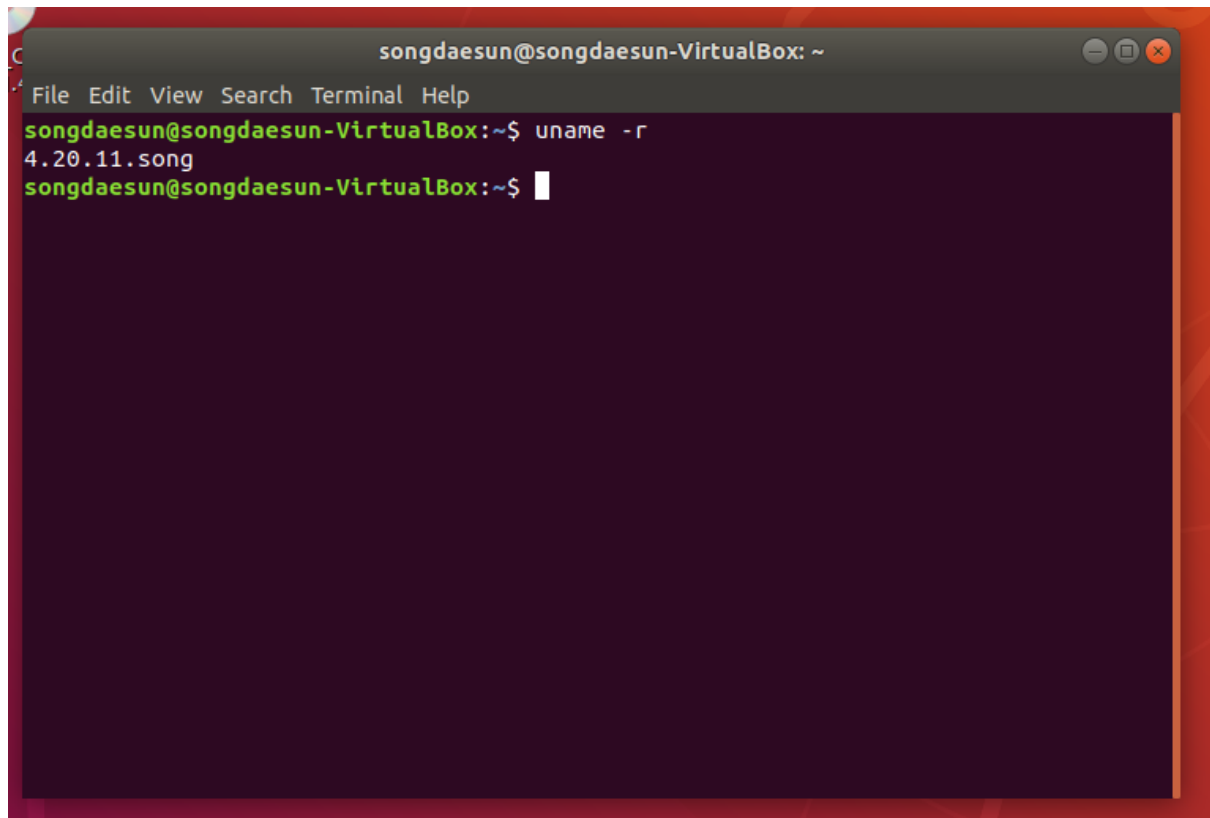


# 0차과제

## 2018320161\_컴퓨터학과\_송대선

### 1. uname -r 결과를 캡처

A screenshot of a terminal window titled 'songdaesun@songdaesun-VirtualBox: ~'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt is 'songdaesun@songdaesun-VirtualBox:~\$'. The command 'uname -r' has been entered, and the output '4.20.11.song' is displayed on the next line. The prompt is now 'songdaesun@songdaesun-VirtualBox:~\$' with a cursor.

```
songdaesun@songdaesun-VirtualBox: ~  
File Edit View Search Terminal Help  
songdaesun@songdaesun-VirtualBox:~$ uname -r  
4.20.11.song  
songdaesun@songdaesun-VirtualBox:~$
```

### 2. 과제 수행중 발생한 문제점과 해결방법

크게 어려운 점 없이 무사히 설치하였습니다. 감사합니다.

