COSE341 Operating Systems

2nd Assignment Report

학과: 경영학과

학번: 2017120486

이름: 서천함

제출 날자: 2021 년 6 월 13 일

프리 데이 사용 일수: 5

1. Introduction & Development Environment

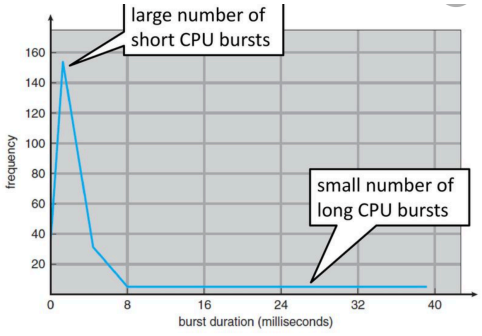
In order to understand processes and process schedulers in Linux, we can explore CPU burst time and vruntime of processes in this assignment. Since the CFS scheduler in Linux aims at giving each process a fair share of processor, it will weight the vruntime value so that it can schedule based on priority of processes. In this assignment, we will investigate on how the CFS scheduler allocates CPU processing time to processes in Linux OS by measuring the vruntime and CPU-burst time of tasks at different priority levels.

Development Environment: Ubuntu 18.04.02(64bit), Linux Kernel 4.20.11

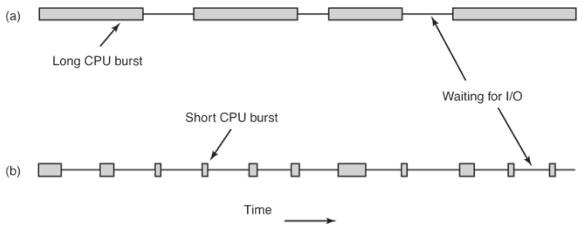
2. Concept Illustration

a) Process & CPU-I/O Burst Cycle

Process execution consists of a cycle of CPU execution and I/O wait. CPU burst is when the process is being executed in the CPU. I/O burst is when the CPU is waiting for I/O for further execution. Processes alternate between these two states, whose execution begins with a CPU burst and then followed by an I/O burst, and so on. The duration of CPU bursts can vary greatly from process to process and from computer to computer. However, CPU bursts tend to have a frequency curve similar to the exponential curve shown below as Graph 2.1, characterized by a large number of short CPU bursts and a small number of long CPU bursts. We can conclude that a CPU-bound program has a few long CPU bursts (Graph 2.2.a) and an I/O-bound program has many short CPU bursts (Graph 2.2.b).



<Graph 2.1>



<Graph 2.2>

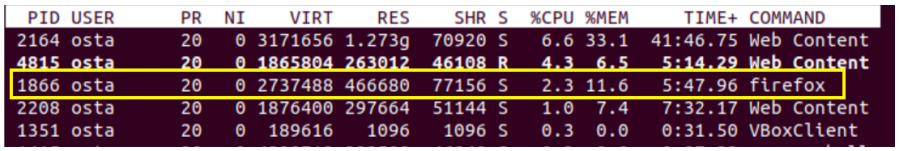
b) Linux Scheduling: Completely Fair Scheduler(CFS)

CPU scheduling decides which of the processes in the ready queue will be allocated to the CPU's core. In Linux, Completely Fair Scheduler (CFS) is used by default, which attemps to give all processes a fair share of the processor. To determine the balance, CFS maintains virtual time(vruntime) for each process, which is the amount of time provided to a given task. Whenever a context switch happens or at every scheduling point, current running process virtual time is increased by real execution time scaled by the weighting factor:

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

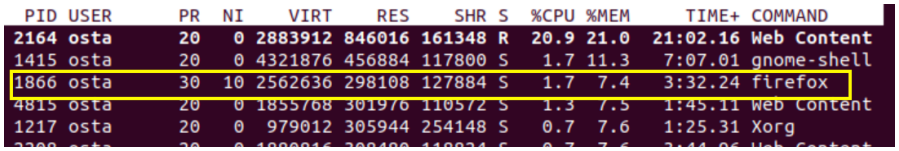
CFS selects the process with the minimum vruntime. Processes with lower nice value will receive a higher proportion of CPU processing time, indicates a higher relative priority. Comparing with CPU-bound process, I/O bound process runs only for much shorter periods on a processor. Therefore, the value of vruntime will be lower for I/O bound, giving it higher priority and making it eligible to run or preempt the CPU-bound task while waiting. In this way, starvation of I/O-bound process can be avoided.

