2021 년 1 학기 운영체제 2 차 과제

지도 교수님	유혁 (hxy@os.korea.ac.kr)
주제	프로세스 및 리눅스 스케줄링의 이해
학과	**컴퓨터학과**
학번	**2017320123**
이름	**김장영 (dkel03@korea.ac.kr)**
제출일	**2021. 05. 20. (목)**
FreeDay 사용일 수	**0 일**
환 경	- Oracle VM VirtualBox (Host: Window) - VM: 4core process, 4096MB memory, KVM Para-Virtualization - 리눅스 Ubuntu 18.04 LTS – 64bit - Kernel Version 4.20.11
목 차	

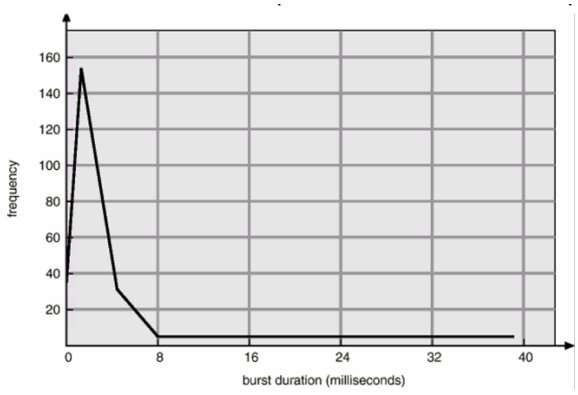
I. 과제 개요

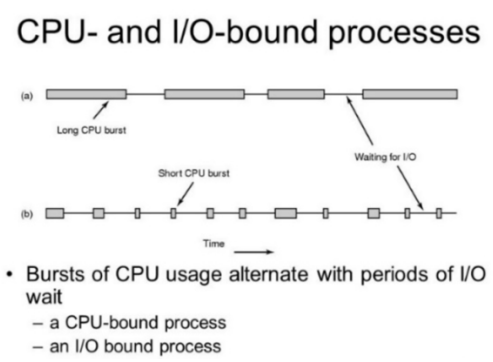
과제는 리눅스의 프로세스 및 프로세스 스케줄러를 이해하기 위해, 프로세스의 CPU 점유 시간(burst)과 vruntime 을 살펴본다. 리눅스 프로세스 스케줄러는 각 프로세스에 CPU 를 점유하는 프로세스 점유 시간을 할당하는데, 점유 시간의 할당 방식은 프로세스의 종류, 컴퓨터의 종류, 스케줄러의 종류에 따라 그 특성이 다르게 나타난다. 또한 vruntime 값에 가중치를 부여하여 프로세스의 우선순위에 따라 CPU 를 점유할 수 있도록 한다. 이에 과제에서는 실제 리눅스 운영체제에서 스케줄러가 프로세스에 어떻게 CPU 점유 시간을 할당하는지, 그리고 현대의 프로세스와 시스템에서 프로세스의 CPU 점유 시간이 어떠한 분포를 나타내는지 실제로 조사해본다.

II. 프로세스와 리눅스 CFS 스케줄러의 개념 설명

II.1. 프로세스

CPU 의 I/O, Burst Cycle 은 CPU 로 연산을 수행하는 시간인 CPU Burst 와 I/O 처리를 위해 기다리는 시간인 I/O Burst 로 구성되며, 일반적인 프로세스는 두 Burst 를 번갈아 가며 수행된다. 이때, 긴 CPU Burst 를 가지는 프로세스를 CPU-bound 프로세스 (오른쪽 그림의 a), 짧은 CPU Burst 를 가지는 프로세스를 I/O bound (오른쪽 그림의 b)라고 한다. 서로 다른 프로세스, 시스템임에도 불구하고, CPU-burst time 은 대체적으로 아래의 왼쪽 사진과 같은 분포를 보이게 된다. 이는 일반적인 프로세스들이 대부분 I/O bound 프로세스임을 뜻한다고도 볼 수 있다. (단, x 축인 burst time 의 duration 때문에 언제 자료인지를 파악할 필요가 있다)





II.2. CFS 스케줄러

CFS(Completely Fair Scheduler)는 리눅스의 기본 스케줄러이다. CFS 이란 용어를 그대로 풀면 '완벽하게 공정한 스케줄러'라고 해석할 수 있다. 즉, 런큐에서 실행 대기 상태로 기다리는 프로세스를 공정하게 실행하도록 기회를 부여하는 스케줄러이다.

