Operating Systems - Project 1

Wednesday, 06.02.2019

Team members

- *Harshit Maurya (17114037)*
- *Hritvi Bhandari (17114039)*
- *Mansi Agarwal (17114048)*
- *Abhijeet Shakya (17114002)*
- *Gaurav Dewat (17114032)*
- *Pulkit Jeph (17114060)*
- *Rahul Gome (17114062)*
- *Atul Sinha (17114016)*

Implementing a new scheduling policy for Linux

Introduction

Scheduling policies

The Linux scheduler is a priority based scheduler that schedules tasks based upon their static and dynamic priorities. The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next. Each thread has an associated scheduling policy and a static scheduling priority denoted by sched_priority. The scheduler makes its decisions based on one of the scheduling policy and static priority of all threads on System. For threads scheduled under one of the normal scheduling **(SCHED_OTHER, SCHED_IDLE, SCHED_BATCH)**, sched_priority is not used in scheduling decisions.

Processes scheduled under one of the real-time policies **(SCHED_FIFO, SCHED_RR)** have a sched_priority value in the range 1 (low) to 99 (high). Conceptually, the scheduler maintains a list of runnable threads each possible sched_priority value. In order to determine which thread runs next, the scheduler looks for the nonempty list with the highest static priority and selects the thread at the head of this list.

A thread's scheduling policy determines where it will be into the list of threads with equal static priority and how it will move inside this list.

All scheduling is preemptive: if a thread with a higher static priority becomes ready to run, the currently running thread will be preempted and returned to the wait list for its static priority level. The scheduling policy determines the ordering only within the list of runnable threads with equal static priority.

SCHED_FIFO:
SCHED_FIFO is a simple scheduling policy designed for special time-critical applications. It uses the First In-First Out scheduling algorithm. The algorithm is without time slicing. A SCHED_FIFO thread runs until either it is blocked by an I/O request, it is preempted by a higher priority thread, or it calls sched_yield.

SCHED_BATCH:
SCHED_BATCH is a scheduling policy designed for CPU-intensive tasks. It can be used only at static priority 0. This policy is similar to SCHED_OTHER in that it schedules the thread according to its dynamic priority (based on the nice value). The difference is that this policy will cause the scheduler to always assume that the thread is CPU-intensive. Consequently, the scheduler will apply a small scheduling penalty with respect to wakeup behavior, so that this thread is mildly disfavored in scheduling decisions. This policy is useful for

workloads that are noninteractive but do not want to lower their nice value, and for workloads that want a deterministic scheduling policy without interactivity causing extra preemptions (between the workload's tasks).

SCHED_IDLE:
Scheduling policy intended for very low prioritized tasks. The process nice value has no influence for this policy.

SCHED_OTHER:
Default Linux time-sharing scheduling policy used by the majority of processes. The thread to run is chosen from the static priority:0 list, based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice value (see below) and is increased for each time quantum the thread is ready to run but denied to run by the scheduler. This ensures fair progress among all SCHED_OTHER threads.

SCHED_RR:
Similar to SCHED_FIFO, but uses the Round Robin scheduling algorithm where each thread is allowed to run only for a maximum time quantum. If a SCHED_RR thread has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A SCHED_RR thread that has been preempted by a higher priority thread and subsequently resumes execution as a running thread will complete the unexpired portion of its round-robin time quantum. The length of the time quantum can be retrieved using sched_rr_get_interval.

Description of Code Design

The project is concerned with adding a new scheduling process, namely SCHED_BACKGROUND which will be designed to support processes that will run when the system is not occupied.

We will be working on Linux-2.6.24 which is the default kernel version in Ubuntu-8.04, as for higher versions, the encapsulation is increased due to which the complexity of scheduler code is increased. In higher versions, there is no sched.c File, instead there is a sched/ directory which consists of many different parts of the old scheduler.

We would be mainly changing the **sched.c** and **sched.h** as they are responsible for handling the scheduler function in the kernel.

An auxiliary file **chrt.c** will also be modified as it handles all the system calls to the scheduler(modified file is named as chrt_bg.c).

