OS 2차과제 보고서

컴퓨터학과 2020320132 조민규

2023.05.17 Freeday 0day

-과제 개요

1. clock_gettime을 이용해 process 수행시간 측정 및 fork함수를 통한 프로세스 생성 및 실행과정 이해

1-1. 추가과제: signal 함수를 이용한 signal 핸들러 구현

2. cpu core를 1개로 제한 후 sched_setattr함수를 이용해 round robin scheduling 정책을 선택해 time slice를 변화시켜가며 성능변화 측정

3. 앞선 실험에선 context switching 등의 이유로 clock_gettime을 이용해 구한 시간과 실제 프로세스의 cpu 점유 시간은 달라진다. 따라서 sched_info_depart 커널 코드를 수정해 cpu burst time을 로그로 출력해서 성능변화를 측정한다.

-round robin 스케줄링과 time slice

현재 과제 환경에서는 cpu.c 프로세스는 입출력을 받지 않고 행렬연산만 수행 하기 때문에 time slice가 커지면 context siwitching에 소요되는 시간이 증가해 연산량이 감소한다.

-소스코드 리뷰(3번 과제까지 진행했을 때 기준)

1. calc()함수

```c
int calc(){
    int matrixA[ROW][COL];
    int matrixB[ROW][COL];
    int matrixC[ROW][COL];
    int i,j,k;
    count = 0;
    struct timespec begin, end, entire;
    clock_gettime(CLOCK_MONOTONIC, &begin);
    clock_gettime(CLOCK_MONOTONIC, &entire);

    while(1){
        for(i = 0;i<ROW;i++){
            for(j=0;j<COL;j++){
                for(k=0;k<COL;k++){
                    matrixC[i][j] +=matrixA[i][j]*matrixB[i][j];

                }
            }
        }
        count++;
        clock_gettime(CLOCK_MONOTONIC, &end);
        execute_time = (end.tv_sec - entire.tv_sec)*1000+(end.tv_nsec - entire.tv_nsec)/1000000.0;
        double time = (end.tv_sec - begin.tv_sec)*1000+(end.tv_nsec - begin.tv_nsec)/1000000.0;
        if(time>100){
            printf("PROCESS # %02d count = %d time = %f\n",my_pid, count, time);
            clock_gettime(CLOCK_MONOTONIC, &begin);
        }
        if(execute_time > exetime*1000){
            printf("DONE!! PROCESS # %02d totalCount = %d time = %f\n",my_pid,count,execute_time);
            exit(0);
        }
    }
}
```

우선 전체 시간과 현재epoch(100ms)를 모두 측정해야 하기 때문에 begin(epoch)과 entire(전체)라는 변수를 선언 후 시작 시간을 기록한다. 그 후 count가 증가 될 때 마다 clock_gettime을 이용해 end라는 변수에 현재 시각을 저장 후 시간간격(time)을 ms단위로 계산한다. 이 값이 100을 넘을 경우 로그를 출력하도록 한다. 또 전체 프로세스 실행시간(execute_time)을 계산해 이 값이 처

음에 bash에서 넘겨준 실행시간(exetime)을 넘을 경우 현재까지 연산량을 출력 후 종료한다.
-문제: 처음에는 정확한 시간을 재기 위해서 pthread를 기반으로 멀티스레드를 구현해 시간을 기록했으나 thread에 새로운 pid가 할당되어 추후 과제에서 제대로 된 성능 분석이 어려워 for문에 루틴을 넣어 기록했다. 따라서 또한 가장 안쪽 for문에 넣어봤지만 clock_gettime의 잦은 호출 때문에 count값이 적게 나와 성능변화를 분석하기 어려워져서 가장 바깥쪽에 넣었다. 따라서 epoch 실행시간과 100ms 간 오차가 커졌다.

2. 구조체 선언

```c
int proc_num;
int exetime;
int my_pid;
double execute_time;
int count;
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);
}
```

2차 과제에서 프로세스의 priority와 schedule정책을 설정해야 하는데 sched_setattr()함수는 wrapping함수를 제공해주지 않아서 직접 구조체와 함수를 작성해야했다.

