프로세스 및 리눅스 스케줄링의 이해

2017320114 컴퓨터학과 최재원

2021. 6. 12. (Freeday 4일 사용)

I. 과제 개요 및 프로세스와 리눅스 CFS 스케줄러의 개념

(1) 프로세스

프로세스란 쉽게 말해서 실행 중인 프로그램이다. 프로세스는 text section으로 알려진 프로그램 코드 이상이다. 프로세스는 또한 Program Counter의 값과 처리기 레지스터의 내용으로 대표되는 현재 활동을 포함한다. 프로세스는 일반적으로 함수의 매개변수, 복귀 주소와 로컬 변수와 같은 임시적인 자료를 가지는 프로세스 Stack과 전역 변수들을 수록하는 Data Section을 포함한다. 또한 프로세스는 프로세스 실행 중에 동적으로 할당되는 메모리인 Heap을 포함한다. 각 프로세스는 PCB(Process Control Block)을 통해 관리되고 pid를 통해 각 프로세스를 구분한다. 우리가 작업하는 환경에서는 여러가지 프로세스가 time slice 단위로 번갈아 가면서 실행이 되고, CPU Scheduler는 각 프로세스가 얼마 만큼의 CPU time을 할당 받을지 정해준다. 만약 정해진 time quantum안에서 다 소모가 된다면 context switch를 통해 다른 process로 넘어가게 된다. 결과적으로 프로세는 운영체제의 스케줄러가 관리하는 최소의 단위라고 할 수 있다.

(2) CFS 스케줄러

스케줄러는 어떻게 어느 프로세스를 실행할 것인지 그리고 그 프로세스가 어느 시간만큼 CPU를 점유할 것인지를 경정한다. 운영체제의 스케줄링의 목표는 각각 의도하는 것에 따라서 달라지게 되는데, 보통 우리가 쓰는 컴퓨터의 목적은 응답 시간을 최소화하는데 목표를 가진다. 따라서 스케줄링도 응답시간의 최소화에 중점을 둬서 스케줄링이 이루어지고, 반면에 CPU의 성능을 주 목적으로 두는 슈퍼 컴퓨터 같은 경우에는 최대의 CPU 사용률을 목적으로 스케줄러가 프로세스를 관리하게 된다.

리눅스에서 사용되는 CFS(Completely Fair Scheduler)는 영어 뜻 그대로 “완벽하게 공정한 스케줄러”라고 해석할 수 있다. 즉, run queue에서 실행 대기 상태로 기다리는 프로세스를 공정하게 실행하도록 기회를 부여하는 스케줄러이다. 프로세스에게 time quantum을 제공할 때, 성능과 공평성 간의 균형이 중요한 스케줄러이다.

CFS 스케줄러는 프로세스마다 설정된 priority를 기준으로 vruntime을 설정합니다. Vruntime이란 가상 실행 시간을 의미하며, 프로세스가 그 동안 실행한 시간을 정규화한 시간 정보라고 할 수 있다. CFS는 run queue에 실행 대기 상태에 있는 프로세스 중 vruntime이 가장 적은 프로세스를 다음에 실행할 프로세스로 선택합니다. Priority가 높은 프로세스는 우선 순위가 낮은 프로세스에 비해 vruntime이 더 서서히 증가합니다. 즉, priority가 높은 프로세스가 낮은 프로세스에 비해 vruntime이 덜 증가한다는 것이다.

II. CPU burst, vruntime에 대한 그래프 및 결과분석

(1) 실습 환경 및 Priority, Process 설정

실습은 Virtual Box를 통한 Ubuntu (64-bit) 환경에서 진행을 했다. Normal Priority, Low Priority, High Priority를 각각 30분씩 1시간 30분 정도 실습을 진행 했으며, 실행 시킨 프로세스는 기본 커널 데몬 및 쓰레드 외에, Firefox를 통한 유튜브 노래 및 영상 재생, Thunderbird Mail, RythmBox, 마지막으로 sudoku를 실행시켜 놓았다. Priority를 변경 시킨 프로세스는 Firefox 였으며, Firefox에 대해 normal priority, 즉 default 값으로 설정된 120(NI = 0), Low Priority인 130(NI = 10), 그리고 High priority인 110(NI = 0)으로 설정을 해놓았다. Priority를 바꾸려는 대상을 firefox로 설정한 것은 개인적으로 Youtube 재생 시 소리 출력, 화면 출력, 영상 재생 및 정지 등 여러가지 I/O 가 있을 것이라고 생각했기 때문에 지정을 했다.

USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
jaewon	20	0	4338988	380172	132544	R	27.9	9.4	3:26.71	gnome-shell
jaewon	30	10	3786816	408064	203220	S	20.9	10.1	2:07.12	firefox

<Low-Priority>

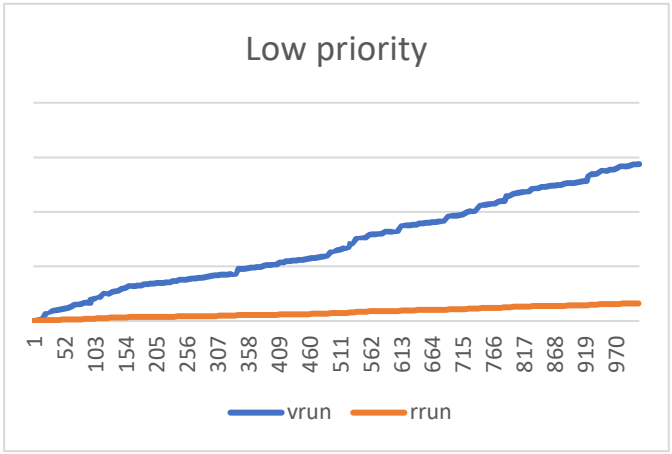
USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
jaewon	20	0	4338988	379888	132548	S	22.8	9.4	3:59.71	gnome-shell
jaewon	10	-10	3786800	406336	203048	S	14.2	10.1	2:31.05	firefox

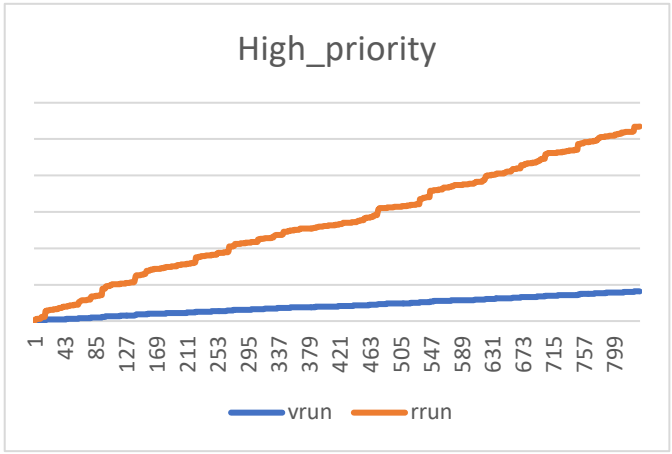
<High-Priority>

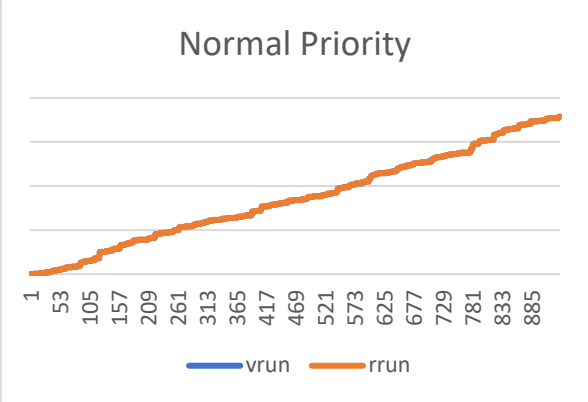
USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
jaewon	20	0	3734676	343984	180128	S	17.9	8.5	0:32.29	firefox
jaewon	20	0	2893856	254056	113416	S	11.3	6.3	0:22.24	Web Content
jaewon	20	0	395488	58108	42524	S	5.0	1.4	0:06.17	RDD Process

