운영체제 3차 과제

Operating System 3rd Assignment Report

- 학과 : 컴퓨터학과
- 학번 : 2015410056
- 이름 : 김지윤

- 제출 일자 : 2017년 6월 11일
- 담당 교수 : 유혁 교수님

1. Procfs

A. Proc File System

리눅스 운영체제에서는 프로세스 정보를 파일 형식으로 저장하는 proc 파일 시스템을 제공한다. 이는 프로세스 아이디 PID에 따라 디렉토리에 프로세스 단위로 저장되고 하위 파일 또는 디렉토리에는 실행상태, 메모리 사용(proc/[PID]/stat), 프로세스에 영향을 미치는 환경 변수들의 이름과 값(proc/[PID]/environ), 해당 프로세스에 의해 시작된 작업들에 대한 하드 링크(proc/[PID]/task)등의 프로세스 정보들이 포함되어 있다. Procfs를 이용하면 사용자레벨에서도 복잡하게 커널로부터 데이터를 요청할 필요 없이 간단한 디렉토리 함수를 이용해 프로세스 상태 및 정보에 접근 할 수 있기 때문에 매우 유용한 시스템이라고 할 수 있다.

a. proc/[PID]/stat

proc/[PID]/stat파일에는 PID, process name, state, nice 등의 프로세스 기본 정보가 차례대로 저장되어 있다. 웬만한 프로세스 정보들은 모두 포함되어 있기 때문에 이번 과제에서 필요한 프로세스 정보를 추출 하기 위해서는 proc/[PID]/stat에만 접근해도 충분하다.

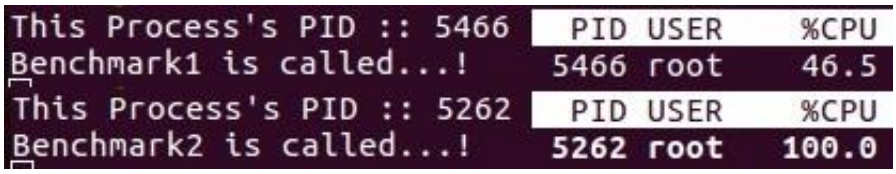
B. Proc에서의 정보 추출

Procfs에서는 실행중인 프로세스가 디렉토리로, 각 프로세스 정보가 하위 파일들로 저장되어 있기 때문에 간단한 디렉토리 함수(opendir(), readdir())를 통해 실행 프로세스들을 파악, 각 프로세스 정보에 접근 할 수 있다. Proc/[PID]/stat 파일의 경우 아래와 같이 정보들이 공백을 마디로 하는 문자열로 저장되어 있기 때문에 문자들을 처음부터 훑으면서 공백 단위로 정보를 나눠서 배열 등에 담고 파일에 정보가 저장된 순서 (예를 들면 process name은 두 번째로 저장되는 정보)를 이용해 필요한 프로세스 정보를 추출한다. 디렉토리 및 파일 접근과 정보 추출의 구현은 **[4.D.c]**, **[4.D.d]**에 설명되어 있다.

```
cat stat
13402 (android.youtube) S 1564 1564 0 0 -1 4194624 7994 1266 5 0 88 21 2 2 20 0
30 0 17338799 146669568 7077 4294967295 32768 36564 3197859624 3197858456 294969
5372 0 4612 0 38120 4294967295 0 0 17 0 0 0 0 0 0
```

2. Benchmark

Benchmark1과 Benchmark2를 각각 실행시키면서 동시에 top 명령어를 통해 CPU 사용률을 살펴보았더니 Benchmark1의 CPU 사용률은 50%가 안되는 데에 반해 Benchmark1의 CPU 사용률은 98%와 100% 사이였다. 따라서 Benchmark2는 CPU bound, Benchmark1은 I/O bound라고 볼 수 있다.