만약 A, B 두 개의 태스크가 진행되고 있다면 A 와 B 의 CPU 사용시간은 항상 1:1 로 같아야 한다. 그러나 두 태스크가 번갈아 가며 수행되므로 임의의 시점에 두 태스크의 CPU 사용 시간이 항상 1:1 로 같을 수 없다. 따라서 CFS 는 정해진 '시간 단위'로 봤을 때 시스템에 존재하는 태스크들에게 공평한 CPU 시간을 할당하는 것을 목표로 한다. 만약 1 초를 '시간 단위'로 한다면

0.5 초 동안 A 태스크를 수행시키고, 그런 뒤 0.5 초간 B 태스크를 수행시킴으로써 1 초가 지난 이후 A 와 B 의 CPU 사용시간이 1:1 이 되도록 하는 것이다.

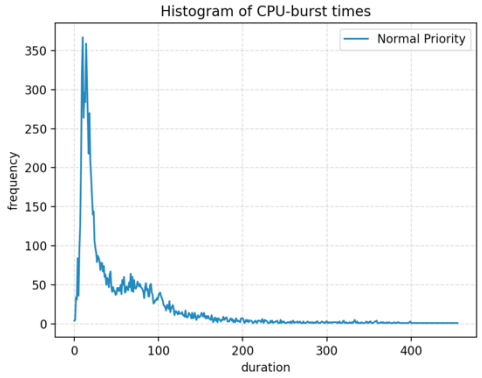
CFS 의 기본 개념은 작업에 프로세서 시간을 제공할 때 밸런스(공평성)를 유지하는 것이다. 즉 프로세스에 공평한 양의 프로세서(=CPU)가 제공되어야 한다. 작업 시간의 밸런스가 무너진 경우에는(다른 작업에 비해 하나 이상의 작업에 공평한 양의 시간이 주어지지 않은 경우) 작업 시간이 적게 지정된 작업에 실행 시간이 주어져야 한다. CFS 에서는 밸런스를 결정하기 위해 vruntime(virtual runtime; 가상 런타임)이라는 지정된 작업에 제공된 시간의 양을 관리한다. 작업의 vruntime 이 작을수록 즉, 프로세서에 액세스할 수 있도록 허용된 시간이 작은 작업일수록 더 많은 프로세서 시간이 필요하다. vruntime 은 아래의 식으로 계산될 수 있다.

A 의 vruntime 증가량 = (실행주기) * (A 의 priority) / (A's priority + B's priority) / (A 의 priority)
= **(실행주기)/ (A's priority + B's priority)**

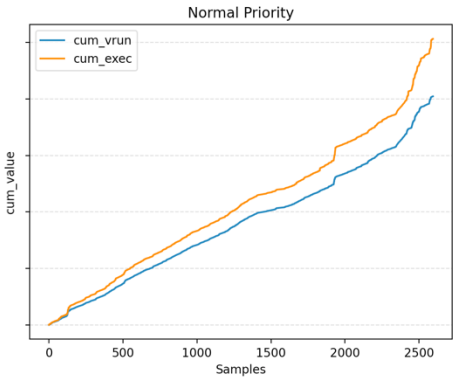
B 의 vruntime 증가량 = (실행주기) * (B 의 priority) / (A's priority + B's priority) /(B 의 priority)
= **(실행주기)/ (A's priority + B's priority)**