<Normal-Priority>

(2) CPU burst, vruntime에 대한 결과 그래프







800번 누적 기준	Normal priority	Low priority	High priority
Total CPU-burst	60649427	52252919	51278611

(3) 그래프 결과 분석

CPU Burst는 CPU 명령을 실행하고 연산을 수행하는 시간을 말하고, I/O Burst는 I/O 요청을 한 다음에 처리를 위해 기다리는 시간을 의미한다.

vruntime += execution time x (default weight/process weight)

- High weight -> small vruntime

사진에서 나온 것처럼, vruntime은 실제 cpu 실행 시간에서 가중치 즉 priority를 곱해서 더해주는 형식이다. Process weight가 분모에 들어가 있으며, 우선순위가 높을수록 weight가 커지게 되어 Vruntime이 적게 증가하게 되고, CFS는 vruntime이 가장 작은 프로세스를 선택하기 때문에, 결과적으로 우선순위가 높은 프로세스가 더 자주 CPU를 점유할 수 있게 된다.

이러한 이론적인 결과에 대해서 실험 결과도 비슷한 경향을 보여줬다. Firefox 에 대해서 우선순위를 두지 않았을 때는 동일하게 누적되서 증가되는 모습을 볼 수 있다. 하지만, Firefox에게 높은 우선순위를 부여한 경우 그림과 같이 vrun time은 rrun time(CPU – Burst)에 비해 현저히 작은 누적 합계를 보였으며, 역시 마찬가지로 우선순위를 낮게 두었을 때는 급격히 누적되며 증가하는 Vrun time의 결과를 볼 수 있었다.

800번의 sampling 횟수를 기반으로 total cpu-burst를 측정한 결과 total cpu-burst 누적시간은 Normal priority가 가장 높게 나왔다.

‘

III. 작성한 소스코드에 대한 설명

(1) /include/linux/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_exec; /*[2017320114][Choi Jaewon] add delta_exec for delta rrun time */
        u64                             delta_vruntime; /* [2017320114][Choi Jaewon add delta vruntime */
        u64                             prev_sum_exec_runtime;

        u64                             nr_migrations;