So the primarily modified files can be listed as:

1. /include/linux/sched.h
2. /kernel/sched.c
3. /usr/bin/chrt

A policy named SCHED_IDLE already handles the background tasks in Linux.

So, we will implement a new policy SCHED_BACKGROUND to handle background tasks with a lower priority than SCHED_IDLE. The new policy will handle tasks only if the queues of the other policies are empty.

We define this new policy in sched.h and make the required changes in sched.c. To support making system calls from the terminal, we also add this policy in chrt.c, generate its object file and then replace it with the default system file.

There are mainly two scheduling classes in linux-2.6.24 :

- Real time scheduled class (rt_sched_class)(sched_rt.c) (for real time processes)
- Fair Scheduled Class (fair_sched_class)(sched_fair.c)(for fair scheduling)

The SCHED_BACKGROUND policy needs to handle processes in its queue in a similar manner to SCHED_OTHER which follows fair scheduling. So it also follows fair scheduling and its operating class is sched_fair.c.

We have also modified the sys_sched_get_priority_max() and sys_sched_get_priority_min() functions in order to return 0 when policy is set to SCHED_BACKGROUND (policy=6 as set by us).

Steps to Compile Kernel

First of all, install ubuntu 8.04 64-bit desktop edition in a virtual machine (Virtual Box) so that any changes to the system kernel leave your own system unharmed. The VM also serves as a customisable hardware for benchmarking tests.

1. To start, we need to figure out what version of the kernel we are currently running.

   We'll use the uname command for that :-

```
$ uname -r
2.6.24-generic
```

2. Since we are working on an old version of ubuntu, the default repository sources will not be available. Run this command to switch to sources intended for old distros.

```
sudo sed -i -re
's/([a-z]{2}\.)?archive.ubuntu.com|security.ubuntu.com/old-releases.ubuntu.c
om/g' /etc/apt/sources.list
```

3. Now we need to Install the Linux source for our kernel. We also need to install the curses library and some other tools to help us compile

```
$ sudo apt-get install kernel-package libncurses5-dev fakeroot
```

4. Download the linux 2.6.4 source package from mirror

```
$ wget
https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/linux-2.6.24.ta
r.gz

$ tar -xzf linux-2.6.24.tar.gz
$ cd linux-2.6.24/
```

5. To make things easier, we will put ourselves in root mode by using sudo to open a new shell. There are other ways to do this, but we will do by this way

```
$ sudo /bin/bash
```

6. Make the required changes to implement the new background scheduler in the sched.c , sched.h files and replace them in their original directories.
7. Also change the chrt.c accordingly ,compile it and paste the object file chrt in usr/bin

```
$ gcc chrt_bg.cc  -o /usr/bin/chrt
```

8. Make a copy of existing kernel configuration to use for the custom compile process

```
$ cp /boot/config-2.6.24-generic /usr/src/linux/.config
```

9. First we will do a make clean, just to make sure everything is ready for the compile

```
$ make-kpkg clean
```

10. Next we will actually compile the kernel. This will take a "LONG TIME" maybe 40-50 mins.

```
$ fakeroot make-kpkg --initrd --append-to-version=-modified
kernel_image kernel_headers
```

11. This process will create two .deb files in ../ (one level up source directory) that contain the kernel
12. Please note that when running these next commands, this will set the new kernel as the new default kernel. This could break things! If the machine doesn't boot, hit Esc at the GRUB loading menu, and select the old kernel. Then disable the kernel in /boot/grub/menu.lst or try and compile again.

```
$ dpkg -i linux-image-2.6.24-custom_2.6.24-custom-10.00.Custom_i386.deb
$ dpkg -i linux-headers-2.6.24-custom_2.6.24-custom-10.00.Custom_i386.deb
```

13. Now reboot the machine. If everything works, it should be running our new custom kernel. check this by using uname.

```
$ uname -r
2.6.24-modified
```

14. To change the policy of a process to SCHED_BACKGROUND policy in its runtime, command used :-

```
$ chrt --background -p 0 <pid>
```

Evaluation

We tested our scheduling policy as specified in the assignment.