# os1\_2018320161\_송대선\_보고서

학과 : 컴퓨터학과

학번 : 2018320161

이름 : 송대선

제출 날짜 : 2020-04-30

Freeday 사용 일수 : 0일

## 개발환경

# 개발환경

노트북 기종 : XPS 15 9550

CPU : intel i7 inside

RAM : 32GB

OS : Window 10 home 위의 Oracle VM Virtual Box 6.1.6의 Ubuntu 18.04.2

## 리눅스의 시스템 콜 (호출 루틴 포함)에 대한 설명

시스템 콜에 대한 기본적인 이해는 다음과 같다. 운영체제는 kernel mode와 user mode로 나뉘어 구동되는데, user mode에서 구동되는 user application은 hardware에 직접적으로 접근할 수 없다. Hardware에 직접 접근 할 수 있는 것은 kernel mode의 kernel이다. user application이 하드웨어에 간접적으로 접근하는 그 통로가 system call인 것이다. system call interface를 통해 커널 영역의 기능을 user mode가 사용하는 것이 가능해진다. 즉, system call은 응용프로그램에서 운영체제에게 기능(시스템 자원)을 수행해달라고 하는 하나의 수단인 것이다.

사용자 프로세서가 system call을 요청하면 제어가 커널로 넘어간다. 즉, user mode에서 kernel mode로 전환된다는 것이다. 커널은 내부적으로 각각의 시스템콜을 구분하기 위해 기능별로 고유번호를 할당해 놓는다. 해당번호는 커널내부에 제어루틴을 정의한다. 커널은 요청받은 system call에 대응하는 기능번호를 확인한다. 커널은 그 번호에 맞는 서비스 루틴을 호출하게 된다. 서비스 루틴을 모두 처리하고나면 kernel mode에서 user mode로 다시 넘어온다.

## 수정 및 작성한 부분과 설명 (이유)

### syscall\_64.tbl

syscall\_64.tbl 파일에 다음의 코드를 추가하였습니다.

```
335 common  oslab_push  __x64_sys_oslab_push
336 common  oslab_pop   __x64_sys_oslab_pop
```

그 이유는 저의 system call 함수인 oslab\_push와 oslab\_pop에게 고유번호를 부여하고 정의해주기 위함입니다.

## syscalls.h

syscalls.h 파일에 다음의 코드를 추가하였습니다.

```
asm linkage int sys_oslab_push(int);
asm linkage int sys_oslab_pop(void);
```

asm linkage를 함수 앞에 선언함으로써 assembly code에서도 C함수 호출을 가능하게 했습니다.

sys\_oslab\_push, sys\_oslab\_pop의 prototype을 정의했습니다

## my\_stack\_syscall.c

my\_stack\_syscall.c 파일은 직접 작성했고, 다음과 같은 흐름으로 진행됩니다.

**oslab\_push의 세부적인 구현은 다음과 같습니다.**

1. 일단 입력받은 int값이 이미 stack에 존재하는지의 여부를 판단하고, 이미 존재한다면 stack에 추가하지 않습니다. 만일 stack에 존재하지 않는다면 step 2로 넘어갑니다.
2. 지금의 stack이 full인지의 여부를 판단합니다. 이는 int variable top과 MAXSIZE의 크기비교로 정합니다. 만일 stack이 full이 아니라면, step 3로 넘어갑니다.
3. 입력받은 int 값을 stack에 추가하고 top의 값을 1 증가시킵니다.

**oslab\_pop의 세부적인 구현은 다음과 같습니다.**

1. stack이 empty인지의 여부를 판단합니다. top과 int -1의 비교를 통해 이를 정합니다. 만일 stack이 empty라면 -1을 return합니다. 만일 stack이 empty가 아니라면 step 2로 넘어갑니다.
2. stack에서 top에 해당하는 값을 저장해두었다가 top을 1만큼 감소시키고 저장해 두었던 값을 return합니다.

## Makefile

Makefile의 obj-y부분에 my\_stack\_syscall.o를 추가하였습니다.

```
...
async.o range.o smpboot.o ucount.o my_stack_syscall.o
```

kernel make에서 제가 작성한 my\_stack\_syscall.c이 object로 같이 커널로 컴파일되게 하려 합니다.

## app\_oslab.c

User Application인 app\_oslab.c을 작성했고, 다음과 같은 흐름으로 진행됩니다.

1. 1, 2, 3을 순서대로 oslab\_push합니다.
2. 3번을 연속으로 oslab\_pop합니다.
3. 1, 2, 2을 순서대로 oslab\_push합니다.