3. CPU Burst & Vruntime Graph & Results Analysis



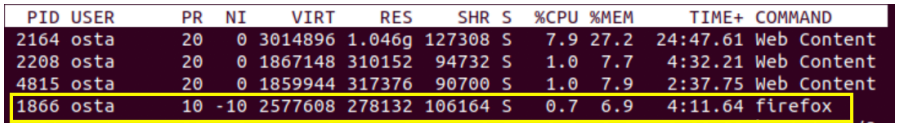
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2164	osta	20	0	3171656	1.273g	70920	S	6.6	33.1	41:46.75	Web Content
4815	osta	20	0	1865804	263012	46108	R	4.3	6.5	5:14.29	Web Content
1866	osta	20	0	2737488	466680	77156	S	2.3	11.6	5:47.96	firefox
2208	osta	20	0	1876400	297664	51144	S	1.0	7.4	7:32.17	Web Content
1351	osta	20	0	189616	1096	1096	S	0.3	0.0	0:31.50	VBoxClient

<Normal Priority: 120 (Default), Nice: 0>



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2164	osta	20	0	2883912	846016	161348	R	20.9	21.0	21:02.16	Web Content
1415	osta	20	0	4321876	456884	117800	S	1.7	11.3	7:07.01	gnome-shell
1866	osta	30	10	2562636	298108	127884	S	1.7	7.4	3:32.24	firefox
4815	osta	20	0	1855768	301976	110572	S	1.3	7.5	1:45.11	Web Content
1217	osta	20	0	979012	305944	254148	S	0.7	7.6	1:25.31	Xorg

<Low Priority: 130, Nice: 10>



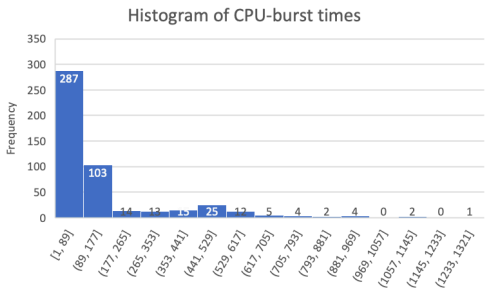
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2164	osta	20	0	3014896	1.046g	127308	S	7.9	27.2	24:47.61	Web Content
2208	osta	20	0	1867148	310152	94732	S	1.0	7.7	4:32.21	Web Content
4815	osta	20	0	1859944	317376	90700	S	1.0	7.9	2:37.75	Web Content
1866	osta	10	-10	2577608	278132	106164	S	0.7	6.9	4:11.64	firefox

<High Priority: 110, Nice: -10>

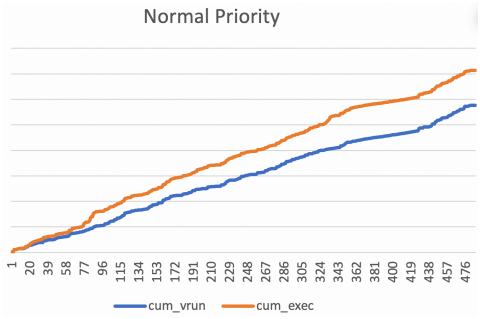
The web browser Firefox was chosen as the measured process. The priority of the process

can be changed by using the command `sudo su` and `ls /proc/$PID/task| xargs renice $PRIO`. The measurement time for each level of priority was about 40 minutes. The printed information can be collected by using the command `dmesg > log.txt`.

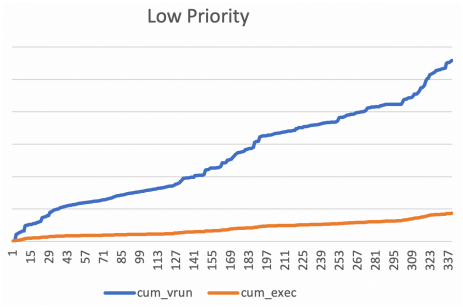
From the histogram of CPU-burst times of normal priorities shown below, we can see that there're large number of short CPU bursts and small number of long CPU bursts. Therefore, we can conclude that the web browser Firefox is an I/O-bound program according to our observation, which spends more time waiting on I/O than other potential bottlenecks such as CPU resources. In fact, we know that web browser is a program with a large number of I/O operations so we can verify the rationality of the result based on our experience.