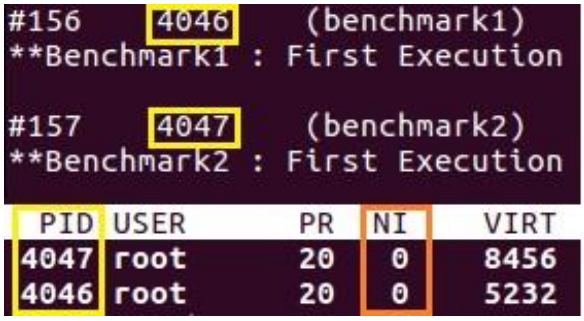
3. CFS Scheduler 수정

A. CFS Scheduler

a. CFS 스케줄링 정책

CFS는 공정한 스케줄링을 위해 각 프로세스마다 CPU 사용 시간에 따라 vruntime이라는 가상 실행시간을 증가시켜 vruntime값이 최소인 프로세스에 우선적으로 CPU를 할당한다. 이때 각 프로세스의 우선순위를 고려하여 CPU를 할당하기 위해 CFS 스케줄러는 nice값에 기반해 vruntime 증가 정도를 계산한다. Nice란 리눅스에서 프로세스의 priority를 표현하기 위한 index값으로 그 범위는 -20부터 +20까지이며 값이 작을수록 높은 priority를 나타낸다. 이 nice값에 따라 프로세스의 vruntime 증가율을 결정하는데, 낮은 nice값(높은 priority)을 가진 프로세스일수록 같은 실행 시간당 계산되는 vruntime의 값은 작다. CFS 스케줄러에선 우선순위가 높은 프로세스의 vruntime이 느리게 증가되고 따라서 CPU를 더 자주, 우선적으로 선점하게 된다.

b. 일반 프로세스의 nice값



CFS 스케줄러에선 nice값에 기반해 프로세스의 우선순위를 고려하여 스케줄링 하지만 특정 프로세스를 제외한 일반적인 프로세스에 대해서 nice 값은 디폴트로 0이 부여된다. 따라서 일반적인 프로세스들에 대해서는 특성에 따른 CPU할당이 수행되지 못한다. **[2]**에서 설명한 바와 같이 Benchmark2는 CPU 사용률이 100%에 육박하는 엄청난 CPU bound 프로세스이지만 I/O bound인 Benchmark1와 동일한 정도로 vruntime값이 증가된다. 이렇게 특성이 매우 다른 프로세스에 대해 동일한 CPU 자원을 할당할 주는 것은 상당히 비효율적이다.

B. CPU, I/O bound에 따른 vruntime 조정

특정 프로세스의 특성을 미리 파악해 그의 vruntime 증가율을 직접 조정한다면 위에서 설명한 것과 같은 비효율적인 CPU 자원할당을 피할 수 있다. switching과 overhead, 그에 따른 손실을 막기위해 CPU 사용률이 높은 CPU bound 프로세스의 vruntime은 조금

씩 누적되게 하고 반대로 I/O bound 프로세스는 같은 실행시간당 vruntime 증가량을 크게 계산하여 누적한다면 기존 CFS 스케줄러가 가진 단점을 보완할 수 있다.

C. 수정 코드 설명

a. fair.c

```
723       /*3rd Assignment code*/
724     if(entity_is_task(curr)) {
725       struct task_struct *curtask = task_of(curr);
726
727       if(curtask->pid == faster_pid) {
728         curr->vruntime += (calc_delta_fair(delta_exec, curr) / 100);
729       }
730       else if(curtask->pid == slower_pid) {
731         curr->vruntime += (100 * calc_delta_fair(delta_exec, curr));
732       }
733       else {
734         curr->vruntime += calc_delta_fair(delta_exec, curr);
735       }
736     }
737     else {
738       curr->vruntime += calc_delta_fair(delta_exec, curr);
739     }
740       ////////////////////////
741
742     update_min_vruntime(cfs_rq);
```

vruntime이 누적 계산되는 update_curr 함수를 수정한다. 우선 현재 프로세스가 task인지 (자신의 고유 run queue를 가지는지 아닌지) 확인 후 task라면 task_of함수를 통해 스케줄링 entitiy 정보를 담는 task_struct 구조체(curtask)를 가지고 온다. 그 구조체 속 pid에 대해(curtask->pid) 지정해준 faster_pid, 혹은 slowe_pid와 일치하는지 확인하고 그렇다면 vruntime 증가량(calc_delta_fair(delta_exec, curr)을 100배 혹은 1/100배로 계산하여 vruntime에 누적시킨다. 경우에 해당되지 않는다면 기존 코드대로 vruntime을 업데이트 시킨다.

b. sched.h

fair.c에서 사용되고 system call에서 선언, 정의된 전역변수를 헤더파일 sched.h에 선언해준다.

```
8   /*3rd Assignment variables*/
9   extern signed int faster_pid;
10  extern signed int slower_pid;
11  ////////////////////////////
```

D. System Call

a. sched_faster.c

PID를 인자로 받아 extern 변수로 선언한 faster_pid에 넣어준다. 시스템콜의 정상 호

출을 확인하기 위해 메시지를 출력한다. 또한 변수에 대해 외부 참조를 허용하기 위해 EXPORT_SYMBOL 매크로를 사용한다.

```c
#include <linux/linkage.h>
#include <linux/kernel.h>
#include <linux/sched.h>

signed int faster_pid;

asmlinkage void sys_sched_faster(int proc_pid) {
  faster_pid = proc_pid;
    printk("Syscall : faster\n");
}

EXPORT_SYMBOL(faster_pid);
```

b. sched_slower.c

sched_faser.c와 마찬가지로 PID를 인자로 받아 extern 변수로 선언한 slower_pid에 넣어주고 메시지메 출력한다.