III. CPU burst, vruntime 에 대한 그래프 및 결과분석

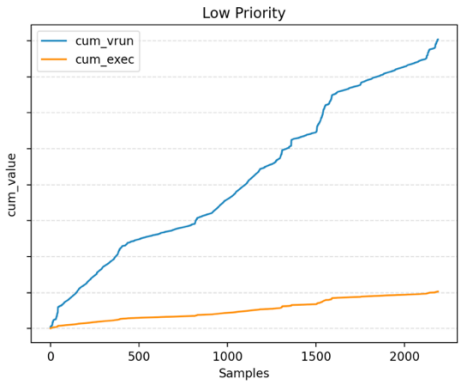
III.1. 그래프 및 관련 데이터 출력



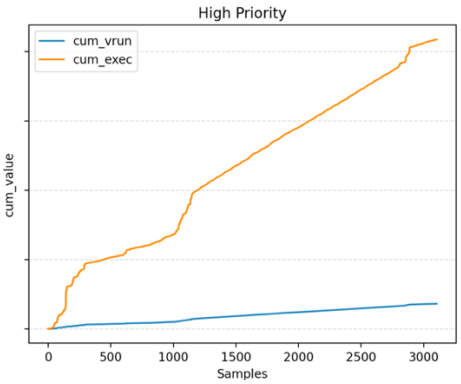
<III.1.1. Histogram of CPU-burst times>



<III.1.2. Normal Priority>



<III.1.3. Low Priority>



<III.1.4. High Priority>

```
<Normal priority, PID=2231>
총 샘플 수 : 2595
Virtual runtime과 CPU burst가 다른 샘플 수 : 238
평균 Overhead: 0.085ns
Total CPU-burst: 101240912
```

<III.1.5. Normal Priority Info>

```
<Low priority, 2729>
총 샘플 수 : 2189
Total CPU-burst: 51510509


<High priority, 2209>
총 샘플 수 : 3105
Total CPU-burst: 83480883
```

<III.1.6. Low, High Priority Info>

III.2. 구동한 프로세스에 대한 설명

기본 커널 데몬 및 쓰레드 외에 CPU 사이클을 소모하는 프로세스로서 **카카오톡**을 설치하여 실행해보았다. 리눅스에 카카오톡을 설치하는 과정은 꽤나 까다로웠는데, 이 부분은 과제 수행시의 문제점 및 해결방안에서 서술한다.

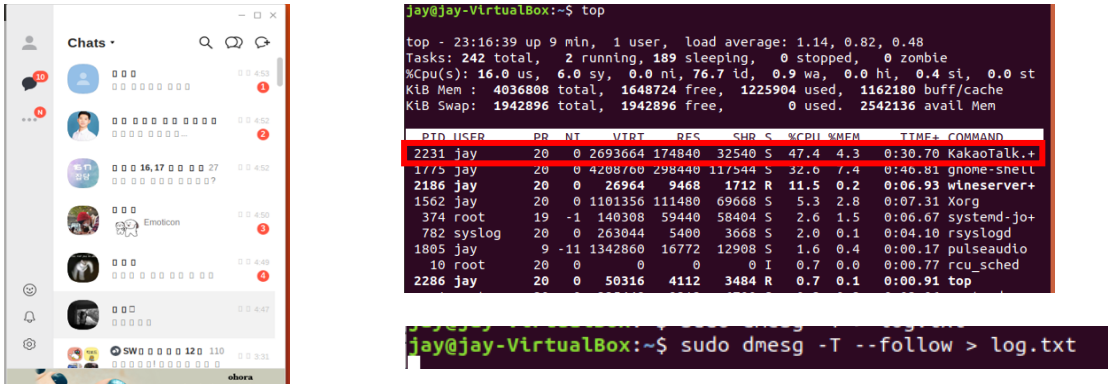
앞서 일반적인 프로세스는 두 가지의 Burst 가 번갈아 수행되며, 긴 CPU Burst 를 가지는 프로세스를 CPU-bound 프로세스, 짧은 CPU Burst 를 가지는 프로세스를 I/O bound 라고 함을 설명했다. 카카오톡의 경우 Normal Priority 에서 CPU-Burst 의 평균값을 측정해본 결과 "Histogram of CPU-burst times" 그래프의 앞쪽에 위치하며, I/O bound 프로세스임을 알 수 있었다.

