운영체제 2차 과제 보고서

학과 : 컴퓨터학과

학번 : 2018320225

이름 : 이지수

제출 날짜 : 2022. 12. 04

Freeday 사용 일수 : 0일

목 차

- 개발 환경

AMD Ryzen 3 PRO 4350G, Windows 10

Oracle VM VirtualBox, Ubuntu (64-bit) 18.04.2, Linux Kernel (ver 4.20.11)

- 과제 개요 및 Round Robin 스케줄링에서 time-slice의 영향 설명

이번 과제에서는 fork를 이용해 여러 process를 생성하는 user program을 작성하고 각 process를 다른 time-slice로 round robin 스케줄링하며 process별 연산의 횟수와 cpu burst를 측정해본다.

Round Robin 스케줄링에서 현재 cpu를 사용하고 있는 process는 할당받은 time slice(time quantum)을 모두 사용하면 다른 process에게 cpu를 넘겨주어야 한다. 여기서 현재 프로세스의 상태/Context를 저장하고 다른 프로세스의 Context를 실행해야 하는데 이를 Context Switching이라 한다. 만약 time-slice가 너무 짧아 Context Switching이 너무 자주 발생한다면 이로 인한 오버헤드로 인해 전체적인 성능이 떨어지게 될 것이다. 반면 time-slice가 매우 클 경우 FCFS 스케줄링과 비슷한 스케줄링 방식이 될 것이다. 이번 과제에서는 time-slice를 1ms, 10ms, 100ms로 조정하며 time-slice에 따른 성능 변화를 측정해 보고자 한다.

- 작성한 모든 소스코드 및 작업 내용에 대한 설명

진행한 작업에 대한 순서와 내용들은 아래와 같다.

1. cpu.c 작성

먼저 행렬 연산을 통해 성능 평가에 사용될 user program인 cpu.c를 작성하였다. 소스 코드에 대한 간략한 설명들은 주석으로 작성하였고 전체적인 코드의 흐름에 대해서 설명하고자 한다.

```
int main(int argc, char **argv){
   int numProcess = atoi(argv[1]);
   int givenTime = atoi(argv[2]); /
```

Main 함수에서 command line input으로 process의 수와 수행 시간을 입력받아 변수에 저장한다.

```
int result;
struct sched_attr attr;
memset(&attr, 0, sizeof(attr));
attr.size = sizeof(struct sched_attr);
attr.sched_priority = 10;
attr.sched_policy = SCHED_RR; // 스케줄링 정책
result = sched_setattr(getpid(), &attr, 0);
```

main함수의 부모 process에서 자식 process를 fork하므로 그 이전에 스케줄링 policy를 과제에서 요구한 round robin 방식으로 설정하여야 한다. 차후 cpu burst 측정을 위한 dmesg 출력 시에 cpu.c에서 생성된 process를 구분하기 위해 priority = 10으로 지정하였다. 위의 코드가 이러한 작업들을 수행한다.

```
for(int i=0; i<numProcess; i++){ // 입력받은 process 개수만큼
        printf("Creating Process: #%d\n", i);
        pid_t pid;
        pid = fork();
        if(pid==0){ // 생성된 자식프로세스인 경우에만 calc함수 수행
                calc(givenTime, getpid() - parentPid -1);
                break;
        }
}
```

스케줄링 policy setting에 error가 없었다면 다음 for문에서 실제로 child process들을 생성한다. 자식 process의 pid ==0 이고 부모 process의 pid != 0이기 때문에 pid와 0의 비교를 통해 프로세스를 구분한다. 부모 프로세스라면 for문을 돌며 입력받은 프로세스 개수만큼 child process를 생성한다. 자식 프로세스인 경우에는 행렬 연산을 위한 calc 함수를 수행하고 다시 for문 처음으로 돌아가 프로세스를 생성하는 것을 방지하기 위해 break를 건다. 여기 calc함수의 인자로 입력받은 수행 시간과 process 번호를 넘기는데 새로 생성되는 process의 pid가 바로 직전에 생성된 process의 pid와 1씩 차이난다는 점을 이용하여 위와 같은 형식으로 구성하였다. 예를 들어, 첫 번째 자식 process는 parentPid와 1이 차이나서 getpid() – parentPid = 1이 되어 getpid() – parentPid -1 = 0 즉 0번 프로세스가 된다.