## 실행 결과 스냅샷

```
song@song-VirtualBox:~$ sudo ./app_oslab
push: 1
push: 2
push: 3
pop: 3
pop: 2
pop: 1
push: 1
push: 2
push: 2
```

user application 실행결과

```

99.304725] [System Call] oslab_push():
99.304726] push 1
99.304727] Stack Top -----
99.304728] 1
99.304728] Stack Bottom -----
99.304787] [System Call] oslab_push():
99.304806] push 2
99.304806] Stack Top -----
99.304807] 2
99.304807] 1
99.304808] Stack Bottom -----
99.304810] [System Call] oslab_push():
99.304810] push 3
99.304811] Stack Top -----
99.304811] 3
99.304811] 2
99.304812] 1
99.304812] Stack Bottom -----
99.304814] [System Call] oslab_pop():
99.304814] pop 3
99.304815] Stack Top -----
99.304815] 2
99.304816] 1
99.304816] Stack Bottom -----
99.304818] [System Call] oslab_pop():
99.304818] pop 2
99.304819] Stack Top -----
99.304819] 1
99.304819] Stack Bottom -----
99.304821] [System Call] oslab_pop():
99.304821] pop 1
99.304822] Stack Top -----
99.304822] Stack Bottom -----
99.304824] [System Call] oslab_push():
99.304824] push 1
99.304825] Stack Top -----
99.304825] 1
99.304825] Stack Bottom -----
99.304827] [System Call] oslab_push():
99.304828] push 2
99.304828] Stack Top -----
99.304828] 2
99.304842] 1
99.304843] Stack Bottom -----
99.304845] [System Call] oslab_push():
99.304845] Push 2
99.304846] element overlap
99.304846] Stack Top -----
99.304846] 2
99.304847] 1
99.304847] Stack Bottom -----
song@song-VirtualBox:~$

```

sudo dmesg 결과

## 숙제 수행 과정 중 발생한 문제점과 해결방법

커널 컴파일이 전혀 되지 않았던 문제가 있었는데, 이는 그 컴퓨터 시스템의 CPU가 AMD 기반이었기에 생긴 문제로 추정됩니다. INTEL기반의 CPU를 사용하는 노트북에서 같은 작업을 실행하였더니 정상적으로 작동하였습니다.

# 1차과제 출력본

app\_oslab.c

```
#include <unistd.h>
#include <stdio.h>

#define my_stack_push 335
#define my_stack_pop 336

int main(){
    int a;
    int index;

    for(index = 1; index <=3; index++){
        a = syscall(my_stack_push, index);
        printf("push: ");
        printf("%d\n", a);
    }

    for(index=3; index>=1; index--){
        a = syscall(my_stack_pop, index);
        printf("pop: ");
        printf("%d\n", a);
    }

    a = syscall(my_stack_push, 1);
    printf("push: ");
    printf("%d\n", a);

    a = syscall(my_stack_push, 2);
    printf("push: ");
    printf("%d\n", a);

    a = syscall(my_stack_push, 2);
    printf("push: ");
    printf("%d\n", a);

    return 0;
}
```

my\_stack\_syscall.c

```
#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <linux/linkage.h>

#define MAXSIZE 500

int stack[MAXSIZE];
int top = -1;

int is_already_stacked; //the indictor variable which show if there is the same element as input 'a' in the stack
int pop_element; //the popped element
int stack_index; //index variable for 'for loop'

SYSCALL_DEFINE1(oslab_push, int, a){ //my push syscall function

    printk(KERN_INFO "[System Call] oslab_push(): ");
    is_already_stacked =0;

    for (stack_index = 0; stack_index <= top; stack_index++){ //check if there is the same element as input 'a' in the stack
        if(a == stack[stack_index]){
            is_already_stacked = 1;
            break;
        }
    }

    if (is_already_stacked == 0){ //if there is no same elemet, then check whether stack is full
        if (top<MAXSIZE){
            //stack is not full, then push element 'a'.

            top++;
        }
    }
}
```

```

    stack[top] = a;
    printk("push %d\n", a);

    //print stack
    printk("Stack Top -----\\n");
    for (stack_index=top; stack_index>-1;stack_index--){
        printk("%d\\n",stack[stack_index]);
    }

    printk("Stack Bottom -----\\n");
    return a;
}
else{
    //stack is full
    printk("Push %d", a);

    printk("Stack is full\\n");

    //print stack
    printk("Stack Top -----\\n");
    for (stack_index=top; stack_index>-1;stack_index--){
        printk("%d\\n",stack[stack_index]);
    }

    printk("Stack Bottom -----\\n");

    return a;
}
}
else{
    //element 'a' is in the stack
    printk("Push %d", a);

    printk("element overlap\\n");

    //print stack
    printk("Stack Top -----\\n");
    for (stack_index=top; stack_index>-1;stack_index--){
        printk("%d\\n",stack[stack_index]);
    }

    printk("Stack Bottom -----\\n");

    return a;
}
}

SYSCALL_DEFINE0(oslab_pop){ //my pop syscall function
    printk(KERN_INFO "[System Call] oslab_pop(): ");

    if (top>-1){ //check if the stack is empty

        //pop element
        pop_element = stack[top];
        top--;

        printk("pop %d\\n", pop_element);

        //print stack
        printk("Stack Top -----\\n");
        for (stack_index=top; stack_index>-1; stack_index--){
            printk("%d\\n",stack[stack_index]);
        }

        printk("Stack Bottom -----\\n");

        return pop_element;
    }
    else{
        // if stack is empty, then return -1
        printk("stack is empty");

        //print stack
        printk("Stack Top -----\\n");
        for (stack_index=top; stack_index>-1;stack_index--){
            printk("%d\\n",stack[stack_index]);
        }

        printk("Stack Bottom -----\\n");

        return -1;
    }
}