카카오톡은 모바일 메신저로서 네트워크 I/O 작업이 잦고, 양방향 통신을 위해 IPC 로 소켓을 사용한다. 이에 CPU 연산이 필요한 작업이 없는 경우가 대부분이고, I/O bound 의 프로세스임을 쉽게 예측할 수 있다.

프로세스 선정에 앞서, 유저의 입장에서 "우선순위에 따른 딜레이"를 체감할 수 있는가에 대해 궁금증을 가지게 되었고, 이 물음을 쉽게 확인해볼 수 있는 메신저 프로세스를 선택하게 되었다.

III.3. CPU burst, vruntime 측정 실험 수행

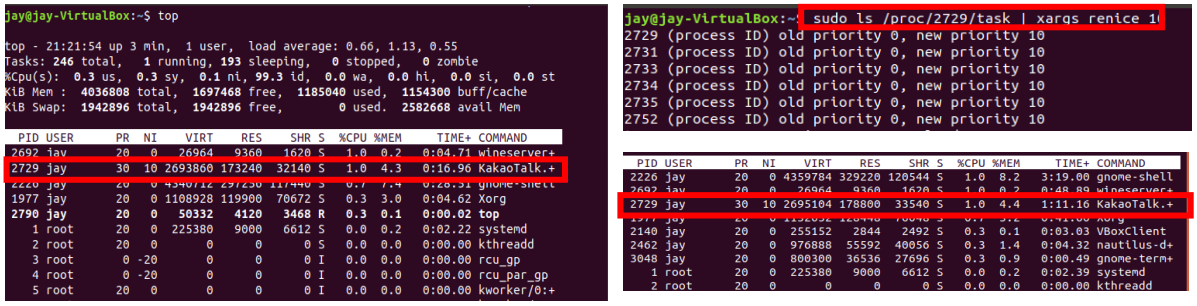
수정한 스케줄러 코드에 따라 CPU-burst, vruntime 값을 burst 값이 1,000 회 바뀔 때 마다 1 회 샘플링하여 기록하였다. 30 분 동안 전체 프로세스의 CPU burst, vruntime 을 측정하였고, **카카오톡** 프로세스에 대하여 기본적인 우선순위인 120 (NI=0), 더 낮은 우선순위인 130 (NI=10), 더 높은 우선순위인 110 (NI=-10) 총 3 번의 실험을 진행하였다.