4. Monitoring Tool

A. 개괄적인 동작 과정

일정 주기(0.5초)마다 Proc 파일 시스템을 이용해 프로세스 정보를 모니터링한다. 프로세스 정보가 포함된 Proc/[PID]/stat에 접근하여 정보를 단위별로 추출하고 그 중에 필요한 정보 (PID, process name, CPU 사용률)만을 DB에 저장한다. 새로 실행되는 프로세스에 대해 초기 정보만을 저장하며 특정 프로세스(Benchmark)가 재실행 될 경우에는 **[3.D]**의 시스템콜 중 프로세스 특성에 맞는 것을 호출한다. 프로세스에 따른 처리는 아래에서 더 자세히 설명하도록 하겠다.

B. Benchmark, 그 외의 프로세스 처리

이 모니터링 툴에서는 특정 프로세스 (Benchmark)에 대해서는 추가적인 동작이 필요하기 때문에 프로세스 정보 중 process name을 이용해 Benchmark와 그 외의 프로세스를 구분하고 각각에 대해 다르게 처리하였다.

a. Benchmark

프로세스 정보를 추출하여 DB에 저장하는 과정은 같지만 Benchmark1, Benchmark2에 대해서는 첫 번째 실행인지, 두 번째 실행인지 확인하여 두 번째 실행이 시작되었다면 DB의 정보를 이용해 시스템콜을 호출하고, monitoring tool 종료 시점을 정하기 위하여 두 프로세스의 두 번째 실행이 모두 종료되었는지 체크하는 추가적인 동작이 필요하다. 따라서 각 benchmark에 대해 프로세스 상태(state)를 저장하는 변수를 만들어 그 값에 따라 단계별로 동작하도록 하였고 (1, 첫 실행, 2. 첫 실행 종료 3. 두 번째 실행 시작 -> 시스템 콜 호출, 4. 시스템 콜 호출 이후 실행 상태, 5. 두 번째 실행 종료) Proc 파일에 접근할 때마다 프로세스가 benchmark인지 확인하여 그렇다면 해당 벤치마크의 state 값을 변경시켰다.

b. 그 외의 프로세스

Benchmark에 대한 추가 동작 외의 것들을 수행한다. Procfs을 이용하여 실행되는 프로세스에 대해 정보를 추출, DB에 저장하는데, 이때 process name을 검사해 이미 DB에 저장된 것인지 확인한다. 만약 중복되는 process name이라면 저장하지 않고 그대로 넘어간다. (이 과정은 두 Benchmark에 대해서도 적용된다.)

자세한 구현 내용은 **[4.D]**에서 설명하도록 하겠다.

C. DB 구현

DB에 저장하는 내용은 index, PID, process name, CPU usage(%) 4가지 정보로 했고 여러 형식이 섞여 있는 것을 string으로 통일해 vector <string>에 저장하였다. DB를 vector <string>으로 구현한 이유는 string 형식을 배열에 쉽게 저장할 수 있고 따라서 index를 이용해 정보별로 접근이 용이하기 때문이다. 또한 일반 array와 다르게 크기를 정하지 않고 뒤에 계속 정보를 붙여 나갈 수 있기 때문에 몇 개의 프로세스가 실행 될 지, 사이즈에 대해 신경 쓰지 않아도 돼서 구현이 비교적 단순해진다. DB의 첫 부분에는 임의의 정보("start")를 저장하여 index 1부터 4칸 단위로 프로세스 정보를 저장했고 각 정보에 따른 접근은 DB에 저장된 정보 수/4 +1, 2, 3, 4로 할 수 있다. (예를 들어 process name 은 DB[DB 정보 수/4 +3]을 index 네 칸 씩 뛰면서 탐색하여 접근 가능하다.

D. 구현 코드 설명

a. 헤더파일 및 함수 prototype, 전역 변수

```
#include <stdio.h>
#include <unistd.h>
#include <cstdlib>
#include <iostream>
#include <cstring>
#include <string>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <vector>
#include <sys/syscall.h>

#define CHECK_MICRO_SEC 500000 //500000 microsec == 0.5 sec
#define FASTER 326
#define SLOWER 327

using namespace std;

void check_dir();
vector<string> infoExtract(char*);
int storeInfo(vector<string>);
double cpuUsageCalc(vector<string>);
int checkBenchmark1(vector<string>);
int checkBenchmark2(vector<string>);
void callSyscall(char*);

int db_cnt = 0;

vector<string> db; //Monitoring TOOL Database
int end = 0;

int bm_1_pid = -1; //pid for  syscall
int bm_2_pid = -1;

int bm_1_stat = 0; //state of benchmark
int bm_2_stat = 0;
```