자식 process의 행렬 연산을 위한 calc 함수의 주요 내용은 아래와 같다.

```
struct timespec start, end;
clock_gettime(CLOCK_REALTIME, &start); // 시작 시간을 start에 저장
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][k] * matrixB[k][j];
         }
      }
      count++; // 2중 for문이 돈 이후 count을 증가시켰음
      clock_gettime(CLOCK_REALTIME, &end);// count가 증가할때마다 현재 시각을 측정하여 end라는 변수에 넣고
      elapsedTime = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_nsec - start.tv_nsec) / 1000000;
      if(elapsedTime / 100 >= quantum){ // 100ms quantum 구간을 넘어갈때마다 현재 count를 출력하고 quantum
         ++quantum;
         printf("PROCESS #%02d count = %d 100\n",cpuid, count);
      }
      if(elapsedTime>=time*1000 || play==0){ // 입력된 시간이 초 단위이므로 1000을 곱하여 ms단위로 맞추고 비교하
         printf("DONE!! PROCESS #%02d : %06d %ld\n", cpuid, count, elapsedTime);
         return 0;
      }
   }
}
return 0;
```

3중 for문 이후에 count를 증가시키는 과제 예시의 방법은 연산 수가 너무 적어서 내부의 2중 for문이 끝날 때마다 count를 증가시키는 방법으로 수정하였다. Count가 증가할 때마다 현재 시각을 측정하여 ms단위로 바꾸어 elapsedTime이라는 변수에 저장한다. 100ms가 경과할때마다

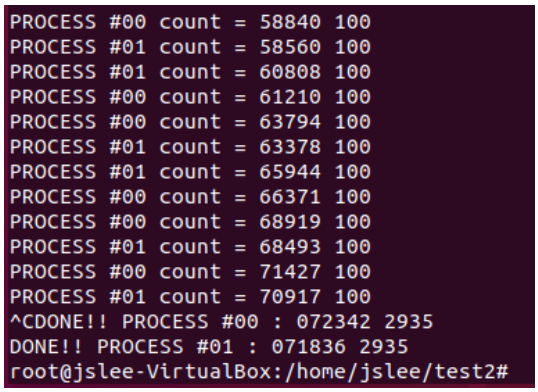
count값을 출력해야 해서 처음에는 elapsedTime % 100 == 0와 같은 방식으로 구현하였으나 time slice 설정값에 따라 정확하게 100ms 단위에 cpu를 할당받지 못하는 경우 출력이 발생하지 않는다는 점을 고려하여 quantum이라는 변수를 두고 elapsedTime의 100ms를 한 quantum삼아서 각 quantum에서 한 번 출력이 되는 즉시 quantum값을 증가시켜 100ms 구간마다 한 번씩만 출력되게끔 구현하였다. 그리고 주어진 수행 시간을 경과하면 함수를 종료시킨다.

- 추가 수행 내용 : Signal Handling

```
int play=1; // signal h

void signalHandler() {
   play=0;
} // main 함수의 signal함

signal(SIGINT, signalHandler);

if(elapsedTime>=time*1000 || play==0){
   printf("DONE!! PROCESS #%02d : %06d
   return 0;
}
```

해당 과제에서 signal handling을 구현한 내역은 위와 같다. c언어의 Ctrl+C의 입력 감지는 signal 함수와 SIGINT를 통해 가능하고 이를 감지했을 때 signal함수의 2번째 인자인 특정 콜백 루틴을 실행한다. 이를 signalHandler()라는 함수로 지정하였고 기존에 1로 선언한 play라는 전역 변수를 0으로 바꾸는 역할을 한다. 그래서 ctrl+c의 입력이 감지되어 play가 0으로 바뀐다면 calc함수의 종료조건 검사에서 play==0이 true가 되어 process를 종료시키는 역할을 하도록 구현하였다.

Signal handling의 결과는 아래 사진과 같다.



2. CPU 코어 설정 및 time slice 변경을 통한 성능 테스트

SSH 세션에서 CPU 코어를 1개로 설정하였다. CPU 코어를 1개로 설정하기 전과 후의 연산 수 비

교 결과는 아래와 같다.