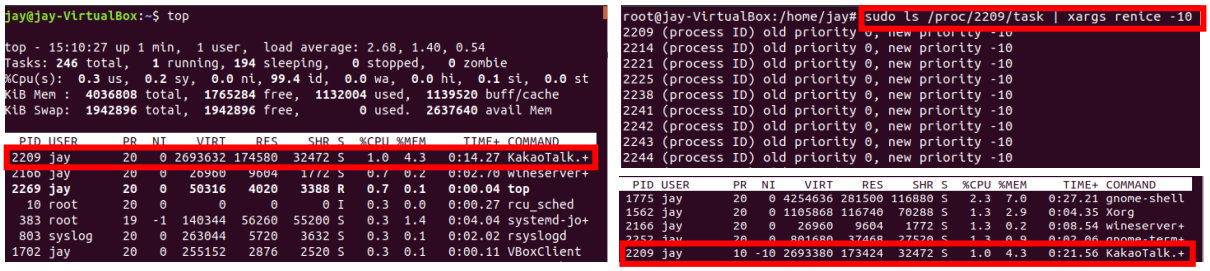
<Normal Priority = 120>

위와 같이 카카오톡을 켜두고 실시간으로 채팅을 진행하며, 30 분간 프로세스의 로그를 기록하도록 했다. 우측의 사진을 보면, 카카오톡에 해당하는 PID=2231 의 프로세스가 Priority=20 (120), Nice=0 임을 볼 수 있다. 우측 아래의 사진의 명령어는 dmesg 명령어에 -T 옵션과 --follow 옵션을 주어 log.txt 파일에 저장하도록 한 것인데, 각각 "사람이 읽을 수 있는 Time step" 옵션과 "라이브 이벤트 시청" 옵션이다. 이를 통해 30 분간의 로그를 log.txt 파일에 기록할 수 있었고, 기록하기 직전에는 dmesg 명령어에 -c 옵션을 주어 log 에 기록될 로그를 초기화 하였다.

세 가지 우선순위 옵션에 대해 동일하게 진행하였으며, log_normal, log_low, log_high 텍스트 파일에 기록하도록 하였다. 낮은 우선순위 프로세스의 PID 는 2729, 높은 우선순위 프로세스의 PID 는 2209 로 아래 사진에서 확인할 수 있으며, renice 명령어를 통해 변경해주었다.



<Low priority =130>



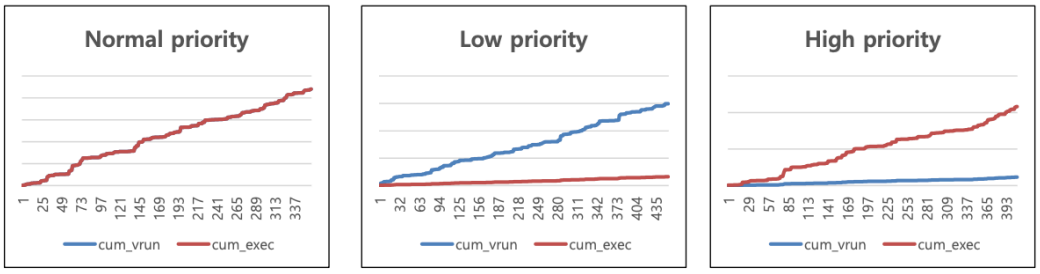
<High priority = 110>

III.4. 실험 결과 분석

III.4.1. Histogram of CPU-burst times

III.1.1 그래프의 경우 Normal-Priority 카카오톡 프로세스 실행 시의 전체 프로세스에 대해. CPU-burst time 을 측정하고, 측정값에 대한 빈도를 계산하여 히스토그램으로 표현한 것이다. 앞서 프로세스의 개념을 살펴볼 때 보았던 이론적인 히스토그램과 거의 유사한 분포를 보이는 것을 확인할 수 있었다. 이를 통해 **일반적인 유저 환경에서 대부분의 프로세스는 I/O bound Process 임**을 확인할 수 있었다.

III.4.2. 우선순위에 따른 CPU-burst & vruntime 누적 합의 변화



	Normal priority	Low priority	High priority
Total CPU-burst	88175980	64193490	108192919

CPU-burst 과 vruntime 의 누적 합에 대한 이론적인 결과는 위의 그림과 같다. 따라서 Normal priority 의 경우 두 경우가 같아야 하고, Low priority 는 vruntime 값이 높게, High priority 의 경우 CPU-burst 가 높게 측정되어야 한다. 실험 결과 그래프 III.1.3, III.1.4 를 살펴보면 **Low priority 와 High priority 에서는 그래프의 형태가 이론상의 그래프와 거의 동일**한 것을 확인할 수 있다. 또한 Total Sample 수에 비례하여 Total CPU-burst time 의 차이가 있는 것 까지도 확인하였다.

