CS342 Operating Systems – Fall 2017
Project 2: CFS Scheduler Simulation
Document last updated: Nov 15, 2017, 00:40

Assigned: 25 Oct 2017
Due date: 12 Nov 2017, 23:55

Objectives:

- *Learn a real CPU scheduling algorithm in detail; implement it in a simulator.*
- *Practice writing a simulation program.*
- *Practice designing and conducting experiments and interpreting results.*
- *Learn how to generate a random variate (exponential).*
- *Apply knowledge of probability and statistics.*

You will do this project individually. You have to program in C under Linux. You are recommended to use the following distribution of Linux: Ubuntu 16.04 – 64 bit.

In this project, you will simulate and evaluate the Completely Fair Scheduler (CFS) algorithm of the Linux operating system. The algorithm is described in the textbook. There are also a lot of other resources describing the algorithm. If interested, you can also learn about it directly from the Linux source code (linux-src-tree-root/kernel/sched/fair.c) Not everything specified here will be the same with the real CFS scheduler, but the idea will be similar. You will write a simulation program. You will use the program to do a lot of simulation experiments to evaluate the scheduling algorithm. As you know, CFS scheduling algorithm basically models an ideal, precise multi-tasking CPU on real hardware.

Your program will take a workload file as input. The workload file will describe a set of processes (i.e., tasks) with their sequence of cpu and i/o burst definitions. The number of i/o bursts will be 1 less than the number of cpu bursts for a process. Your program will read the workload file and will simulate the running of the processes in a single cpu system that is using CFS algorithm. When all processes are simulated, the simulation will end and some statistics will be printed out. An output file showing the execution trace will also be generated.

The following statistics will be printed out:

- The *turnaround time* of each process.
- The *waiting time* of each process. The waiting time of a process includes all waiting happening in the runqueue (ready queue) during the lifetime of the process. It is not including waiting for i/o.
- For each process, the average waiting time till its next cpu execution, i.e., the *response time*. Here, we are defining response time as the time it passes from when a process returns from an i/o wait (or from starting new) till it starts running in the cpu. Do not consider the cases where a process is taken out from the cpu and is waiting in the runqueue till its next execution. Just consider the cases where a process returns from i/o wait. You will measure the response time for a process in all such cases and then take the average. This will be the response time for the process.

The statistics output will be in the following format:

<pid> <prio> <starttime> <finishtime> <turnaround> <waittime> <responsetime>

For example:

17 20 1000 3000 2000 1500 100

18 25 1000 4000 3000 1600 150

The output file will contain information about what has happened in the cpu during the simulation time. This is like a *log* file. For each continuous time interval during which the cpu is running a different job or cpu is idle, the file will contain the information in the following format: <pid> <duration>. This indicates that process <pid> used the cpu for <duration> amount of time. If cpu was idle for some time, then the respective output line will be: idle <duration>. Below is an example. Time starts at 0. So the process 17 started running at 0 and run for 60 ms. Then process 9 started in cpu. That means process 9 started running in the cpu at time 60. It run for 50 ms. At time 110, the cpu started to be idle. It remained idle for 130 ms.

17 60
9 50
idle 130
17 30
18 70
9 20

We will use the outputfile to test your program.

The program will be called **cfs** and will be invoked like this:

cfs <workloadfile> <outputfile>

Here, <workloadfile> is the input file describing a set of processes, their cpu and i/o burst times.

The workload file content will be in the following format:

<pid> start <startime> prio <priority>
<pid> cpu <cpu-bursttime>
<pid> io <io-bursttime>
<pid> cpu <cpu-bursttime>
<pid> <end>

For example, for a process with pid = 17:

17 start 2000 prio 20
17 cpu 70
17 io 150
17 cpu 30
17 io 300
17 cpu 50
17 end

Note that the start time is an absolute time. Simulation will start time 0, and at time <starttime>, a process will start. The units are all milliseconds. In fact, a cpu burst can be much less than 1 ms for a process (1 ms is 1000000 ns, which is a long time for cpu), but for this project we will assume it is a multiple of 1 ms.

The following is the specification related to the CFS algorithm:

- There is priority (prio), or nice value, associated with each process.