```



```
}  
}
```

result.txt

```
song@song-VirtualBox:~$ sudo ./app_oslab  
push: 1  
push: 2  
push: 3  
pop: 3  
pop: 2  
pop: 1  
push: 1  
push: 2  
push: 2  
song@song-VirtualBox:~$  
  
song@song-VirtualBox:~$ sudo dmesg  
...(중략)...  
[ 99.304725] [System Call] oslab_push():  
[ 99.304726] push 1  
[ 99.304727] Stack Top -----  
[ 99.304728] 1  
[ 99.304728] Stack Bottom -----  
[ 99.304787] [System Call] oslab_push():  
[ 99.304806] push 2  
[ 99.304806] Stack Top -----  
[ 99.304807] 2  
[ 99.304807] 1  
[ 99.304808] Stack Bottom -----  
[ 99.304810] [System Call] oslab_push():  
[ 99.304810] push 3  
[ 99.304811] Stack Top -----  
[ 99.304811] 3  
[ 99.304811] 2  
[ 99.304812] 1  
[ 99.304812] Stack Bottom -----  
[ 99.304814] [System Call] oslab_pop():  
[ 99.304814] pop 3  
[ 99.304815] Stack Top -----  
[ 99.304815] 2  
[ 99.304816] 1  
[ 99.304816] Stack Bottom -----  
[ 99.304818] [System Call] oslab_pop():  
[ 99.304818] pop 2  
[ 99.304819] Stack Top -----  
[ 99.304819] 1  
[ 99.304819] Stack Bottom -----  
[ 99.304821] [System Call] oslab_pop():  
[ 99.304821] pop 1  
[ 99.304822] Stack Top -----  
[ 99.304822] Stack Bottom -----  
[ 99.304824] [System Call] oslab_push():  
[ 99.304824] push 1  
[ 99.304825] Stack Top -----  
[ 99.304825] 1  
[ 99.304825] Stack Bottom -----  
[ 99.304827] [System Call] oslab_push():  
[ 99.304828] push 2  
[ 99.304828] Stack Top -----  
[ 99.304828] 2  
[ 99.304842] 1  
[ 99.304843] Stack Bottom -----  
[ 99.304845] [System Call] oslab_push():  
[ 99.304845] Push 2  
[ 99.304846] element overlap  
[ 99.304846] Stack Top -----  
[ 99.304846] 2  
[ 99.304847] 1  
[ 99.304847] Stack Bottom -----  
song@song-VirtualBox:~$
```

# **운영체제 2차 과제**

**학과 : 컴퓨터학과**

**학번 : 2018320161**

**이름 : 송대선**

**제출 날짜 : 2020-06-09**

**Freeday 사용 일수 : 0일**

# 과제 개요 및 프로세스와 스케줄러의 개념

과제의 개요는 다음과 같다.

: 이번 과제는 프로세스와 스케줄러의 개념을 이해하기 위해 진행되었고, 과제의 목적은 CPU burst의 시간을 실제로 측정해보고 그것을 통계적으로 분석하는 것이다.