<변경 이전>

```
DONE!! PROCESS #00 : 727691 30000
DONE!! PROCESS #01 : 730737 30000
```

<변경 이후>

```
DONE!! PROCESS #00 : 380126 30046
PROCESS #01 count = 387378 100
DONE!! PROCESS #01 : 387378 30000
```

확인 결과 연산량이 절반 가량으로 줄어들었다. 이후 time slice를 각각 1ms, 10ms, 100ms로 변경하며 연산량의 변화를 확인하였고 1ms에선 총 730568번, 10ms에서는 총 758813번, 100ms에서는 총 767950번의 연산 결과를 보였다. 1ms < 10ms < 100ms 순으로 연산량이 많음을 확인할 수 있었다.

3. CPU burst 측정

/usr/src/linux-4.20.11/kernel/sched/stats.h의 sched_info_depart 함수에 다음과 같은 코드를 추가하였다.

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

Cpu.c에서 process의 rt_priority를 10으로 설정해놓았기 때문에 해당 process에 대한 cpu burst값을 dmesg로 확인하기 위한 작업이다. 파일 수정 이후 커널을 다시 컴파일한 이후 cpu.c를 실행하였다. 출력되는 dmesg들을 log.txt 파일에서 확인한 결과는 아래와 같다.

```
[ 1759.731046] [Pid: 2530], CPUburst: 103371135, rt_priority:10
[ 1759.847469] [Pid: 2529], CPUburst: 116421562, rt_priority:10
[ 1759.959663] [Pid: 2530], CPUburst: 112195295, rt_priority:10
[ 1760.063008] [Pid: 2529], CPUburst: 103342811, rt_priority:10
[ 1760.167673] [Pid: 2530], CPUburst: 104663679, rt_priority:10
[ 1760.276141] [Pid: 2529], CPUburst: 108469888, rt_priority:10
[ 1760.379709] [Pid: 2530], CPUburst: 103569576, rt_priority:10
[ 1760.510652] [Pid: 2529], CPUburst: 130941987, rt_priority:10
[ 1760.615205] [Pid: 2530], CPUburst: 104553833, rt_priority:10
[ 1760.738741] [Pid: 2529], CPUburst: 123533762, rt_priority:10
[ 1760.873171] [Pid: 2530], CPUburst: 134432114, rt_priority:10
[ 1760.987505] [Pid: 2529], CPUburst: 114332724, rt_priority:10
[ 1761.162434] [Pid: 2530], CPUburst: 174930051, rt_priority:10
[ 1761.162695] [Pid: 2529], CPUburst: 263905, rt_priority:10
[ 1761.253692] [Pid: 2530], CPUburst: 90993889, rt_priority:10
[ 1761.253761] [Pid: 2528], CPUburst: 72295, rt_priority:10
jslee@jslee-VirtualBox:~/test2$
```

Time slice를 변경할 때마다 log.txt 파일을 비워가며 각각의 dmesg들을 파이썬 프로그램을 통해 CPUburst값만을 추출한 뒤 pid가 가장 작은 process(위의 예시에서는 Pid: 2528)는 부모 process이므로 해당 pid값을 가진 값들을 전부 제외하고 엑셀에서 홀수 행과 짝수 행의 합계를 계산하였다. Round Robin 스케줄링에서 두 개의 프로세스가 CPU를 번갈아가며 사용하고 있기 때문에 홀수 행과 짝수 행의 burst값의 합계가 각 프로세스의 CPU burst time이다.

각 time slice 별로 CPU burst time의 합계는 아래와 같았다.