그러나 그래프 Normal Priority process 에 대한 결과인 III.1.2 를 보면, 이론상의 그래프와 다른 점이 몇 가지 발견되었다. 첫째로 **CPU-burst 와 vruntime 의 누적 합 그래프에 약간의 차이가 생기는 것**을 확인하였다. 차이를 측정해본 결과, III.1.5 와 같이 9.1%정도의 sample 에서 차이가 발생했고, 0.085ns 정도의 평균적인 차이가 있음을 확인했다. 둘째로 이론상으로는 High Priority 보다는 적게 Low Priority 보다는 많게 샘플링 되어, Total CPU-burst time 도 비례해야 하지만 High Priority 보다도 **더 큰 Total CPU-burst time 을 가지는** 것으로 확인되었다. 분명 Sample 수는 우선순위에 따라 비례적인데 누적 CPU-burst time 이 크다는 것은 이상한 부분이다. 두 가지 경우에 대해서는 의문 사항에서 다루도록 하였다.

IV. 작성한 모든 소스코드에 대한 설명

IV.1. sched.h

```
struct sched_entity {
        /* For load-balancing: */
        struct load_weight              load;
        unsigned long                   runnable_weight;
        struct rb_node                  run_node;
        struct list_head                group_node;
        unsigned int                    on_rq;

        u64                             exec_start;
        u64                             sum_exec_runtime;
        u64                             vruntime;
        u64                             delta_vruntime; //[2017320123][KimJangYoung] Add delta_(virtual)runtime variable
        u64                             prev_sum_exec_runtime;

        u64                             nr_migrations;

        struct sched_statistics         statistics;
```

첫 번째로 sched.h 파일의 sched_entity 구조체에 u64 type 의 delta_vruntime 변수를 선언해 주었다. 이 변수는 프로세스가 작업을 마치고 context switching 될 때 호출되는 sched_info_depart 에서 vruntime 값을 출력해 주기 위한 변수이다.

IV.2. fair.c

```
static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_clock_task(rq_of(cfs_rq));
        u64 delta_exec;
        u64 delta_vruntime; //[2017320123][KimJangYoung] Add delta_vruntime variable to temporary store delta_vruntime

        if (unlikely(!curr))
                return;

        delta_exec = now - curr->exec_start;
        if (unlikely((s64)delta_exec <= 0))
                return;

        curr->exec_start = now;

        schedstat_set(curr->statistics.exec_max,
                      max(delta_exec, curr->statistics.exec_max));

        curr->sum_exec_runtime += delta_exec;
        schedstat_add(cfs_rq->exec_clock, delta_exec);

        delta_vruntime = calc_delta_fair(delta_exec, curr); //[2017320123][KimJangYoung] Edit code for efficiency
        curr->vruntime += delta_vruntime;                   //[2017320123][KimJangYoung] Edit code for efficiency
        curr->delta_vruntime = delta_vruntime; //[2017320123][KimJangYoung] Store delta_vruntime in sched_entity
```

두 번째로 fair.c 파일의 updata_curr 함수에서 매개변수로 넘어온 sched_entity 구조체에 실제로 delta_vruntime 을 저장해주었다. sched_entity 에는 Total virtual runtime 을 저장하기 위한 vruntime 변수가 존재하고, 이 변수에 더해지는 calc_delta_fair 함수의 결과값은 delta_vruntime 에 해당하는 값이다. 따라서 해당 리턴 값을 별도의 변수에 저장하여, sched_entity 의 vruntime 에는 더해주고, delta_vruntime 에는 할당하도록 하였다. 다음에 설명할 sched_info_depart 함수에서 할당하였던 delta_vruntime 을 출력하는 데에 사용할 것이다.