이를 위해서 스케줄러 코드를 직접 수정할 필요성이 있다. kernel/sched/stats.h의 sched\_info\_depart 함수는 프로세스가 CPU 점유를 마쳤을 때 호출되는 함수이다. 따라서 이 함수로부터 프로세스의 CPU burst를 측정하는 것이 합리적이다.

sched\_info\_depart 함수에는 CPU burst를 의미하는 delta가 이미 구현되어 있고, PID는  $t \rightarrow pid$ 에, 그리고 프로세스의 호출 횟수는  $t \rightarrow sched\_info.pcount$ 에 구현되어 있다. 따라서 이 변수들을 이용하여 SAMPLE\_TIMES 1000을 기준으로, 즉 프로세스 당 1000번에 한번씩 CPU burst 값을 추출한다.

커널을 컴파일하고, 적용한 뒤에 크롬으로 유튜브 영상 10개, firefox tab 2개, Thunderbird Mail 창 2개, Files창 2개, Rhythmbox의 Radio → All 9 genres → HBR1.com-Dream Factory 라디오 1개, LibreOffice Writer 창 2개, Ubuntu Software 창 1개, Help 창 1개, 대기중인 Terminal 2개를 띄웠다. 그후 약 30분간 CPU burst 측정 실험을 진행하였다.

dmesg > log.txt 명령어로 로그를 출력한 뒤 엑셀로 파싱한 뒤에 그래프를 그렸다.

프로세스의 개념은 다음과 같다.

: 프로세스는 디스크에 저장되어 있는 프로그램 파일을 실행시켰을 때, 그 프로그램을 메모리로 옮겨놓은 것이다. 프로세스는 전원이 끊기면 사라지지만, 프로그램은 전원을 꺼도 유지된다.

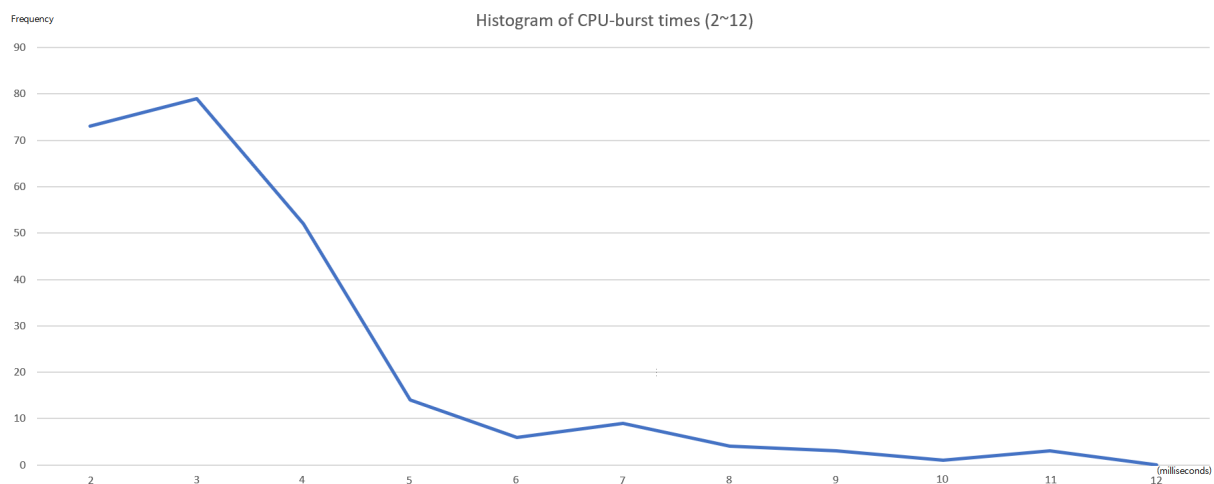
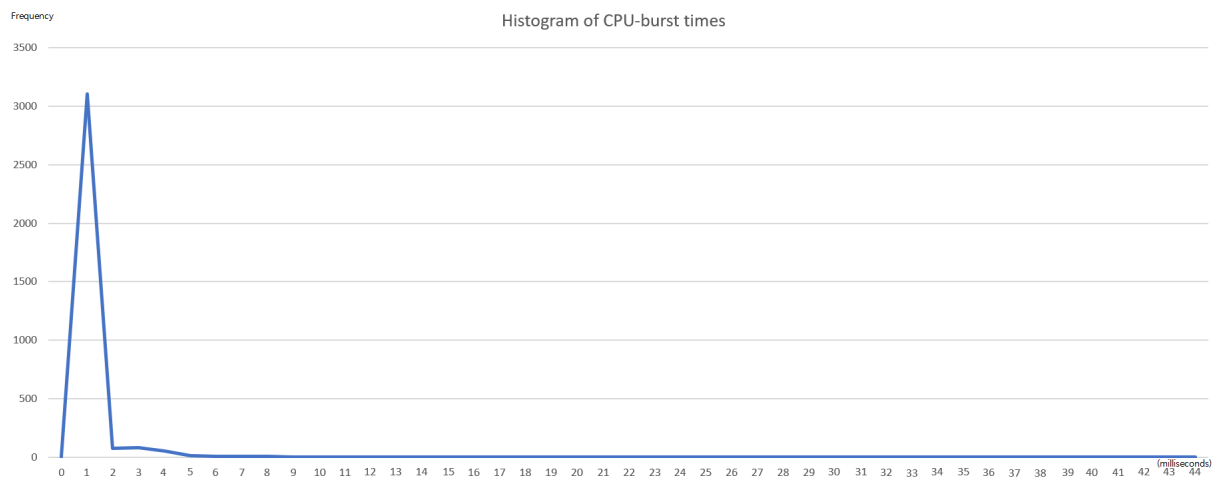
프로세스의 abstraction은 Execution unit과 Protection domain이다. Execution unit은 스케줄링의 단위라는 뜻이다. 이는 스케줄링이 process단위로 스케줄링이 된다는 의미이다. Protection domain이라는 말은 한 프로세스가 다른 프로세스의 메모리 영역을 침범할 수 없다(그렇게 되어서는 안된다)는 의미이다. 이 원칙을 깨는 것이 해킹의 기본적인 아이디어이다.

