 동작 시간 30초로 진행하였다.

- 구현 방법

sched_attr 구조체를 정의하고 sched_setattr 시스템 콜(syscall(SYS_sched_setattr, pid,attr, flags))을 이용해 스케줄링 정책을 RT_RR로 설정하여(attr.sched_policy = SCHED_RR) Round Robin scheduling이 진행되도록 한다. 우선순위는 95로 설정한다. Time Slice 변경은 proc을 이용해 1, 10, 100으로 변경하였다.(echo 1(10, 100) > /proc/sys/kernel/sched_rr_timeslice_ms)

```c
struct sched_attr{
        uint32_t size;
        uint32_t sched_policy;
        uint64_t sched_flags;
        int32_t sched_nice;
        uint32_t sched_priority;

        uint64_t sched_runtime;
        uint64_t sched_deadline;
        uint64_t sched_period;
};

static int sched_setattr(pid_t pid, const struct sched_attr* attr, unsigned int flags){
        return syscall(SYS_sched_setattr, pid, attr, flags);
}


int main(int argc, char* argv[]){
        int i, num, time, result;
        struct sched_attr attr;

        memset(&attr, 0, sizeof(attr));
        attr.size = sizeof(struct sched_attr);
        attr.sched_priority = 95;
        //attr.sched_priority = 10;
        attr.sched_policy = SCHED_RR;
        result = sched_setattr(getpid(), &attr, 0);
        if(result == -1){
                perror("Error calling sched_setattr.");
        }
```

그림 1 cpu.c 내 RR 구현

- 실험 결과

각 time slice마다 5번의 실험에서 각 프로세스별 연산량의 평균과 총합을 표와 그래프로 나타낸 결과는 다음과 같다.

RR Time slice	1ms	10ms	100ms
	# of calc.	# of calc.	# of calc.
Process #0	3063	3149	3184
Process #1	3104	3150	3212
Total calc.	6167	6299	6396

그림 2 실험 결과 표

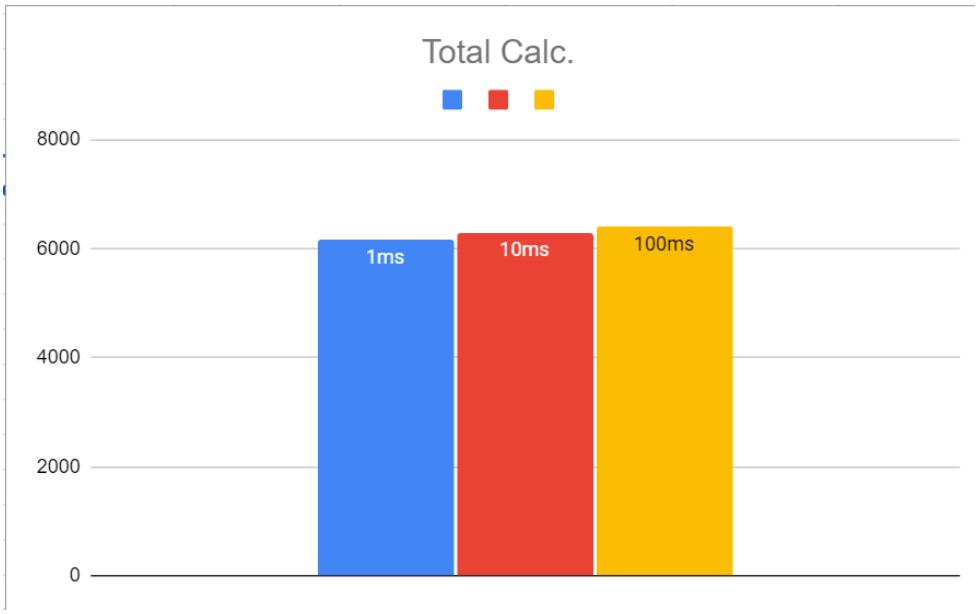


그림 3 실험 결과 그래프

- 결과 분석

표를 보면 같은 Time slice 조건에서 process 0과 process 1은 거의 비슷한 연산량을 보여준다. 이는 모든 process가 공평하게 CPU를 점유할 수 있다는 Round Robin scheduling의 특징을 잘 보여주고 있다.

다른 time slice에서의 결과를 비교하자면 1ms의 경우, 10ms와 비교 시 약 2.14%의 연산 횟수 하락이 발생하였고, 10ms의 경우, 100ms와 비교 시 약 1.54%의 연산 횟수 하락이 발생하였다. Time slice마다 context switching이 발생하여 오버헤드가 생기므로 Time slice가 클수록 context switching 오버헤드가 줄어들고 이로 인해 주어진 시간 내 연산 횟수가 가장 큰 것으로 분석할 수 있다. 해당 결과를 통해 Time slice가 클수록 연산 횟수는 증가하는 것으로 이해할 수 있다. 그러나 이 실험을 통해서는 각 프로세스별 CPU 점유 시간을 알 수 없기에 CPU 점유 시간에 따른 연산량을 비교할 수 없다는 한계가 있다.