3.main함수 및 signalhandler

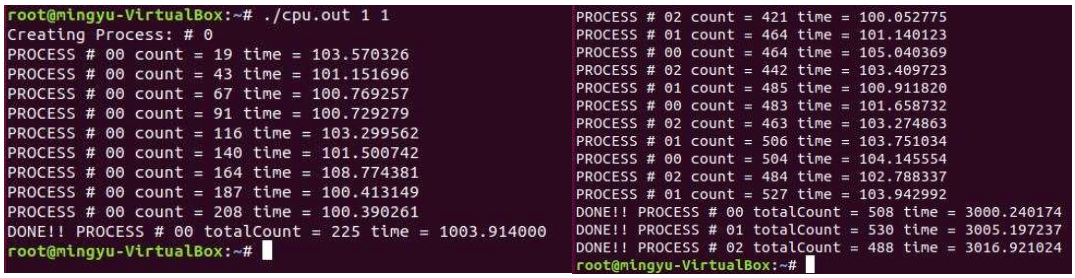
```c
int main(int argc, char* argv[]){
    struct sched_attr attr;

    memset(&attr,0,sizeof(attr));
    attr.size=sizeof(attr);
    attr.sched_priority=10;
    attr.sched_policy = SCHED_RR;
    if(sched_setattr(getpid(),&attr,0)==-1){
        int err=errno;
        char * msg;
        msg=strerror(err);
        printf("sched set error: %s\n",msg);
    }

    proc_num=atoi(argv[1]);
    exetime=atoi(argv[2]);
    signal(SIGINT,handle_sigint_parent);
    pid_t pid;
    for(int i=0;i<proc_num;i++){
        my_pid=i;
        printf("Creating Process: # %d\n",my_pid);
        pid=fork();
        if(pid==0){
            break;
        }
    }
    if(pid==0){
        signal(SIGINT,handle_sigint);
        calc();
    }
    for(int i=0;i<proc_num;i++){
        wait(NULL);
    }
}
```

```c
void handle_sigint(){
    printf("DONE!! PROCESS # %02d totalCount = %d time = %f\n",my_pid,count,execute_time);
    exit(0);
}
void handle_sigint_parent(){
    for(int i=0;i<proc_num;i++){
        wait(NULL);
    }
    exit(0);
}
```

우선 sched_setattr함수를 호출해 스케줄링 정책을 설정 후 signal handler를 등록한다. 또 정해진 프로세스 수만큼 fork를 호출한다. 이후 자식 프로세스는 calc함수를 실행하고 부모프로세스는 자식프로세스가 종료될때까지 기다린다. 또 signalhandler는 부모프로세스와 자식프로세스에게 다르게 등록했는데 이는 SIGINT를 입력 받았을 때 또한 부모프로세스가 자식프로세스보다 먼저 종료되는 경우를 방지하기 위함이다.(bash입력창의 혼란이 생김)

-Time slice 변화에 따른 성능 분석 결과

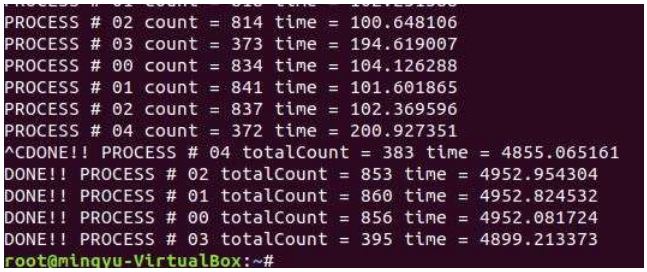
1. 과제 1번



왼쪽에서 차례로 ./cpu.out 1 1, ./cpu.out 3 3을 실행한 모습

앞서 설명했듯이 for문 가장 바깥쪽에 시간재는 루틴을 넣어서 time에 오차가 발생했다.

1-1 추가과제

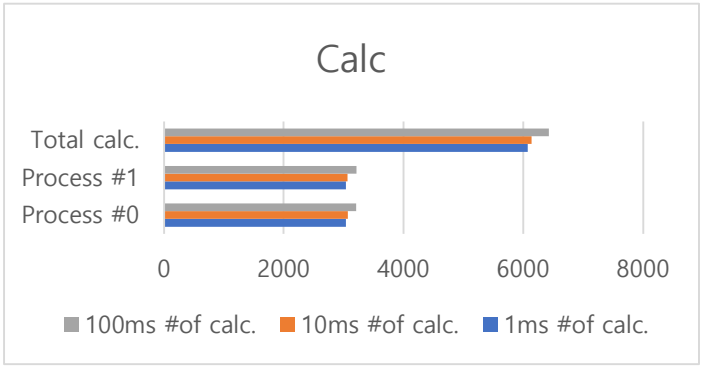


./cpu.out 5 5를 실행중에 ctrl+c를 눌러 강제 종료한 모습

강제 종료 시에 현재까지 수행한 연산결과를 출력 후 종료.

2. 과제 2번

RR Time Slice	1ms	10ms	100ms
	#of calc.	#of calc.	#of calc.
Process #0	3038.2	3072.6	3211.2
Process #1	3039.2	3069	3218.4
Total calc.	6077.4	6141.6	6429.6



