Adding another scheduling policy in Linux

—

Group Members:

N.V. Navaneeth
15115086

Chirayu Asati
15115040

Deepak Verma
15115045

Rishabh Jain
15111033

Overview

The short-term process scheduler is one of the most important components of an operating system. The algorithms used for scheduling can determine how much of the CPU is utilized, and also how interactive the OS is. Hence choosing a scheduling policy that suits the application needs of the OS is crucial. In this project, we are to add another scheduling policy to linux to schedule background tasks.

Goals

1. Study the linux scheduler in depth to understand how it chooses the processes to be run
2. Add another scheduling policy for background scheduling.
3. Evaluate and comment on the performance of the newly implemented policy

Background scheduling means that the processes should run only when there are no tasks in the other scheduling classes to be run.

Linux scheduler

The linux scheduler has had some major changes over the years, including major rewrites of the code. One of such versions was the 2.6.23, where the scheduler was changed from the previous O(1) scheduler to a new Completely Fair Scheduler (CFS). In this version, they also introduced the SCHED_IDLE scheduling policy which almost (it's not exactly a background scheduler though) does what the SCHED_BACKGROUND is meant to do. Also prior to 2.6.23, the scheduler code was mainly in one file: /kernel/sched.c, and this resembles the hints given for the project. From 2.6.23, it was rewritten to multiple files, abstracting away the implementation details of each scheduling class. Due to the above reasons, we decided to go work with a kernel version prior to 2.6.23. We decided on using the 2.6.20 version which matched the kernel version of ubuntu 7.04, since we could just install ubuntu 7.04 in a VM, download its source and make changes to it to avoid any major compatibility issues. This remainder of this section describes the scheduler in **linux kernel version 2.6.20.**

MAIN SCHEDULER:

The primary data structure used a priority array, corresponding to each index of the array is a list of all the process in that priority. When a process is to be scheduled, the non-empty queue with the highest priority is identified, and the process at the head of the queue is scheduled next. The scheduler actually maintains two priority arrays: **active** and **expired.** Tasks are scheduled from the active array. When the time slice of the current process is used up, the process is push into the expired array. Thus when the active array becomes empty, all that is to be done is to swap the active and expired arrays. This swapping is done in constant time and hence scheduling is O(1).

The number of levels of priority is determined by the MAX_PRIO constant. This is 140 in our case. Out of this levels 0 to (MAX_RT_PRIO - 1) are reserved for only real time tasks. MAX_RT_PRIO is 100 in our case. The remaining level, i.e MAX_RT_PRIO to (MAX_PRIO-1) are for normal tasks.

Details on the different scheduling policies are given below.

SCHED_FIFO:

This scheduling policy handles only real time tasks i.e. priority values must be from 0-99. When SCHED_FIFO tasks are scheduled, they have no assigned time slices and are allowed to execute till completion.

SCHED_RR:

This policy also handles only real time tasks. Unlike SCHED_FIFO, they schedule tasks in a round robin fashion, i.e. tasks when scheduled, are given a time slice and when it expires, they are put back at the end of the (expired) queue.

SCHED_OTHER:

This is the default scheduling policy for any task, and handles normal tasks. They can take priorities from 100-139 depending on the nice value (discussed later). These tasks are also scheduled in round robin fashion.

SCHED_BATCH:

This policy is similar to SCHED_BATCH, but they are assumed to be non-interactive processes and never get any interactivity boost in priority (discussed later).

Nice value:

It is a value in the range -20 to +19 associated with SCHED_OTHER and SCHED_BATCH tasks which determines their priority (SCHED_RR and SCHED_FIFO tasks are unaffected by their nice values). -20 to +19 range directly maps to the priority range

100-139. The default nice value is 0 (i.e priority 120). The nice values of a process can be changed using the *nice* or *renice* terminal commands.