500000ms (=0.5sec)와 시스템콜 번호는 각각 CHECK_MICRO_SEC, FASTER, SLOWER로 정의했고 DB에 저장된 정보 수(정보가 저장된 최대 index)를 나타내는 db_cnt와 데이터베이스 db, 모니터링 툴 프로그램 종료시점을 알리는 end, 두 벤치마크의 상태를 저장하는 bm_1/2_stat과 두 번째 실행 시 PID를 저장하는 bm_1/2_pid는 전역변수로 지정한다.

b. main()

우선 비어있는 db에 "start"라는 임의의 값을 넣어 프로세스 정보를 index 1부터 네 칸 단위로 저장할 수 있게 한다. end가 2가 될 때까지 0.5초 간격으로 같은 작업(check_dir() 호출)을 반복하고 만약 bm_1/2_stat이 3이라면 (벤치마크가 첫 번째 실행 종료 이후 두 번째 실행이 감지됨) DB를 탐색해 시스템 콜을 호출하는 callSyscall을 호출한다.

```
39  int main()
40  {
41    char* bm1;
42    char* bm2;
43    string str1 = "(benchmark1)";
44    string str2 = "(benchmark2)";
45
46    bm1 = (char *)str1.c_str();
47    bm2 = (char *)str2.c_str();
48
49    db.push_back("start");
50    printf("**MONITORING TOOL START**\n\n");
51    while(end < 2)
52    {
53      check_dir();
54      if(bm_1_stat == 3){
55        callSyscall(bm1);
56      }
57      if(bm_2_stat == 3){
58        callSyscall(bm2);
59      }
60      usleep(CHECK_MICRO_SEC);
61    }
62    printf("**MONITORING TOOL EXIT**\n");
63    return 0;
64  }
```

c. void check_dir()

실행 프로세스 목록을 확인하기 위해 우선 proc 디렉토리에 접근해야 한다.

디렉토리 포인터를 선언하고, 선언된 디렉토리 포인터를 이용하여 해당 디렉토리를 open한다. while문을 이용하여 디렉토리 내 파일을 하나하나 탐색하는데, 이때 필요한 불필요한 디렉토리('.'으로 시작하는 것, 혹은 PID는 숫자로 표현되는데 숫자로 시작하지 않는 것)를 걸러낸다. (line 91~92) 디렉토리 name을 문자열에 삽입해 proc/[PID]/stat의 경로를 만들어 파일이 접근 가능한지 확인하고 해당 위치의 정보를 단위별로 저장하는 InfoExtract함수의 인자로 전달한다. (line 94~98)

```
66  void check_dir()
67  {
68    DIR *dp = NULL;
69
70    struct dirent* entry = NULL;
79    if((dp = opendir("/proc")) == NULL)
80    {
81      printf("Error: Inaccessible Directory\n");
82      exit(0);
83    }
84
85    while(true)
86    {
87      entry = readdir(dp);
88      if(entry == NULL)
89        break;
90
91      if((entry -> d_name[0] == '.') || (entry -> d_name[0] > 57))
92        continue; //except file or dir start with . proc dir start with number
93
94      sprintf(path,"/proc/%s/stat", entry -> d_name); //path <- /proc/PID/stat
95
96      if(access(path, F_OK) != 0)
97        continue;
98
99      process = infoExtract(path);
```

InfoExtract의 리턴 값 (프로세스 정보가 단위별로 저장된 vector<string>)을 이용해 해당 프로세스가 benchmark인지 확인한다. 만약 벤치마크라면 리턴 값은 1이 되는데 그렇다면 bm_1/2_tf를 true로 바꿔준다. bm_1/2_tf의 초기값은 false로 설정되어 있는데 이게 while문 빠져나갈 때까지 유지된다면 현재 벤치마크가 실행 중에 있지 않다는 것이고 이를 이용해 벤치마크 1과 2의 첫 번째 두 번째 실행 종료 시점을 감지한다. 해당 프로세스가 벤치마크이고 두 번째 실행이라면 (bm_1/2_stat의 값이 2, 첫 번째 실행 종료 상태인데 실행되었다면)라면 그 값을 3으로 바꿔주고 이어서 해당 PID를 bm_1/2_pid에 저장한다. (line 111~125)

```
101       /*check if it is benchmark*/
102       temp_val1 = checkBenchmark1(process);
103       temp_val2 = checkBenchmark2(process);
104
105       /*for checking benchmark process exists or not */
106       if(temp_val1)
107         bm_1_tf = true;
108       if(temp_val2)
109         bm_2_tf = true;
110
111       if(temp_val1 && bm_1_stat == 2){
112         bm_1_stat = 3;
113         printf("**Benchmark1 : Second Execution\n\n");
114       }
115       if(temp_val2 && bm_2_stat == 2){
116         bm_2_stat = 3;
117         printf("**Benchmark2 : Second Execution\n\n");
118       }
119       if(temp_val1 && bm_1_stat == 3){
120         bm_1_pid = atoi((char *)process[0].c_str());
121       }
122       if(temp_val2 && bm_2_stat == 3){
123         bm_2_pid = atoi((char *)process[0].c_str());
124       }
```