        struct sched_statistics         statistics;
```

Sched.h 함수에 u64 자료형의 CPU-burst time의 계산을 위한 delta_exec 변수와 마찬가지로 vruntime을 계산하기 위한 u64 자료형의 delta_vruntime 변수를 선언했다.

(2) /kernel/sched/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;

        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);

        curr->delta_exec = delta_exec; //[2017320114][Choi Jaewon] make new delta for rruntime
        curr->vruntime += calc_delta_fair(delta_exec, curr);
        curr->delta_vruntime = calc_delta_fair(delta_exec, curr); //[2017320114][Choi Jaewon] add delta_vruntime in sched entity
        update_min_vruntime(cfs_rq);

        if (entity_is_task(curr)) {
                struct task_struct *curtask = task_of(curr);

                trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
                cgroup_account_cputime(curtask, delta_exec);
                account_group_exec_runtime(curtask, delta_exec);
        }

        account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

Sched.h 함수에서 선언한 구조체를 fair.c 파일에서 update_curr 함수에서 사용하고 있는 것을 볼 수 있다. Curr -> vruntime은 calc_delta_fair 함수를 이용해서 sched.h에서 선언된 vruntime 값을 누적해서 더해주고 마찬가지로 구조체에서 선언된 delta_vruntime의 값을 계산해주는 것을 할당했다. 추후에 delta로 진행하고 나서 공지사항에서 normal priority 일 때 vruntime 과 cpu burst time이 다르게 나올 수도 있다는 것을 확인하고 sched.h 함수에서 delta_exec을 따로 선언한 후 update_curr 함수에서도 계산했다.

(3) /kernel/sched/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;
/*[2017320114][Choi Jaewon] Code for printing tgid, delta_vruntime, CPU burst for every 1000 times when scheduled */
        if(t->sched_info.pcount % 1000 == 0){
                printk("tgid, %d, delta_vrun, %llu, delta_rrun, %llu, prio, %d", t->tgid, t->se.delta_vruntime, t->se.delta_exec, t->prio);
        }
/*[2017320114][Choi Jaewon] Code end for printing output */
        rq_sched_info_depart(rq, delta);

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

Stats.h 파일에서는 sched_info_depart 함수에서 내가 보고자 하는 값들을 출력할 수 있도록 했다. 어떠한 프로세스가 CPU를 다 사용하고 나서 이 함수의 task_struct 구조체를 받고, 이 구조체는 프로세스의 PCB(Process Control Block)의 역할을 해주고 있다. 따라서 이 구조체의 포인터를 이용해서 tgid, 미리 선언해 두었던 delta_vrun 과 delta_rrun 그리고 priority를 접근해서 출력을 할 수 있었다. 또한 bootlin을 통해 sched.h 파일에 있는 task_struct의 구조체를 확인할 수 있었고, 그 과정에서 struct sched_entity 를 se를 통해 받고, priority의 정확한 자료형을 확인할 수 있었다. 또한 tast_struct에서 context switch가 일어나면 프로세스가 호출이 되고, stats.h에 있는 sched_info_arrive 함수에서 sched_info 구조체에 있는 pcount 변수가 1만큼 증가한다는 것을 확인했다. 결과적으로 pcount를 이용해서 1,000번째 마다 해당 값을 출력 할 수 있도록 pcount 변수를 사용해서 구현을 했다. 마지막으로 pcount는 sched_info 구조체를 가지고 있는 sched_info_depart의 t 를 통해 접근을 하도록 구현을 했다.

```
static void sched_info_arrive(struct rq *rq, struct task_struct *t)
{
    unsigned long long now = rq_clock(rq), delta = 0;

    if (t->sched_info.last_queued)
        delta = now - t->sched_info.last_queued;
    sched_info_reset_dequeued(t);
    t->sched_info.run_delay += delta;
    t->sched_info.last_arrival = now;
    t->sched_info.pcount++;
```

IV. 과제 수행 시의 문제점과 해결 과정

(1) Excel을 통한 누적 결과값 시각화

처음에 dmesg > log.txt 를 이용해서 30분 동안 특정 priority를 기준으로 실행한 결과값을 봤을 때 이것을 어떻게 시각화를 할지에 관해서 많은 고민이 있었다. Excel에 숙련도가 높지가 않아서 처음에 텍스트 파일에 출력된 결과값들을 엑셀에서 각 열 별로 그리고 셀 별로 정리하는데 시간이 걸렸다. 또한 누적값을 계산해서 그래프로 시각화 하는 과정에서 주어진 자료와는 조금은 다르게 결과가 나오는 것을 확인하였고, 결론적으로는 내 엑셀 함수가 필터링을 못하고 모든 프로세스에 대한 누적값을 구했다는 것을 알 수 있었다. 따라서 새로운 sheet에 필터링을 거친 특정 프로세스의 값을 복사해서 옮겼고, 결과적으로 원하던 누적 그래프의 모습을 보일 수 있었다.

(2) Ubuntu의 성능 문제

커널 코딩을 마치고 우선순위에 따른 출력 값을 도출해 내기 위해 프로세스를 돌리려고 할 때, 최대한 많은 프로세스를 실행시켜보고자 설정해준 ubuntu의 성능을 간과하고 너무 많은 프로그램을 키게 되었다. 결과적으로 ubuntu가 2~3번 견디지 못하여 꺼지는 상황이 발생하였고, 프로그램을 조금 줄여서 우선순위를 바꿀 main process만 집중해서 실습을 진행하게 되었다. 2차 과제를 위하여 ubuntu의 RAM memory 사이즈를 조금 늘렸으면 좋았을 것 같다.