- There is vruntime associated with each process. The vruntime for a process will be kept in ns granularity.
- Consider the runqueue (structure) to consist of the following fields (at least): 1) red black tree (rbtree) containing processes that are in ready state (ordered wrt to vruntime), 2) currently running process (it is not part of the rbtree, but it is in the runqueue data structure), 3) min_vruntime (a monotonically increasing value; never decreases), 4) number of processes in the runqueue (includes the running process), 5) load (weight) of the runqueue, which is the total weight of the processes in the runqueue (includes the running process and the processes in the rbtree).
- The runque contains a red-black tree for the ready processes; time ordered with respect to vruntime of processes. If you wish you can use an ordered linked list instead of red-black tree.
- When scheduling decision is to be made, the process that has the smallest vruntime is selected for execution (picked from the rbtree). That means the task that has run the least in cpu is selected.
- Scheduling decision can be made at the following cases (events): when a new process is created; when a process returns from i/o wait, when the time allowed for the process in the cpu has expired, when running process finishes its cpu burst, or when the running process terminates.
- At each timer tick, the vruntime of the running process is updated. The min_vruntime of the runqueue is also updated. It is also checked if the current running process expired its allowed timeslice or finished its cpu burst.
- The minimum granularity will be 10 ms (10 000 000 ns).
- The default targeted latency will be 200 ms (200 000 000 ns). Targeted latency is a period of time (epoch) during which every process in the runqueue needs to get a chance to execute once.
- If there are 20 or less processes in the runqueue, then the targeted latency is 200 ms. But when the number of processes in runqueue (including the running process if any) becomes more than 20 (i.e., 200 ms/ 10 ms), the targeted latency becomes: NR * 10 ms, where NR is the number of processes in the runqueue.
- The timeslice for a process selected for execution is set to be as follows: *timeslice = targeted_latency * load_of_process / total_load.* The total load is the sum of loads (weights) of all processes in the runqueue. The load of the process is its weight (see Table 1). A process can run at most timeslice period of time before it is preempted. The CFS scheduler calculates the timeslice for a process just before the process is dispatched into cpu for execution and also at each timer tick (dynamically in this way). For this project, you don't need to recalculate it when a new process arrives or when a process returns from I/O.
- When a process start in the cpu, the timeslice of the process can not be less than the minimum granularity, which is 10 ms. Of course, a process can voluntarily start an i/o operation and leave the cpu or terminate before 10 ms passes. Otherwise, it will run at least 10 ms after scheduled into cpu.
- Timeslice value will be computed for the running process at each timer tick as well. Hence it is only computed for the running process. When the running process has run in the cpu for timeslice time, it is suspended. This means that the timeslice values of processes are computed dynamically. When process goes into cpu, at that time and at each timer tick, the timeslice of the process is computed using the load of the process and the load of the runqueue. When process is out of

cpu, its timeslice needs not be computed till it is executed again. Timeslice for a process becomes relevant when a process starts executing in the cpu.

- The weight of a process depends on its priority value. Use the Table 1 for this mapping.
- The priority (nice value) of a process can be between 0 and 39 (corresponding to Linux priorities -20 to 19). In Linux, 0 is the default priority. Here, in this project, priority 20 will be the default priority (in the middle of the priority range). We call it NICE0 priority.
- A min_vruntime value can be maintained and can keep the minimum virtual runtime among the vruntimes in the runqueue (at each timer tick, for example, it can be updated). But it must be *monotonically* increasing. It can not decrease. Hence, for newcoming processes and for processes returning from wait, we need to be careful in updating the min_vruntime of the runqueue (see blow).
- Whenever you update the min_vruntime (rq.min_vruntime in the pseudo-code below), update it as shown in the pseudo-code below, so that min_vruntime is monotonically increasing. Let p be the running process and rq.list is the runqueue rbtree (or linked list).

```
if len(rq.list) > 0: // if there is a ready process
      firstnode = rq.list[0]  // firstnode is the one that has
                                  the smallest   vrtuntime in the tree
       vruntime = min(p.vruntime, firstnode.vruntime)
else:
       vruntime = p.vruntime
rq.min_vruntime = max(rq.min_vruntime, vruntime)
```

- When a process returns from i/o wait, it will have a virtual runtime (vruntime) which is set as follows:
  - *vruntime = maximum (old_vruntime, min_vruntime – targeted_latency).*