이후에 InfoExtract의 리턴 값을 storeInfo의 인자로 전달해 호출하고 그 밑의 내용은 앞서 설명한 것과 같이 각 벤치마크의 종료시점을 감지하는 코드이다. 모든 작업이 다 끝나면 open된 디렉토리를 close해준다.

```
126       storeInfo(process);
127     }
128     /*check if benchmark has ended*/
129     if(bm_1_stat == 1 && bm_1_tf == false){
130       bm_1_stat = 2;
131       printf("**Benchmark1 : First Execution End\n\n");
132     }
133     if(bm_2_stat == 1 && bm_2_tf == false){
134       bm_2_stat = 2;
135       printf("**Benchmark2 : First Execution End\n\n");
136     }
137     if(bm_1_stat == 4 && bm_1_tf == false){
138       printf("**Benchmark1 : Second Execution End\n\n");
139       bm_1_stat = 5;
140       ++end;
141     }
142     if(bm_2_stat == 4 && bm_2_tf == false){
143       printf("**Benchmark2 : Second Execution End\n\n");
144       bm_2_stat = 5;
145       ++end;
146     }
147
148     closedir(dp);
149   }
```

d. vector<string> infoExtract(char *)

전달 받은 경로(proc/[PID]/stat)의 파일에 접근하여 단위별로 정보를 추출하는 함수이다. 디렉토리 접근과 동일한 과

정 (파일 포인터 선언, 파일 open ,,, 파일 close.)을 통해 동작하며, **[1.B]**에서 설명한 것과 같이 공백 별로 텍스트를 parsing한다. 각 추출 정보는 vector<string>형에 차례대로 저장하고 그 값을 리턴한다.

```
151  vector<string> infoExtract(char* file_path)
152  {
153    /*Extract information stored in proc/stat by unit*/
154    vector<string> proc_info;
155    FILE *fp = NULL;
156    char buffer[700];
157    char *cp;
158
159    if((fp = fopen(file_path, "r")) == NULL)
160    {
161      printf("Error: Inaccessible File\n");
162      exit(0);
163    }
164    fgets(buffer, 700, fp);
165    cp = strtok(buffer, " ");
166    while (cp != NULL)
167    {
168      proc_info.push_back(cp);
169      cp = strtok(NULL, " ");
170    }
171
172    fclose(fp);
173    return proc_info;
174  }
```

e. int storeInfo(vector<string>)

단위 별로 추출된 정보를 이용하여 필요 정보만을 DB에 저장하는 함수이다. Index를 통해 PID와 process name, CPU 사용률을 추출하는데, CPU사용률은 여러 정보를 이용하여 계산해야 하는데 cpuUsageCalc를 통해 계산하고 리턴 값을 저장한다. 이때 DB에는 process별로 초기 정보가 한번만 저장되어야 한다. 따라서 process name의 중복 여부를 체크해줄 필요가 있는데 db의 네 칸 단위 중 세 번째 칸을 탐색하고 같은 process name이 존재하면 그대로 함수를 종료한다. (line 196~204)

```
176  int storeInfo(vector<string> proc_info)
177  {
196    /*Check if same process name already exists in DB*/
197    if(db_cnt/4 >= 1)
198    {
199      for(i=3; i < db_cnt; i += 4)
200      {
201        if(name.compare(db[i]) == 0)
202          return 1;
203      }
204    }
```

그렇지 않다면 필요 프로세스 정보를 db에 저장하고 저장 내용을 출력한다. 그리고 해당 프로세스가 벤치마크라면 첫 실행이므로 메시지를 출력하고 bm_1/2_stat을 1로 바꿔준다.

```
212     /*Save information in DB*/
213     db.push_back(dbnum);
214     db.push_back(pid);
215     db.push_back(name);
216     db.push_back(dbusage);
217
218     db_cnt += 4;
219
220     printf("#%d    %s    %s    %.2f%%\n", (db_cnt/4), cpid, cname, cpu_usage);
221
222     if(!(name.compare("(benchmark1)")) && bm_1_stat == 0)
223     {
224       bm_1_stat = 1;
225       printf("**Benchmark1 : First Execution\n\n");
226     }
227     if(!(name.compare("(benchmark2)")) && bm_2_stat == 0)
228     {
229       bm_2_stat = 1;
230       printf("**Benchmark2 : First Execution\n\n");
231     }
```