<1ms>

15121675031	#0
14865185627	#1
29986860658	#0 + #1

<10ms>

15211345887	#0
14782252011	#1
29993597898	#0 + #1

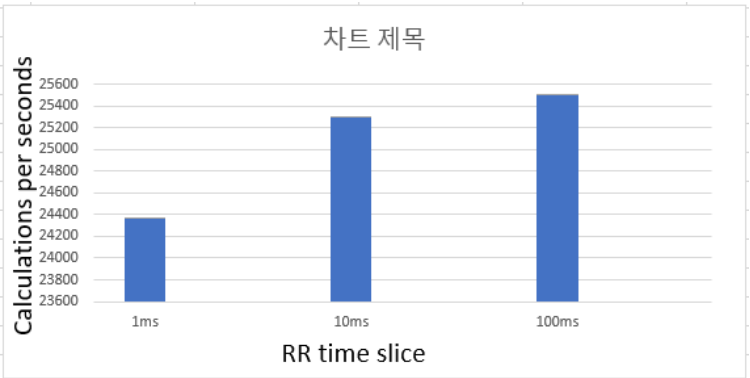
<100ms>

14841921890	#0
15278261784	#1
30120183674	#0 + #1

- time-slice 의 변화에 따른 성능을 관찰한 표 및 결과 분석

	A	B	C	D	E	F	G
1	RR Time Slice	1ms		10ms		100ms	
2		# of calc.	Time(s)	# of calc.	Time(s)	# of calc.	Time(s)
3	Process #0	368529	15.12167503	373158	15.21134589	382291	14.84192189
4	Process #1	362039	14.86518563	385655	14.78225201	385659	15.27826178
5	Total calc. and Time	730568	29.98686066	758813	29.9935979	767950	30.12018367
6							
7	RR Time Slice	1ms		10ms		100ms	
8	Calculations per seconds	24362.9371		25299.16559		25496.19246	
9	Baseline=1ms	100.00%		103.84%		104.65%	
10	Baseline=10ms	96.30%		100.00%		100.78%	
11							
12							

위 표는 time slice를 각각 1ms, 10ms, 100ms로 변경해가며 process의 연산 횟수와 CPU burst를 측정한 결과이다. 연산 횟수를 Time(CPU burst)으로 나누어 계산한 초당 연산 횟수를 각 time slice 별로 그래프화하면 아래와 같다.



Context switching을 위해서 현재 수행중인 process의 state, register 값 등을 저장해야 하고 이 작업의 처리를 위해 시간과 메모리를 소모하는데 이를 context switching overhead라 한다. 위의 결과 표와 그래프에서 이 overhead의 영향을 관찰할 수 있다. Time slice를 각각 1ms, 10ms, 100ms로 변경해가며 ./cpu 2 30을 수행한 결과이다. 먼저 time slice를 짧게 1ms로 설정하였을 때, 두 프로세스의 행렬 연산 합계는 730568번으로 세 time slice 측정값 중 가장 적었고 두 프로세스의 CPU burst time 합 또한 가장 작았다. 이것이 time slice를 너무 작게 설정하여 많은 context switching overhead가 발생하였고 그에 따라 process가 실제로 cpu에서 수행된 시간이 줄어든 결과라고 해석할 수 있을 것이다. 또한 time slice를 1ms로 설정하였을 때는 cpu에서 switching이 굉장히 많이 발생하기 때문에 log.txt에 기록된 dmesg들도 굉장히 많은 것을 확인할 수 있었다. 반면 time slice를 100ms로 설정하였을 때에는 두 프로세스의 행렬 연산 합이 767950번, CPU burst time의 합은 30초가 살짝 넘는 시간으로 가장 큰 값을 보였다. 세 결과값을 Calculations per seconds를 기준으로 비교하였을 때, 10ms에 비해 100ms는 소폭의 성능을 확인할 수 있었고 1ms에 비해 100ms에서는 약 4~5%의 성능 향상을 관찰할 수 있었다.

- 과제 수행 시의 문제점과 해결 과정 또는 의문 사항

1. Cpu.c의 결과값이 정상적으로 출력되는지 확인해보는 과정에서

```
jslee@jslee-VirtualBox:~/test2$ ./temp 2 3
Creating Process: #0
Creating Process: #1
jslee@jslee-VirtualBox:~/test2$ time:100 PROCESS #00 count = 2435 100
time:100 PROCESS #01 count = 2393 100
time:200 PROCESS #00 count = 4856 100
time:200 PROCESS #01 count = 4828 100
time:300 PROCESS #00 count = 7282 100
time:300 PROCESS #01 count = 7298 100
time:400 PROCESS #00 count = 9848 100
time:400 PROCESS #01 count = 9685 100
time:500 PROCESS #01 count = 11722 100
time:506 PROCESS #00 count = 11713 100
time:600 PROCESS #00 count = 13863 100
```