Interactivity boost:

The linux scheduler is designed so that interactive tasks get more priority than cpu hogs. The scheduler assumes that tasks that spend time sleeping are blocked on I/O and hence are interactive. So, based on the amount of time a task spends sleeping, a process might get a boost in priority. SCHED_OTHER processes can get a priority level boost of -5 to +5. SCHED_BATCH processes on the other hand cannot be boosted to higher priorities, hence boost is from 0 to +5.

Adding SCHED_BACKGROUND

This section describes our approach for implementing the SCHED_BACKGROUND policy.

FIrst step was to define SCHED_BACKGROUND as a scheduling policy in sched.h

```
30  /*
 1   * Scheduling policies
 2   */
 3  #define SCHED_NORMAL     0
 4  #define SCHED_FIFO    1
 5  #define SCHED_RR    2
 6  #define SCHED_BATCH   3
 7  #define SCHED_BACKGROUND 4
```

Taks under this policy should be run only when there are no other tasks to be run. Hence what we need to do is add another priority level that has the lowest priority, and put all SCHED_BACKGROUND tasks in this level.

Adding another priority level in sched.h

```
 1
528  #define MAX_PRIO    (MAX_RT_PRIO + 41) // Added one more priority level.
 1
```

Next step was to locate the use of MAX_PRIO and adjust for its increase in value, if needed. One of the locations was where the task_struct was initialized, we still want all tasks to default to a priority of 120 (or nice level 0). Hence these statements were changed from MAX_PRIO - 20 to MAX_PRIO - 21.

Fixing init_task.h

```
102   .prio    = MAX_PRIO-21,
  1   .static_prio  = MAX_PRIO-21,
  2   .normal_prio  = MAX_PRIO-21,
  3   .policy    = SCHED_NORMAL,
```

Another location was where the task priority was recalculated based on its interactivity. We do not want SCHED_OTHER or SCHED_OTHER tasks to get shifted to the newly added priority level when their priorities are decreased.

Fixing sched.c : __normal_prio()

```
  1   // Adjust for change in MAX_PRIO.
742   if (prio > MAX_PRIO-2)
  1     prio = MAX_PRIO-2;
```

Next, whenever a process is set to SCHED_BACKGROUND, we need to put it at the lowest priority level, that is set static_prio to the lowest priority.

Setting SCHED_BACKGROUND tasks to lowest priority in sched.c: __setscheduler()

```
  3   /*
  2    * SCHED_BACKGROUND tasks have the minimum priority possible.
  1    */
4187  if (policy == SCHED_BACKGROUND) {
  1     p->static_prio = MAX_PRIO - 1;
  2   }
```

Next, we needed to make sure SCHED_BACKGROUND tasks did not get boosted to better priority values. This can be done by returning the current priority itself at invocation.

Setting sleep time to 0 in sched.c: recalc_task_prio()

```
  1
882   // Prevent SCHED_BACKGROUND tasks from updating priority.
  1   if (background_task(p)){
  2     // Return the current priority itself.
  3     return p->prio;
  4   }
```

Other changes minor changes were also made, like in sys_sched_get_priority_max() and sys_sched_get_priority_min().

How to setup the project.

Steps 1-3 are not necessary if you already have a VM running

1. Install QEMU to create a virtual machine (and virt-manager to use it via a convenient GUI)

   ```
   sudo apt-get install qemu-kvm qemu virt-manager virt-viewer libvirt-bin
   ```

2. Get the ISO image for Ubuntu 7.04 from

   http://old-releases.ubuntu.com/releases/7.04/

3. Create a VM, boot from the above ISO image and install Ubuntu once booted.

4. Download the kernel source code for version 2.6.20 from

   https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/

5. In the source code, copy the the modified files to appropriate locations:

   init_task.h -> include/linux/

   sched.h -> include/linux/

   sched.c -> kernel/

6. Compile the kernel: this will take a long time.

   1. Copy the config file of the running kernel to the source code directory.
      `$ cp -v /boot/config-$(uname -r) .config`
   2. Configure the kernel
      `$ make menuconfig`
   3. Compile the kernel image
      `$ make -j 4`
   4. Install kernel modules
      `$ sudo make modules_install`
   5. Install the kernel
      `$ sudo make install`
   6. Update grub config
      `$ sudo update-initramfs -c -k 2.6.20`
      `$ sudo update-grub`