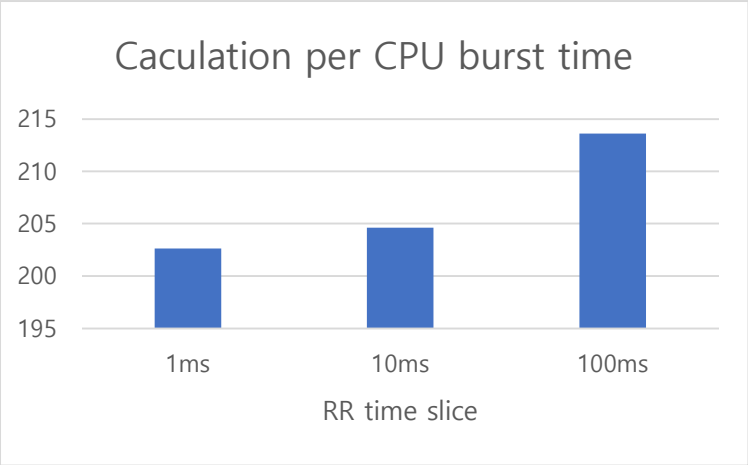
각 time slice별로 5번씩 실행한 결과에 평균을 사용했다.

1ms에서 10ms로 변화했을 때는 대략 1.01%의 성능이 향상됐고, 10ms에서 100ms로 변화했을 때는 대략 1.05%의 성능 향상을 보였다. 이는 앞서 예측했듯이 잦은 문맥교환에서 발생하는 성능하락으로 분석할 수 있다.

3. 과제 3번

RR Time Slice	1ms		10ms		100ms	
	#of calc.	Time(s)	#of calc.	Time(s)	#of calc.	Time(s)
Process #0	3038.2	15.00171	3072.6	15.00231	3211.2	15.02685
Process #1	3039.2	14.98968	3069	15.01127	3218.4	15.07432
Total calc. and Time	6077.4	29.99139	6141.6	30.01358	6429.6	30.10117

RR Time Slice	1ms	10ms	100ms
Caculation per CPU burst time	202.6382	204.6274	213.5997
Baseline=1ms	100	100.9817	105.4094
Baseline=10ms	99.02788	100	104.3847



3번 과제도 2번과 마찬가지로 5번 실행한 결과에 평균을 사용했다. 그러나 과제 예시에 나온 표는 calculations per second값을 사용했는데 이는 처음에 지정한 30초를 이용한 결과이다. 따라서 3번과제의 본 취지와는 맞지 않다고 생각해 calculation per cups burst time(연산 횟수/cpu burst time)을 사용했다. 또 그래프에서도 해당 변량을 그대로 사용했다. 이렇게 분석했을 때 1ms에서 10m로 갔을 때 0.98%의 성능향상을 보였고, 10ms에서 100ms로 변할 때는 4.38%의 성능향상을 보였다.

1ms-10ms의 성능 차이가 10ms-100ms의 성능차이가 큰이유는 다음과 같은 것으로 예상된다.

CPUburst : 4709740, CPUburst : 11782369, CPUburst : 100484638,
CPUburst : 3865373, CPUburst : 12579427, CPUburst : 100102722,
CPUburst : 3932574, CPUburst : 11734757, CPUburst : 99808091, r
CPUburst : 3474427, CPUburst : 12611892, CPUburst : 103613341,
CPUburst : 4070068, CPUburst : 11781177, CPUburst : 99964340, r
CPUburst : 4172719, CPUburst : 11664977, CPUburst : 100482369,

이는 차례대로 time slice가 1ms, 10ms, 100ms일 때의 커널메시지의 일부분인데 cpu burst time이 대략 4ms, 10ms, 100ms정도이다. 이는 설정된 time slice를 1로 설정했을 때 time slice가 너무 짧아 나타나는 현상으로 예상된다. 따라서 이러한 이유 때문에 1ms-10ms간 성능차이가 예상보다 적게 나온 것으로 생각된다.

-추가적인 사항

1. dmesg의 버퍼 크기

우선 1ms같은 경우 30초간의 커널 메시지를 dmesg버퍼가 모두 담지 못해 앞부분 메시지가 잘리는 현상이 발생했었다. 이를 해결하기 위해서 터미널을 한 개 더 실행하여 ./cpu.out의 실행 중에 "dmesg -c > log.txt"명령어를 3번 정도 실행해 3번에 나누어 커널 메시지를 기록했다.

2. 또 커널메시지가 4000줄이 넘기 때문에 일일이 cpu burst time의 합을 계산하기 힘들어 아래와 같은 python프로그램을 작성해 자동화했다.

```
result=[0,0,0]

filename='/Users/PC/Desktop/os/log/time1_1.txt'
with open(filename,'r') as file:
    for line in file:
        parent = int(line.split(':')[1].split(']')[0])
        break
    for line in file:
        pid=int(line.split(':')[1].split(']')[0])
        bursttime=int(line.split(':')[2].split(',')[0])/1000000000
        result[pid-parent]+=bursttime
print(result)
with open(filename, 'a') as file:
    file.write('\n')
    file.write('process#00 : {}, process#01 : {}'.format(result[1],result[2]))
```

3. 또한 sched_setattr함수나 cpucore수 제한 등 다양한 측면에서 sudo 권한을 필요로 했기 때문에 해당 과제는 모두 root계정에서 실행했다.