f. double cpuUsageCalc(vector<string>)

stat의 내용 중 필요 정보를 이용해 CPU 사용률을 계산하는 함수이다. User mode에서 스케쥴 되는 프로세스 시간인 utime, kernel mode에서 스케쥴되는 프로세스 사간인 ctime등으로 프로세스에 사용된 총 시간을 계산하고 이를 hertz (시스템의 초당 clock수)를 이용해 CPU사용률을 퍼센트로 값으로 계산라여 리턴 한다.

```
257     /*Number of clock ticks per second of my system*/
258     hertz = (double)sysconf(_SC_CLK_TCK);
278     /*Calculate total time spent for the process*/
279     total_time = utime + stime;
280     total_time = total_time + cutime + cstime;
281     //include the time from children processes
282
283     seconds =  uptime - ((double)(starttime) / hertz);
284     cpu_usage = 100 * ((total_time / hertz) / seconds);
```

g. int checkBenchmark1(2)(vector<string>)

해당 프로세스가 벤치마크인지 process name을 이용해 검사한다. 맞다면 1을 아니라면 0을 리턴한다.

h. void callSyscall(char *)

벤치마크가 첫 번째 실행을 마치고 두 번째 실행을 시작하였을 때 (bm_1/2_stat이 3일 때) 호출되는 함수로 프로세스 이름을 통해 db를 탐색하여 cpu usage값을 찾고 CPU bound인지 I/O bound인지 출력한다. 이후에 시스템 콜을 호출하는데 원래는 db의 PID을 이용하려 했지만 두 번째 실행 때는 PID가 새로 부여되기 때문에 초기 정보를 저장하는 db의 정보와는 PID가 일치하지 않는다. 따라서 전역 번수로 따로

선언한 각 벤치마크의 pid를 이용하여 시스템 콜을 호출하도록 하였다.

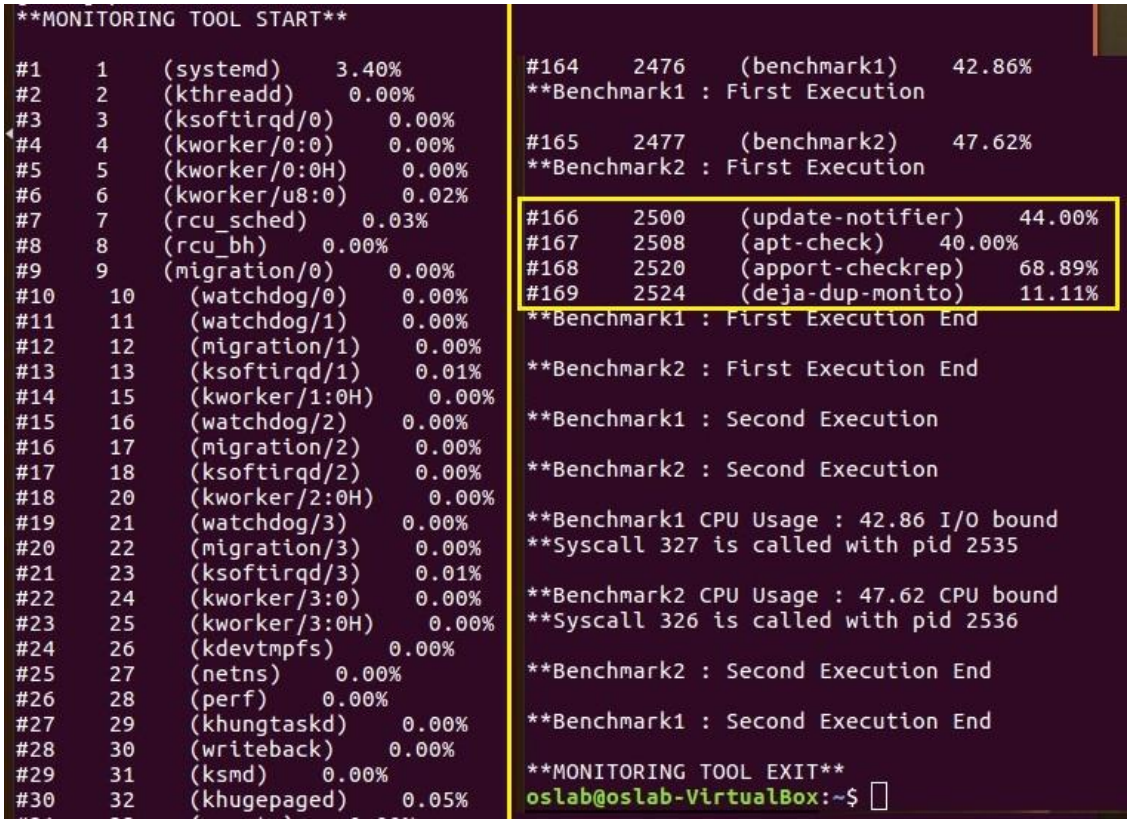
```
316  void callSyscall(char* name)
317  {
328    for(i=3; i < db_cnt; i += 4)
329    {
330      if(strname.compare(db[i]) == 0) //same name exist
331      {
332        break;
333      }
334    }
335    cpid = (char *)db[i-1].c_str();
336    ccpu_usage = (char *)db[i+1].c_str();
```

