CSE 5433 OS Laboratory
Lab3 Report

Group 8
Sixiang Ma, Xiaokuan Zhang

10/09/2017

1 Objectives

Implement the fair-share scheduling. Fair-share scheduling is to equally distribute CPU time among users or groups in the system, as opposed to equal distribution among processes. After implementing the scheduler, we need to write kernel instrumentation code for profiling the kernel performance in terms of the schedulers behavior. Also, we are required to write some scripts to post-process the profiled data and plot figure(s).

2 Scheduling Policy Design

To implement this scheduling policy, there are several key questions we need to answer. To solve these questions, in general, there are two approaches to implement the fair-share scheduling.

2.1 Key Questions

(a) How to implement a system call to change the scheduling policy at run-time?

(b) How to know how many processes the current user has?

(c) How to calculate the time slice to reflect the new policy?

2.2 Approach 1

The first approach is to maintain a global flag indicating whether we should use the new policy. When the flag is set, calculate the timeslices of processes with `SCHED_NORMAL` policy according to the number of processes the current user has. For example, if the number is N, the new timeslice would be:

$$\text{New Timeslice} = \frac{\text{Old Timeslice}}{N} \tag{1}$$

2.3 Approach 2

The second approach is to implement a new policy `SCHED_CFS`. When encounter a process with such a policy, calculate the timeslice accordingly. Also, it is required to keep track of how many `SCHED_CFS` processes are currently running for each user.

3 Scheduling Policy Implementation

We implemented Approach 1, which is a lighter-weight approach compared to Approach 2. We will explain the implementation in detail and show how we solve the key questions.

3.1 Answers to Key Questions

3.1.1 Adding a System Call to Set Policy

We added a global flag, $sched\_policy\_flag$. If $sched\_policy\_flag == 1$, we schedule `SCHED_NORMAL` processes using our new policy.

```
int sched_policy_flag = 0;
EXPORT_SYMBOL(sched_policy_flag);
```

Moreover, we add a system call to switch the flag between 0 and 1. We also change syscall-related files accordingly.

```
asmlinkage long sys_sched_policy_switch(void) {
        sched_policy_flag = 1-sched_policy_flag;
        printk("SCHED FLAG: %d\n",sched_policy_flag);
        return 0;
}
```

3.1.2 Finding the Number of Processes

Suppose current process is $*p$. We utilized $task\_struct$ and $user\_struct$. We read (p→user→processes).counter (denoted N) to get the process count for the current user. However, this counter covers all the processes, including two irrelevant process: *sshd* and *bash*. So we need to use $N-2$ to get the true value.

3.1.3 Calculating the New Timeslice

We modified the timeslice calculation mechanism to reflect the new policy. If 1) the flag is set; 2) the user id is in the test range (601 to 605); 3) it is a `SCHED_NORMAL` process, we update the timeslice using our new scheme.

```c
static unsigned int task_timeslice(task_t *p)
{
    int N;
        if (p->static_prio < NICE_TO_PRIO(0))
                return SCALE_PRIO(DEF_TIMESLICE*4, p->static_prio);
        else
                if (sched_policy_flag == 0)
                    return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
                else
                {
                    if (((int)(p->user->uid) > 600) && ((int)(p->user->uid) < 606))
                    //test range
                    {
                        N = (p->user->processes).counter;
                        if (N < 3) N = 3;

                        return SCALE_PRIO(DEF_TIMESLICE/(N-2), p->static_prio);
                    }
                    else
                        return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
                }
}
```

4 Kernel Profiling

4.1 Profile CPU Time in the Kernel

Now that the new scheduling policy has been implemented, we need to check its correctness in that CPU time can be equally distributed among users in the system. To this end, we have to profile the kernel performance in terms of how CPU time is assigned to processes among a group of users.

Specifically, in *kernel/sched.c*, we declare a new struct called *struct pid_cpu_stat* to store the accumulative CPU time that a process has used during a period of time. Also, we declare and define another struct called *struct uid_cpu_stat uid_stats* to collect all the *pid_cpu_stat* belonging to each system user.

```c
#define STAT_ARRAY_MAX 10

struct pid_cpu_stat {
        int pid;
        unsigned long accum_run_time;
};

struct uid_cpu_stat {
        unsigned int uid;
        struct pid_cpu_stat pid_stats[STAT_ARRAY_MAX];
};

struct uid_cpu_stat uid_stats[STAT_ARRAY_MAX];
EXPORT_SYMBOL(uid_stats);
```

When a process is going to be scheduled out, we add the local variable *run_time* in the *schedule*() function to the *accum_run_time* variable of the corresponding *pid_cpu_stat* which this process belongs to. The *run_time* varaible stores the difference between the time a process is scheduled in and scheduled out.

4.2 Manipulate Profiling by a System Call

To facilitate users to start and stop/dump *struct uid_cpu_stat uid_stats* in the kernel, we add a new system call named *sys_sched_profile_switch*. Base on this system call, it is easy for users to manipulate the profiling functionality as shown in the following lines of code:

```
int main() {
        sched_profile_switch();
        return 0;
}
```

By invoking *sched_profile_switch*(), the kernel cleans up the data for storing accumulative CPU time and starts to profile. By invoking *sched_profile_switch*() again, the kernel dumps the statistics to stdout as follows and stops profiling.

```
uid = 601
    pid = 3439, accum_run_time = 11503000
    pid = 3440, accum_run_time = 11500000

uid = 602
    pid = 3443, accum_run_time = 7611000
    pid = 3442, accum_run_time = 7590000
    pid = 3441, accum_run_time = 7590000
```

5 Evaluation

To evaluate the correctness of the new scheduling policy, we lauchned different numbers of processes (3-6) for UserA, UserB, UserC, UserD and performed performance profiling about 60 seconds. The results are shown in Table 1. We can observe that CPU time is equally distributed among users. In addition, for a given user, CPU time is equally distributed among its processes. These results proved that the new scheduling policy works correctly as expected.

	Process 1	Process 2	Process 3	Process 4	Process 5	Process 6	Sum
User A	8.33	8.33	8.33	N/A	N/A	N/A	24.99
User B	6.35	6.35	6.35	6.32	N/A	N/A	25.37
User C	5.05	5.08	5.08	5.08	5.08	N/A	25.37
User D	4.06	4.04	4.04	4.04	4.04	4.04	24.26

Table 1: CPU time percentage (%) of processes for each user.