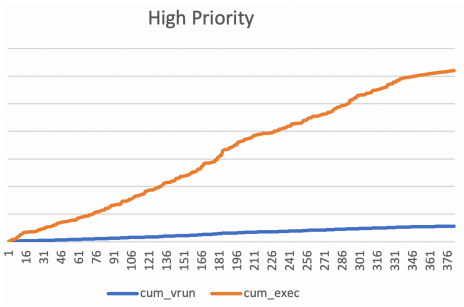
<Graph 3.1>



<Graph 3.2>



<Graph 3.3>



<Graph 3.4>

	Normal Priority	Low Priority	High Priority
Total CPU-burst	71389839	43351477	62056539

<Graph 3.5>

The results of cumulative sum of vruntime and CPU-burst time of different levels of priorities are shown above. We can observe that the results of both low priority and high priority are just almost the same as the predicted results. In accordance with the theory,

processes with lower nice value indicates a higher relative priority and will receive a higher proportion of CPU processing time.

As for process with high priority(graph 3.4), the cumulative sum of execution increases largely while the cumulative sum of vruntime increments relatively slowly, indicating that the CFS works to assign a shorter vruntime to higher-priority tasks so that the scheduler can select the task since it has smaller vruntime and will have more opportunity to run on the processor. This can represents the case of I/O-bound task and shows that the CFS scheduler actually works to give all processes a fair share of the processor. On the contrary, shown in graph 3.3, as the cumulative sum of CPU-burst time increases, the cumulative sum of vruntime increases substantially. This indicates that since lower-priority process exhaust the time period whenever it has an opportunity to run on a processor, the CFS scheduler will give lower proportion of CPU processing and less opportunity to such processes to run on CPU so that a higher-priority task can run. Both graphs all verify the “fairness” of CFS scheduler and proves that a higher-priority task(I/O-bound task) can preempt a lower-priority task(CPU-bound task) and be made eligible to run.

4. Modification of Source Code

a) 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; // #[2017120486][서천함] add to store delta virtual runtime
    u64             prev_sum_exec_runtime;
```

In order to print the vruntime of each process in function `sched_info_depart` , we need to declare the variable `delta_vruntime` to store this value.

b) 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; // #[2017120486][서천함] add variable to store delta virtual runtime

    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); // #[2017120486][서천함]calculate delta_vruntime
    curr->vruntime += delta_vruntime; // #[2017120486][서천함]increment vruntime by delta_vruntime
    curr->delta_vruntime = delta_vruntime; // #[2017120486][서천함]need to store the value of delta_vruntime to sched_entity
```

Here, we need to store the value of `delta_vruntime` in struct `sched_entity`. Firstly, we need to declare the variable `delta_vruntime` in function `update_curr`. This function is used to update the vruntime of each process. The variable `vruntime` stores the cumulative sum of vruntime of processes so we need to modify the existing code. After `delta_vruntime` is calculated by function `calc_delta_fair`, we increment `vruntime` by adding it and then store the value of `delta_vruntime` in struct `sched_entity` pointed by struct pointer `curr`.

c) 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;

    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);
    }
    /* #[2017120486][서천함] nivcsw represents the number of involuntary context switched, indicating the count of scheduled times
       To avoid bottleneck, 1000 units of scheduling per process have been sampled here. Then we print the information.
    */

    rq_sched_info_depart(rq, delta);

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

Lastly, we need to print the tgid, vruntime, CPU burst time as well as the priority by modifying function `sched_info_depart`. To avoid the effect of bottleneck caused by the measurement, 1000 units of scheduling per process are sampled here using the variable `nivcsw` defined in struct `task_struct`. Here, struct `task_struct` serves like the PCB of the

processes executed in CPU. `nivcsw` represents the number of involuntary context switches, which can be used as the count of scheduled times. Besides, by using the struct pointer `t`, we can get the value of `prio` (the priority of each task) and `tgid` (the identifier of each task). Also, the struct pointer `se` pointed to the struct `sched_entity` can be used to get the value of `delta_vruntime`. The variable `delta` is calculated by using `rq_clock(rq)`(current time) minus `t->sched_info.last_arrival` (the start time when process uses CPU) and it represents the time process executed in the run queue, which is the CPU burst time. Therefore, we can print all the information we need.

5. Problems and Solutions

Theoretically speaking, for tasks at normal priority(having nice values of 0), the virtual run time should be identical to the actual physical run time. However, the result in graph 3.2 shows difference between the cumulative sum of vruntime and CPU-burst. According to the announcement on blackboard, if we use variable `delta_exec` in function `update_curr` instead of using the variable `delta` in function `sched_info_depart` , then the result should be the same. So the difference can be explained as the overhead when using variable `delta`. Therefore this problem is solved.