Lab3 Report

Yao Wei (wei.849@osu.edu)
Zuanxu Gong (gong.366@osu.edu)

1. Design

1) Fair-Share Scheduling

static unsigned int task_timeslice(task_t *p) function maps the process "nice" value to the CPU time slice, to achieve fair share schedule , we can assign total 100ms to each user's processes. We can use p->user->processes to get the number of processes each user, so each process gets 100/ p->user->processes time slices.

2) Kernel Profiling

For each process, we log relevant statistics per second. Statistics include uid, pid, current time, accumulated runtime and interval between printing. With these statistics, we can calculate CPU usage for each process and each user.

3) User-level data processing

Based on the profiling data, draw a table to show the way how the CPU time is distributed among existing processes and users within a configurable period of time. Write a bash to store "dmesg" message in a text file and call python code to read this text file and then print the table. There is one input argument to control the period of time in seconds. This table shows the CPU Usage percentage for each user and process. User is presented by uid, while process is presented by pid.

2. Implementation

1) Fair-Share Scheduling

Firstly, we define a variable static int enable_fs to control the Fair-Share Scheduling policy and define two system calls to setstate and getstate.

```
asmlinkage long  sys_set_fs_state(int enable){
        enable_fs=enable;
        return 0;
}
asmlinkage long  sys_is_fs_enabled(){
        return enable_fs;
}
```

We override the schedule policy name SCHED_NORMAL, in static unsigned int task_timeslice(task_t *p) function in sched.c. We add

```
if (enable_fs&&p->user != &root_user&&p->policy==SCHED_NORMAL){
        //fair schedule policy, non root user, and not FIFO/RR policy
        return max(100/atomic_read(&p->user->processes),MIN_TIMESLICE);
}
```

to task_timeslice function, so if the fair_schedule is enabled and current user is not root user and schedule policy is SCHED_NORMAL, we assign to each process 100/number of processes per user for each process.

2) Kernel Profiling

We define a variable static int enable_scheprof to enable or disable the profile function.

Furthermore, we define two system calls to set or get the profile function's state.

```
asmlinkage long sys_set_scheprof(int enable){
        enable_scheprof=enable;
        return 0;
}
asmlinkage long sys_is_scheprof_enabled(){
        return enable_scheprof;
}
```

For struct task_struct, we add two attributes (runtime_print and timestamp_print). "runtime_print" is used to record the accumulated runtime between printing. After each printing, "runtime_print" will be set to 0 and start the new accumulation. "timestamp_print" is used to record the last print timestamp. If current time - timestamp_print > 1s, then we will print statistics. After printing, timestamp_print will be set to current time.

For printing, we use printk(KERN_WARNING "fairSchedule: %u %d %llu %lu %lu\n", prev->uid, prev->pid, now, prev->runtime_print, print_interval); Here, timestamp_print = current time - timestamp_print.

Finally, we can type dmesg in terminal to see the log. Example is shown as below.

```
fairSchedule: 504 2755 4300064791000000 75000000 1197000000
fairSchedule: 503 2723 4300064932000000 99000000 1197000000
fairSchedule: 503 2722 4300064933000000 99000000 1197000000
fairSchedule: 502 2602 4300065015000000 150000000 1197000000
fairSchedule: 501 2584 4300065159000000 299000000 1197000000
fairSchedule: 504 2754 4300065216000000 75000000 1198000000
```

3) User-level data processing

First, we use python to read dmesgLog.txt one line by one line. For each line, we will split them with white space. Then, we will check whether the length equals to 6 and the first data is “fairSchedule”. If these conditions are all satisfied, we will put them into the dictionary and array for easy use.

After that, for each process, we will accumulate runtime and interval for records whose timestamp is larger than the largest timestamp minus X seconds. X is provided by user (Higher X will lead to more stable statistics). Then, we can get CPU usage by totalRuntime / totalInterval. If we accumulate one user’s processes’ CPU usage, we can get the CPU usage of that User.

Example is shown as below. X was set to 80 seconds, which means our table uses the statistics from the past 80 seconds. We ran 4 users with uid 501, 502, 503, 504. Each user has 25% CPU usage. The CPU usage for each user’s processes are also been equally distributed.

```
[root@cse5433-vm03 testlab3]# ./showCPUUsage.sh 80


          User  504
Process 2754   0.0629025539142
Process 2752   0.062481967108
Process 2753   0.0625120215426
Process 2755   0.0625062506251
================================
Total:   0.25040279319
================================


          User  502
Process 2602   0.124406518671
Process 2601   0.12501250125
================================
Total:   0.249419019921
================================


          User  503
Process 2723   0.0825158684362
Process 2722   0.0825158684362
Process 2724   0.0825158684362
================================
Total:   0.247547605309
================================


          User  501
Process 2584   0.248240624699
================================
Total:   0.248240624699
================================
```