IV.3. stats.h

```
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;

        /*
         * #[2017320123][KimJangYoung]
         * if process scheduled 1,000 times, then print logs
         * - logs = tgid, delta_vruntime, CPU burst, priority
         * - sampling_unit = 1,000
         * - nivcsw(context-switching times) = scheduled times
         */
        if ((t->nivcsw) % 1000 == 0) {
                printk(KERN_INFO "tgid, %d, delta_vrun, %lld, delta_rrun %lld, prio, %d\n", t->tgid, t->se.delta_vruntime, delta, t->prio);
        }

        rq_sched_info_depart(rq, delta);

        if (t->state == TASK_RUNNING)
                sched_info_queued(rq, t);
}
```

마지막으로 stats.h 에 존재하는 sched_info_depart 함수에서 로그를 출력하도록 하였다. 이 함수는 프로세스가 CPU 점유를 마쳤을 때 호출되는 함수로 인자로 task_struct 구조체를 받는데, CPU 점유를 마친 프로세스의 PCB 역할을 하는 task_struct 구조체의 포인터 t 를 통해 task 의 tgid, priority, sched_entity 에 저장해 두었던 delta_vruntime 등에 접근할 수 있다. 함수의 첫줄에는 delta 라는 변수가 선언되고 초기화 되는데, "현재 시간 – 프로세스가 CPU 점유를 시작했을 때의 시간"을 할당하는 것으로 미루어보아 CPU-burst time 을 나타내는 것을 알 수 있다.

다음으로는 매번 로그를 출력하는 것이 아닌 burst-time 이 1,000 회 바뀔 때마다 1 회 샘플링 하여 기록해야 하므로, t->nivcsw 변수를 사용했다. Bootlin 을 통해 task_struct 구조체를 살펴본 결과, nivcsw 변수는 Context-switching 이 될 때마다 1 씩 증가하는 변수로서, 이를 통해 burst 값이 바뀔 때마다의 샘플링을 구현할 수 있었다. Task_struct 구조체 내에 아래와 같이 선언되어 있다. 첫번째 변수는 voluntary context switches (**N**umber of **V**oluntary **C**ontext **Sw**itches) 를 두번째 변수 involuntary context switches (**N**umber of **I**n **V**oluntary **C**ontext **Sw**itches)를 뜻하는 변수이다.

```
/* Context switch counts: */
unsigned long                   nvcsw;
unsigned long                   nivcsw;
```

V. 과제 수행 시의 문제점과 해결 과정 또는 의문 사항

V.1. 문제점과 해결과정

V.1.1 샘플링의 기준

프로세스가 1,000 회 스케줄링 될 때마다 샘플링을 해야했는데, 스케줄링이 되는 것을 어떻게 counting 할 수 있을 지의 문제가 있었다. 처음에는 task_struct 내부에 count 하는 변수를 생성하여 sched_info_depart 함수에서 증가시키려고 했으나, 해당 변수의 initialize 가 어디에서 일어나는지 알 수 없어서, 내부에서 스케줄링을 카운트하는 변수를 찾게 되었다.

Bootlin 을 통해 Task_struct 구조체 내부를 살펴보니 context-switching 을 카운트 하는 변수를 찾게 되었다. context-switching 은 곧 프로세스 간의 문맥 전환을 뜻하고, 이를 스케줄링 되는 시점으로 볼 수 있다라고 판단했다.

V.1.2 리눅스에 카카오톡 설치하기

카카오톡이 리눅스를 지원하지 않음에 따라 우분투 18.04 에 카카오톡을 설치하는 데에. 어려움이 있었다. 리눅스에는 Wine 이라는 소프트웨어가 있는데, Wine 은 Linux 기반 시스템에서 Windows 용으로 설계된 프로그램을 실행할 수있게 해주는 특별한 응용

