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 Design

5 Kernel Profiling Implementation

6 Evaluation