7. Reboot and press ‘esc’ to enter grub menu. Select the new kernel.

8. Now we are running on the new kernel that include SCHED_BACKGROUND policy.

9. We will be changing policy from within C/C++ files using the <sched.h> library. SCHED_BACKGROUND is not yet defined in this library, so just go to /usr/include/bits/sched.h

    After the line: #define SCHED_BATCH 3,

    Add another line: #define SCHED_BACKGROUND 4.

10. Now all the executables/shell scripts in the folder ‘test_cases’ can be run.

Results and Analysis.

All the times are obtained using the gettimeofday() function. The counters used in this case count from 0 to 10 billion.

1. Running counter as a normal process:

    Time : 31.155436s

2. Running counter as SCHED_BACKGROUND

    Time : 31.880926s

    Takes about the same time as a normal process, the system was barely on any other load, hence this result should be expected as the counter will be the only process that is running, hence irrespective of the priority, its should take nearly the same amount of time.

3. Running two counters simultaneously as normal processes.

    TIme 1: 62.057757s

    TIme 2: 62.077550s

    Both the counters will have the same priority and will run in round robin fashion in the same priority queue. Hence, both the processes complete nearly at the same time.

4. Running two counters, one as normal process, and the other as SCHED_BACKGROUND

    Time normal: 33.407460s

    Time background: 64.172465s

The normal process will have a higher priority than the background one, and hence, the normal task finishes first, and the background task runs only after the normal task is finished.

5. Running the counter as SCHED_BACKGROUND while compiling the kernel.

   Time : 1502.940226s ~ 25mins

   The background counter only runs when the compile process is finished, and hence this huge run time.

6. Running two counters simultaneously, one as SCHED_BACKGROUND and other as a normal process of lowest priority (nice = +19)

   Time background : 65.555264s

   Time normal (nice +19) : 46.217055s

   The background process is still only run after the normal process, but here the nice +19 process takes more time than the nice +0 process as its priority might be pushed behind other system processes that are running regularly.

7. Running two counters simultaneously, one as SCHED_BACKGROUND and other as a normal process of lowest priority (nice = -20)

   Time background : 62.318247s

   TIme normal(nice -20) : 31.339836s

   Similar results as that of case 4. The background process only starts after the normal process is completed.

For the file write testing, we individually write 300 million characters to a single file.

8. Writing to a large file as a normal process

   Time : 16.595614 s

9. Writing to a large file as SCHED_BACKGROUND

   TIme: 17.936297 s

   Not much different from a normal process, the increased time might be due to other system processes running at higher priority.

10. Writing to two large file simultaneously as normal processes.

    Time 1: 33.971829s

    Time 2: 34.778736s

    Similar to case 3, both the processes are at the same priority and hence round robin till they complete, hence completing next to eachother.

gettimeofday() vs getrusage():

gettimeofday() function returns the epoch time at that instant. When we calculate the time using this function, we get the actual time in real world, that the process takes to execute.

getrusage() function returns the total cpu time that the process has used so far. This is split into time executed in kernel mode (ru_stime) and time executed in user mode (ru_utime). For the counters, we just need to measure the ru_utime as it is just counting and does not require kernel mode. When measuring the time of counters using getrusage(), the time obtained was, for all cases, about 8.7 seconds . This is expected, as all the counters, irrespective of the scheduling policy to have the same underlying code, and should take similar amount of CPU time to execute.

References.

- https://www.cyberciti.biz/tips/compiling-linux-kernel-26.html
- https://pubs.opengroup.org/onlinepubs/7908799/xsh/sched.h.html
- http://www.informit.com/articles/article.aspx?p=101760
- https://www.tecmint.com/set-linux-process-priority-using-nice-and-renice-commands/