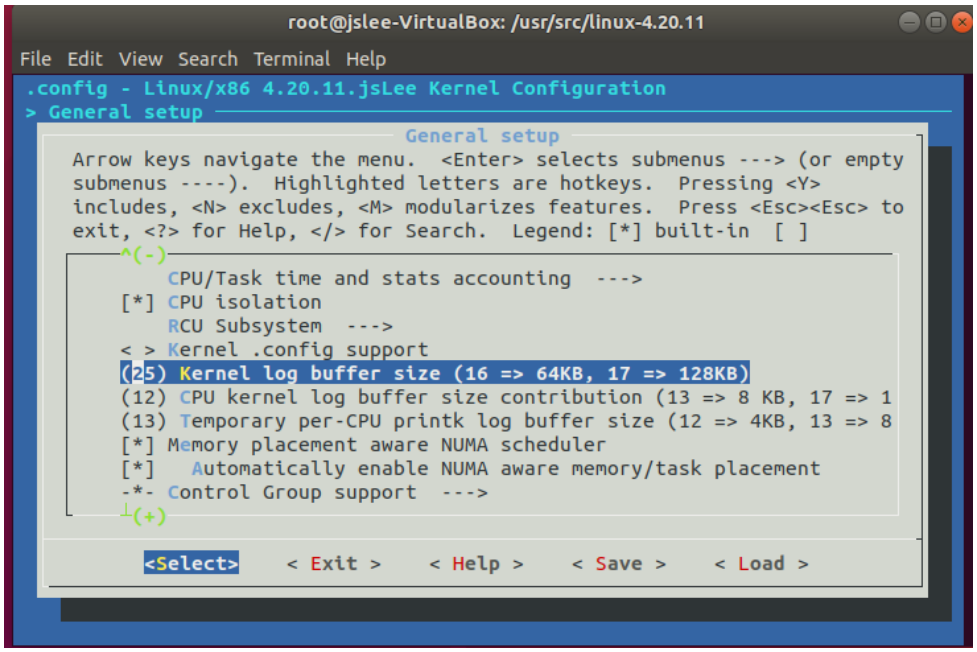
위와 같이 과제에서 모든 process의 실행 결과가 출력된 이후 다음 command를 입력 대기해야하는데 입력을 대기하는 line이 출력 중간으로 섞여버리는 것을 확인하게 되었다. 이를 수정하기 위해서 부모 프로세스가 자식 프로세스들이 모두 종료될 때까지 대기하도록 만들기 위해 main함수의 마지막 부분에 아래와 같은 코드를 추가하였다.

```
while ((wpid = wait(&status)) > 0);
```

<sys/wait.h> 헤더파일에 포함된 wait함수는 자식 프로세스가 종료될 때까지 기다렸다가 자식 프로세스의 pid를 반환하는 함수이다. 만약 현재 수행중인 자식 프로세스가 없다면 에러값 -1을 반환하는데 이 특성을 이용해서 -1이 반환되는 순간 while문에서 탈출하고 부모 프로세스가 종료되도록 구현하여 의도한 출력 결과를 얻을 수 있었다.

2. stats.h의 sched_info_depart 함수를 수정하였음에도 dmesg가 출력되지 않았는데 이는 커널 코드를 수정한 것이므로 커널 컴파일(sudo make install)을 다시 진행하지 않아서 발생한 것이었다.

3. time slice를 1ms로 설정했을 때에 지나친 context switching으로 인해 log가 너무 많이 나와 일부분이 짤려서 cpu burst time의 합계가 30초에 한참 못 미친 16초 가량이 나오는 것을 확인하였다. 이는 kernel configuration에서 kernel log buffer size를 기존 설정 값보다 크게 하여 해결하였다.



kernel에서 sudo make menuconfig > general setup에서 크기를 default값은 18 -> 25로 조정한 후 커널 컴파일을 다시 진행하고 log를 출력해보았더니 time slice 1ms 기준 4000줄 가량 출력되던 log가 7000줄 이상 출력되었고 CPU burst time의 합계 또한 30초 근사치인 것을 확인하였다.