프로세스의 implementation은 다음과 같다. Program counter, Stack, Data section, 이 세가지 CPU의 레지스터를 구현하는 것이 프로세스의 implementation이다. Program counter은 프로세스의 코드 어느 부분을 수행하고 있는지를 계산하는 레지스터이다. Stack은 stack pointer로, stack 부분에서 내가 어디쯤 메모리를 사용하고 있는지를 알려주는 레지스터이다. Data section은 내 global data의 위치를 알려주는 레지스터이다.

스케줄러의 개념은 다음과 같다.

: 스케줄링이란 프로세스들을 실행을 시켜야 하는데, 어떤 순서로 프로세스와 CPU를 맵핑시키는지를 다루는 것이다. 스케줄링은 매우 중요한 영역이며, 스케줄링에 따라서 전체 시스템의 성능이 좌우된다. CPU 스케줄링의 목표는 당연히게도 CPU를 최대로 활용하는 것이다. 이 이야기는 다시말하자면 idle을 최소화하는 것과 같은 내용이다.

## CPU burst에 대한 그래프 및 결과분석



분석결과 대부분의 CPU burst(3106개)가 1millisecond를 넘지 않는 것으로 나왔다. 좀 더 자세히 알아보기 위해 2~12milliseconds구간을 확대하여 보았다. frequency는 급격히 감소하며 11~12 milliseconds 구간은 frequency가 0이었다. 이는 수업시간의 제시된 그래프와 형태는 유사하다. 그러나 처음의 피크점에서 감소하는 속도가 훨씬 더 가파르다. 이 이유는 더욱 성능이 좋은 CPU를 사용했기 때문이라 생각된다.

## 작성한 모든 소스코드에 대한 설명

필자는 kernel/sched/stats.h의 sched\_info\_depart 함수만을 수정하였다. 원래 구현되었던 주어진 프로세스의 state가 running할 때의 조건문을 이용하였다. 그리고  $t \rightarrow \text{sched\_info.pcount}$ 를 이용하여, 그 프로세스가 SAMPLE\_TIMES으로 나누어지는 수만큼 호출되었는지의 조건문을 만들 수 있었다. 만일 위 조건문 두개를 모두 만족한다면, 그 프로세스의 pid와 이미 구현되어있는 변수 delta를 이용해 CPU burst를 출력한다.

## 과제 수행 시의 문제점과 해결 과정 또는 의문 사항

처음에는 포인터 리스트나 구조체를 따로 만들어서 일일이 프로세스당 CPU 점유 횟수를 기록하게 만들었습니다. 하지만 알 수 없는 이유로 자꾸만 오류를 일으켰고, 프로세스당 CPU 점유 횟수를 의미하는 변수가 이미 있지 않을까하는 바램으로 코드를 다시 찾아보았고, pcount변수를 발견하게 되어서 그것을 이용해 구현하게 되었습니다.

# 2차과제\_code

```
#define SAMPLE_TIMES 1000

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->sched_info.pcount % SAMPLE_TIMES == 0){
            printk("[pid: %d], CPUBurst: %llu", t-> pid, delta);
        }
    }
}
```