CPU bound인 Benchmark2는 vruntime을 조금씩 증가시켜 우선적으로 완료하게 하는 시스템콜 326 (sched_faster)을 호출하고 I/O bound인 Benchmark1은 반대로 vruntime을 빠르게 증가시키는 시스템콜 327 (sched_slower)을 호출한다.

```
341    if(strname.compare("(benchmark1)"))
342      bm_num = 2;
343    else
344      bm_num = 1;
345
346    if(bm_num == 2)
347    {
348      printf("**Benchmark2 CPU Usage : %.2f CPU bound\n", cpu_usage);
349      if(bm_2_pid != -1){
350        syscall(FASTER, bm_2_pid);
351        printf("**Syscall 326 is called with pid %d\n\n", bm_2_pid);
352      }
353      bm_2_stat = 4;
354    }
355    if(bm_num == 1)
356    {
357      printf("**Benchmark1 CPU Usage : %.2f I/O bound\n", cpu_usage);
358      if(bm_1_pid != -1){
359        syscall(SLOWER, bm_1_pid);
360        printf("**Syscall 327 is called with pid %d\n\n", bm_1_pid);
361      }
362      bm_1_stat = 4;
363    }
364  }
```

E. 동작 출력 결과

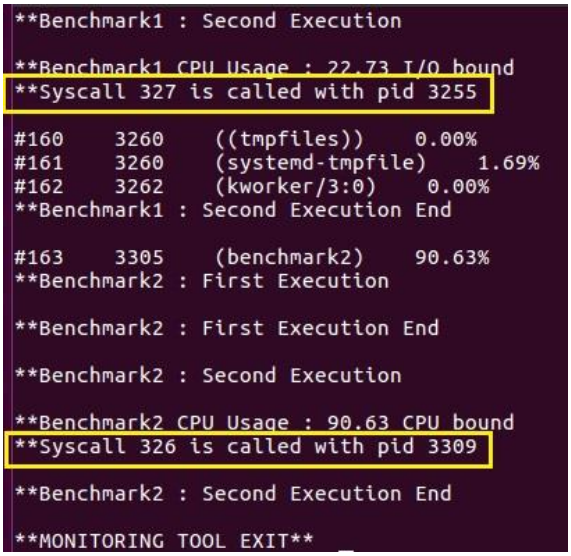
실행되어 DB에 저장된 프로세스 정보들 index, PID, process name, CPU usage(%)을 순서대로 출력하고 각 Benchmark가 실행될 때는 실행 상태에 따른 메시지를 출력한다. 표시한 부분과 같이 성능 측정 과정 중에도 새로운 프로세스가 실행되면 그 저장 내용을 출력한다. 두 Benchmark가 각각 두번의 실행을 마치면 모니터링 툴은 자동으로 종료된다.