5. CPU Burst Time을 활용한 성능변화 분석

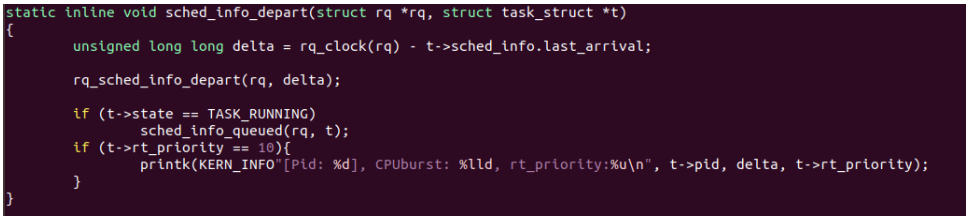
- 실험 방법

이전 실험에서는 설정된 Time Slice에 따른 연산 횟수만을 바탕으로 오버헤드를 분석하였다. 실제로 운영체제에서 프로세스들은 CPU 자원을 점유하는 시간이 제각기 다르기 때문에 각 프로세스별 연산 시간은 우리가 설정한 값과 다르다. 따라서 단위 시간 당 프로세스가 수행한 행렬 연산량을 구하기 위해서는 각 프로세스별로 정확한 CPU Burst Time을 구하고 이를 기준으로 성능을 분석한다. 실험 조건은 이전 실험과 같이 single core에 Round Robin scheduling을 진행하며 time slice 또한 1, 10, 100(ms)

로 변화하며 진행한다. 모든 실험은 프로세스 개수 2, 동작 시간 30초로 진행하였다. 추가로, CPU Burst Time을 구하기 위해서 생성한 모든 프로세스의 우선순위 값을 10으로 설정하고 커널 내 함수를 수정해 우선순위 값이 10인 프로세스의 CPU Burst Time을 printk로 출력하도록 하였다. 모든 프로세스는 종료되면 sched_info_depart 함수가 실행되므로 Time Slice 마다 printk가 진행되고 이를 /var/log/kern.log에서 확인하였다. 해당 출력을 vim과 python으로 엑셀에 옮겨 프로세스별 CPU Burst 값들의 합들의 평균과 연산 횟수의 평균을 통해 부모 프로세스와 자식 프로세스의 CPU Burst를 분석하였다.

- 구현 방법

cpu.c 내 attr.sched_prioirty 값을 10으로 설정하고 linux-4.20.11/kernel/sched/stats.h 내 sched_info_depart 함수를 수정하여 프로세스가 10인 프로세스인 경우 pid와 CPU Burst를 출력하도록 하였다. /var/log/kern.log 파일에서 vim을 이용해 복사한 뒤, 그림 5에 파이썬 코드로 PID와 CPUburst 값을 열로 엑셀화 시켰다.



```
static inline void sched_info_depart(struct rq *rq, struct task_struct *t)
{
        unsigned long long delta = rq_clock(rq) - t->sched_info.last_arrival;

        rq_sched_info_depart(rq, delta);

        if (t->state == TASK_RUNNING)
                sched_info_queued(rq, t);
        if (t->rt_priority == 10){
                printk(KERN_INFO"[Pid: %d], CPUburst: %lld, rt_priority:%u\n", t->pid, delta, t->rt_priority);
        }
}
```

그림 4 stats.h 내 CPU Burst 출력 구현

```python
import re
import pandas as pd

# 로그 데이터
log_data = """
Jun  5 16:37:39 osta-VirtualBox kernel: [ 5329.551233] [Pid: 2709], CPUburst: 2584939, rt_priority:10
Jun  5 16:37:39 osta-VirtualBox kernel: [ 5329.554224] [Pid: 2710], CPUburst: 2994324, rt_priority:10
Jun  5 16:37:39 osta-VirtualBox kernel: [ 5329.557872] [Pid: 2709], CPUburst: 3648394, rt_priority:10

"""

# PID와 CPU Burst 값을 추출하는 정규표현식
pattern = r"₩[Pid: (₩d+)₩], CPUburst: (₩d+)"

# 추출된 데이터를 저장할 리스트
pid_list = []
cpuburst_list = []

# 로그 데이터에서 PID와 CPU Burst 값을 추출하여 리스트에 저장
matches = re.findall(pattern, log_data)
for match in matches:
    pid_list.append(match[0])
    cpuburst_list.append(match[1])

# 추출된 데이터를 DataFrame으로 변환
data = {"PID": pid_list, "CPU Burst": cpuburst_list}
df = pd.DataFrame(data)

# DataFrame을 Excel 파일로 저장
df.to_excel("output.xlsx", index=False)
```

그림 5 출력 엑셀화 구현

- 실험 결과

각 time slice 별로 5번의 실험에서 각 프로세스별 연산량과 CPU burst 시간의 평균을 표와 그래프로 나타낸 결과는 다음과 같다.