  Here, old_vruntime is the old vruntime of the process returning from wait. If the vruntime of the task is smaller than the updated vruntime of the running task minus some amount (see VGRAN below), the running task is preempted. Using "min_vruntime – target_latency" as a lower bound on an awakened task prevents a task that is blocked for a long time from monopolizing the CPU. That means when, for example, a process (p2) returns from io, its vruntime should be at least VGRAN virtual time units behind the vruntime of the currently running process (p1) to trigger an immediate context switch. We consider the priority (load) of the running process to scale MINGRAN to be VGRAN. Use the pseudo-code below:

```
// set p2's vruntime as described above  - it returned from I/O
VGRAN = MINGRAN * NICE0LOAD / p1.load  // p1 is running process
If p2.vruntime  < p1.vruntime -  VGRAN     // p2 is returned from I/O
   Start running p2 (context switch immediately)
else
  continue running p1
```

- Every time a process runs for t ns, its virtual runtime is updated as
  - *vruntime += delta, where delta is:*
    *delta = t * NICE0_LOAD / PROCESS_LOAD.*

  NICE0_LOAD is the weight of the process with priority 20 (See Table 1). It is 1024. PROCESS_LOAD is the weight of the process. It depends on the priority of the process and is found from Table 1. For example, a process with priority value 30 will have its delta = t * 1024 / 110. That means, for a process with a

lower priority, the vruntime is advanced faster (it is charged more for running in cpu).

- When a new task is created, its *vruntime = current min_vruntime – 10 ms* and it will be inserted into the runqueue data structure. It may lead to a scheduling decision.
- Even as the number of runnable processes approaches infinity, each will run for at least 10 millisecond (if they want).

Table 1: Weights corresponding to priority (nice) values. In this project, a process with priority 20 (i.e., a process with priority 0 in Linux) will have a weight of 1024. Priority in this project = Linux priority + 20.

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

You will also write a program to generate workload. It will be called as **loadgen** and will have the following parameters:

loadgen <N> <avg_start_time> <avg_num_bursts> <avg_cpu_len> <avg_io_len> <wordloadfile>

<N> is the number of processes. The mean cpu burst length for a process is <avg_cpu_len> and the mean io burst length for a process is <avg_io_len>. The cpu and io burst lengths are exponentially distributed. The number of cpu bursts that a process has is also exponentially distributed with a mean of <avg_num_bursts>. The number of io bursts for a process is one less than the number of cpu burts. The average time that passes before a process is started is <avg_start_time>. That is also exponentially distributed. Time stats at 0. The <workdloadfile> parameter is the output filename into which the information will be printed.

Experiments:

You will do a lot of experiments with your program. For example, you can measure the response time with respect to priority for some certain load. You can also measure the effect of load on response time. You can design and do more experiments. You will plot the results and interpret them. You will put all these into a report (in PDF form finally).

References:

1. Linux Journal, Completely Fair Scheduler. http://www.linuxjournal.com/magazine/completely-fair-scheduler
2. Shichao's Notes, Chapter 4: Process Scheduling. https://notes.shichao.io/lkd/ch4/

3. A complete guide to Linux Scheduling. Nikita Ishkov. https://tampub.uta.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf
4. CFS and Its Tuning. http://fizyka.umk.pl/~jkob/prace-mag/cfs-tuning.pdf
5. Inside Linux CFS Scheduler. https://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

Clarifications:

- Ignore context switching overhead.
- Assume a *timer tick* is happening with every 10 ms. Hence, at least every 10 ms we need to check if a context switch has to happen (running the scheduling routine).
- Assume all process arrival times and cpu and i/o burst times are a multiple of ms. That means the units in the workloadfile are all ms.
- Assume the i/o burst for a process is exact no matter how many processes are there. That means, if an i/o burst for a process is 70 ms, for example, the process will wait that much time in a wait queue and then will come back to the runqueue again. The waiting in the wait queue will not be affected from scheduling and from the number of processes in the system either in ready state or waiting state.
- When the rate parameter is given for an exponential distribution, you can generate random values for that distribution. This is called random variate generation. You can search and learn that topic. rate parameter for exponential distribution is = 1/mean. For, example, you are given avg_cpu_burst_length. This is the mean. Then you can obtain the rate and generate random variates for exponential distribution.
- You can generate (in loadgen) priority values also in a random manner between 0 and 39 (uniform random). It is better if you can generate random priorities with Gaussian distribution around mean 20 (NICE0).