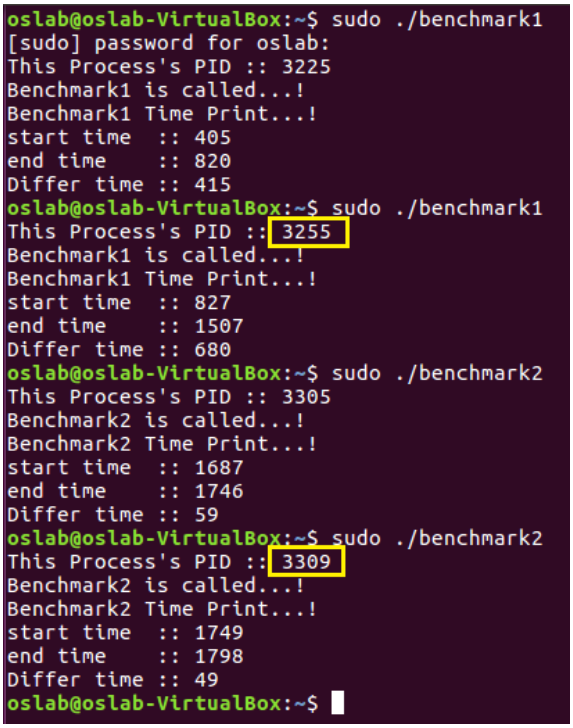


5. Benchmark 성능 측정 결과

A. 성능 측정 결과 변화

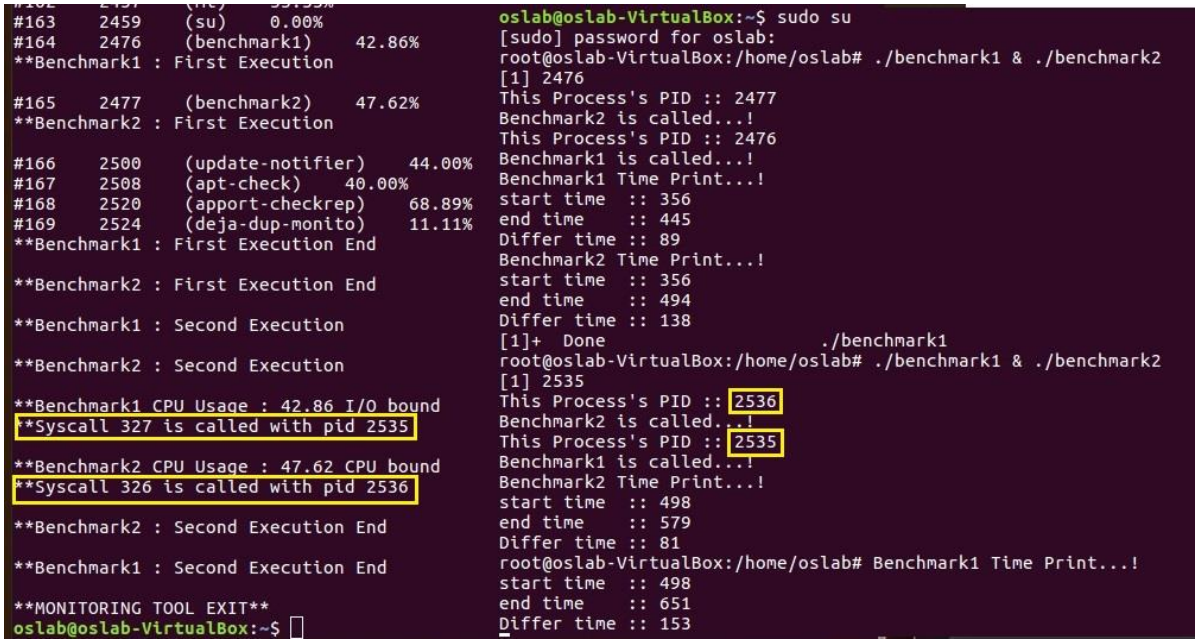
a. Benchmark1, Benchmark2를 각각 실행할 경우





Benchmark1은 415 -> 680로 수행시간이 늘어났고 Benchmark2는 59 -> 49로 수행 시간이 줄어든 모습이다.

b. Benchmark1, Benchmark2를 동시에 실행할 경우



Benchmark1은 89 -> 153으로 수행시간이 늘어났고 Benchmark2는 138 -> 81로 수행시간이 줄어든 모습이다.

B. 결과 분석

Benchmark1의 두 번째 실행에는 sched_slower가 호출되어 첫 번째 실행에 비해 실행 시간이 늘어났고 Benchmark2의 두 번째 실행에는 sched_faster가 호출되어 첫 번째 실행에 비애 실행 시간이 줄어들었다.

6. 과제 진행 중 겪은 어려움

커널 컴파일 후 install하는 과정에서 알 수 없는 동일한 에러 메시지가 계속 발생하면서 진행이 되지 않았다. 커널 코드를 수정 전으로 되돌려봐도 그대로 에러가 발생했고 결국 Ubuntu 파일을 모두 지운 후 0차과제의 9번째 단계부터 다시 진행했다. (다른 전공 강의에서 동일한 Virtual Box 위에 과제를 진행중이어서 이는 그대로 유지한 상태였다.) 우분투를 새로 설치하고 커널 소스도 다시 다운받아 컴파일했다. 정말 처음부터 새로 시작하는 거였는데도 install단계에서 동일한 에러가 발생했다. 정 안되면 Virtual Box까지 지우고 다시 해보기로 하고 마지막으로 인터넷에 검색해보았더니 postinst.d의 vboxadd파일을 제거하라고 나왔다. 지워도 되는 파일인지는 모르겠지만 어차피 과제 진행이 안되는 상황이었으므로 제거했고 인스톨은 성공했다. 찾아보니 우분투 커널 패키지가 깨져서 그렇다는데 우분투 설치부터 처음부터 다시 한 상황에서 문제가 되는 부분이 왜 남아있었는지는 아직 알아내지 못했다.