RR Time slice	1ms		10ms		100ms	
	# of calc.	Time(s)	# of calc.	Time(s)	# of calc.	Time(s)
Process #0	3141	15.02174327	3181	15.03658076	3305	14.99916417
Process #1	3130	14.98420946	3185	14.97988505	3334	15.11134743
Total calc. and Time	6271	30.00595273	6366	30.01646581	6639	30.1105116

RR Time slice	1ms	10ms	100ms
Calculations per second	208.9918642	212.0835957	220.4877847
Baseline = 1ms	100	101.4793549	105.5006546
Baseline = 10ms	98.54221093	100	103.9626775
Baseline = 100ms	94.78614177	96.18836525	100

그림 6 실험 결과 표

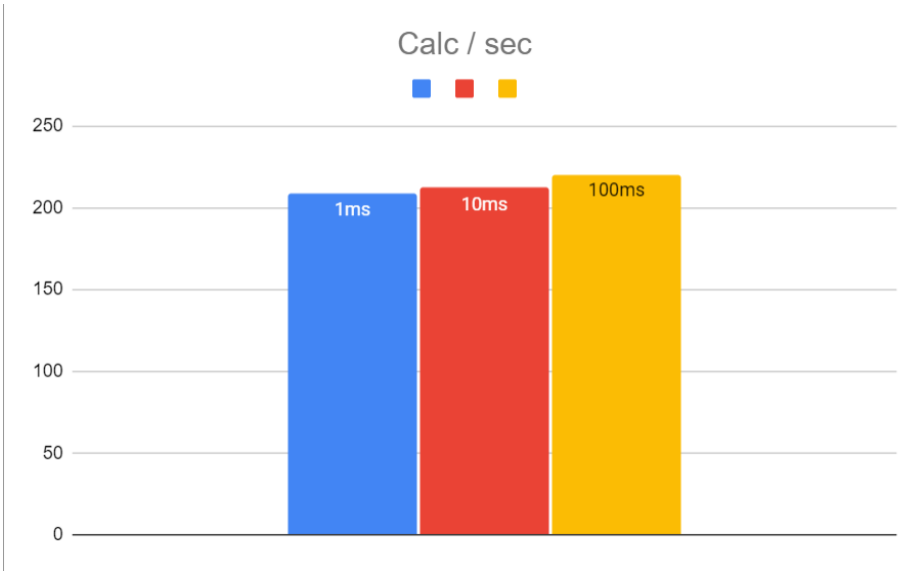


그림 7 실험 결과 그래프

- 결과 분석

표를 보면 같은 Time slice에선 연산량과 CPU Burst time이 비슷한 수치를 보임을 확인할 수 있다. 이는 위 실험과 같이 Round Robin scheduling의 특징을 잘 보여주고 있다.

다른 Time slice에서는 Time slice가 증가할수록 연산량과 함께 CPU Burst time 또한 증가하는 것을 확인할 수 있다. 이는 Time slice가 증가함에 따라 context switching 오버헤드가 감소하며 이에 따라 CPU Burst time과 연산량이 모두 증가한 것으로 확인할 수 있다. 특히, Time slice 100ms와 1ms에서의 CPU Burst time을 비교했을 때 약 0.1초의 차이를 보인다.

CPU Burst time 당 연산 횟수를 비교해봤을 때, Time slice가 커짐에 따라 CPU Burst time 당 연산 횟수도 커짐을 확인할 수 있다.1ms의 경우, 10ms와 비교 시 약 1.46%의 CPU Burst Time 당 연산량 하락이 발생하였고, 10ms의 경우, 100ms와 비교 시 약 3.72%의 CPU Burst time 당 연산량 하락이 발생하였다. 일반적으로 CPU Burst Time과 연산량 사이에는 상관관계가 존재한다. CPU Burst Time이 길면 더 많은 작업을 수행할 수 있기 때문이다. 그러나 Round Robin scheduling에서는 Time slice에 따라 프로세스가 CPU를 사용하는 시간이 제한되므로, 한 번에 수행하는 연산량도 제한된다. 이로 인해 Time slice가 연산량에 영향을 줄 수 있다. 즉, Time slice 변화량에 따른 CPU Burst time 변화량이 Time slice 변화량에 따른 연산량 변화량보다 작기에 이러한 수치가 나옴을 확인할 수 있다.

이러한 결과로 Time slice가 CPU Burst Time과 연산량에 직접적인 영향을 미치는 것을 보여준다. Time slice가 증가할수록 context switching 오버헤드의 감소로 CPU Burst

Time과 연산량은 증가하지만, 프로세스 응답 시간이 상대적으로 느려질 수 있다. 따라서 Time slice가 적절하게 설정되어야 CPU를 효율적으로 활용할 수 있고 연산량 또한 효율적으로 높아질 수 있다.

6. 숙제 수행 과정 중 발생한 문제점과 해결방법

printk를 dmesg 터미널 화면으로 분석하였으나 Time slice가 1인 경우 printk가 너무 많아 화면에 제거되는 부분이 존재하였다. 이에, /var/log/kern.log 파일에서 printk를 복사해 파이썬 프로그램으로 엑셀화시켜 해결할 수 있었다.