프로그램이다. 이 라이브러리는 Windows 라이브러리를 사용하고 다른 시스템에서 작동하도록 개발된 소프트웨어를 Linux 시스템 호출로 대체한다. 아래는 작성한 카카오톡 설치 관련 스크립트이다.

```
sudo apt install wine-stable
WINEARCH=win32 WINEPREFIX=~/.wine wine wineboot
wget https://raw.githubusercontent.com/Winetricks/winetricks/master/src/winetri
cks
chmod 777 winetricks
sudo apt-get install cabextract
./winetricks

# 1. Select the default wineprefix 체크(Checked)
# 2. Install a Windows DLL or component 체크(Checked)
# 3. gdiplus, msxml6, riched30, wmp9 체크(Checked / 거꾸로 찾으시면 금방 찾습니다)

wget http://app.pc.kakao.com/talk/win32/xp/KakaoTalk_Setup.exe
cp KakaoTalk.desktop /usr/share/applications/
```

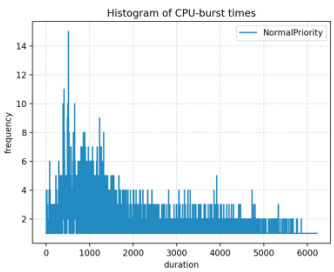
V.2. 의문사항

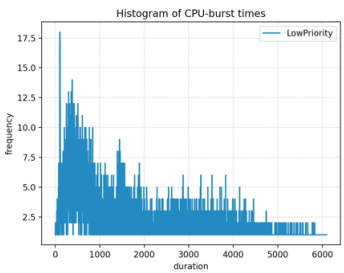
V.2.1 Normal Priority 에서의 실험 결과

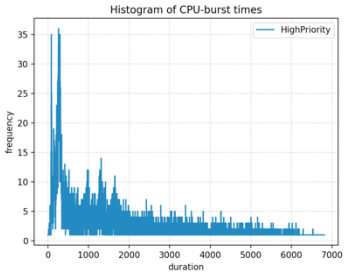
IV.4.2.에서 Normal Priority 의 실험 결과에 대한 의문사항을 제시했다. 첫째로 **CPU-burst 와 vruntime 의 누적 합 그래프에 약간의 차이**가 생기는 부분의 경우, 실제 런타임 과정 중의 오버헤드일 것으로 예상이 된다. Vruntime 의 경우 우선순위를 반영하게 되고, normal priority(120) 인 경우에는 vruntime 과 실제 cpu-burst 가 같아야 하는데, 10% 가량의 sample 에서 그 두 개의 값이 다른 경우가 있었다. 오버헤드에 대해서는 두 가지 정도로 예상이 되는데, 첫 번째는 함수 호출 간의 딜레이가 누적되어 오버헤드로서 작용했다는 것이다. 하지만 이 경우 모든 sample 에서 발생한 것이 아닌 10%에서만 발생하는 오버헤드 임을 설명하지 못한다. 두 번째로는 하드웨어적인 딜레이로 인해서 10% 정도의 sample 에서 오버헤드가 발생한다는 가설이다. 실험 과정 중 잘못된 부분이 있는 것이 아니라면 두 번째 가설이 조금 더 설득력이 있는 것으로 보인다. 가설의 경우 확실하지 않으므로 이 부분을 의문사항에 작성하였다.

(5/27) 첫 번째 의문사항에 대하여 CPU burst 출력을 위해 sched_info_depart 함수의 delta 변수를 사용할 경우 normal priority 일 때 vruntime 과 CPU burst 가 다르게 나오는 경우가 있다는 공지가 올라왔다. 이에 첫번째 의문사항은 delta 변수 사용으로 인한 오버헤드임이 확인되었다.

두 번째로 **Sample 수는 비례하지만 Total CPU-burst time 이 비례하지 않는 문제**의 경우, 측정 상의 오류가 있는지에 대하여 재 측정을 진행했지만, 크게 변화하는 부분은 없었다.







위의 그래프는 CPU-busrt 값에 대한 우선순위 각각의 히스토그램인데, 실제 runtime 에서의 각종 오버헤드를 감안하더라도 조금 이해할 수 없는 결과이기에 의문사항에 작성하였다. 결과적으로만 해석할 때에는 Normal priority 의 경우, 전반적으로 duration 이 긴 CPU-burst time 의 빈도가 높음을 볼 수 있다.