The eight test test cases provided in the problem statement have been written into simple bash scripts to automate the process. Navigate into the test_cases directory provided within the project directory.

```
$ cd test_cases
```

The test cases can be run simply via command line:

```
$ ./[1-8].sh
```

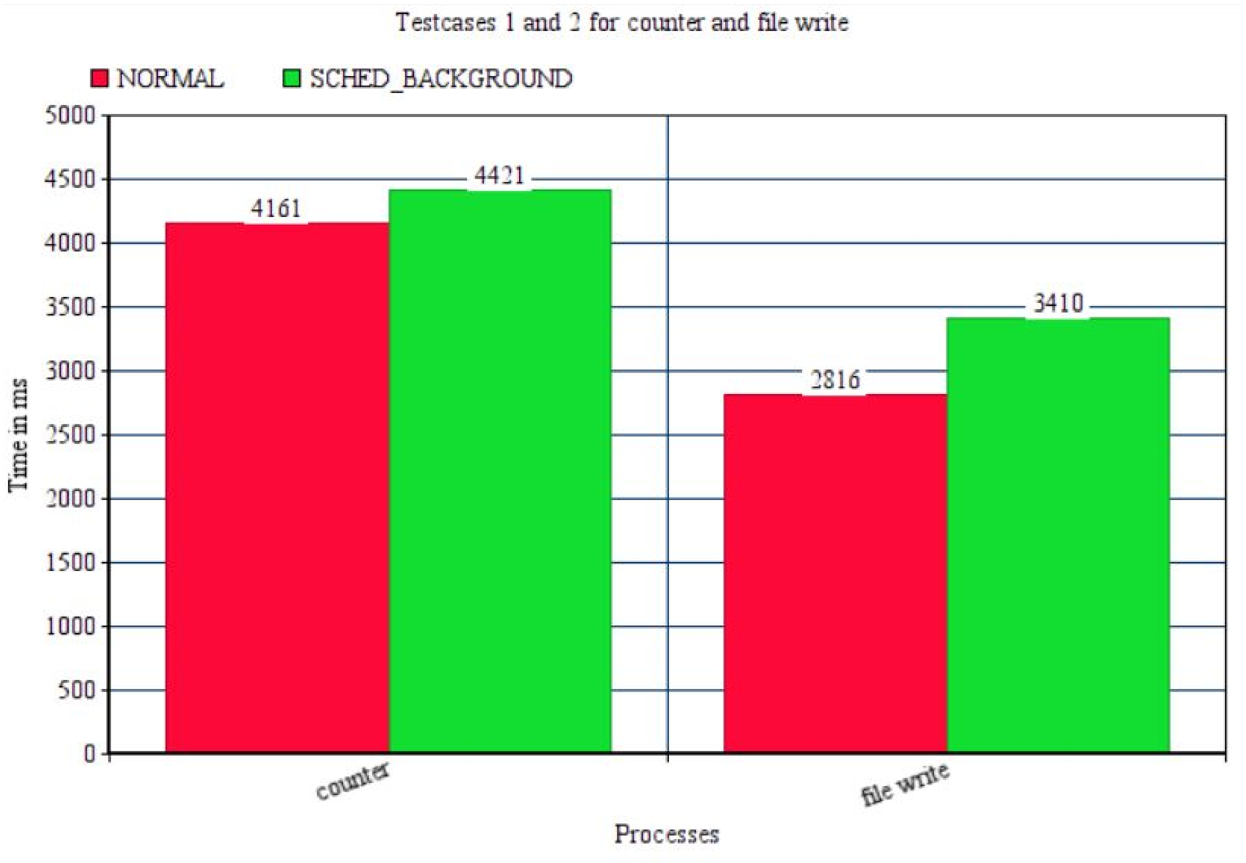
Following are the brief descriptions of the eight conditions under which the performance was evaluated:

1. Ran counter as normal process. It took 4161 ms to complete execution.

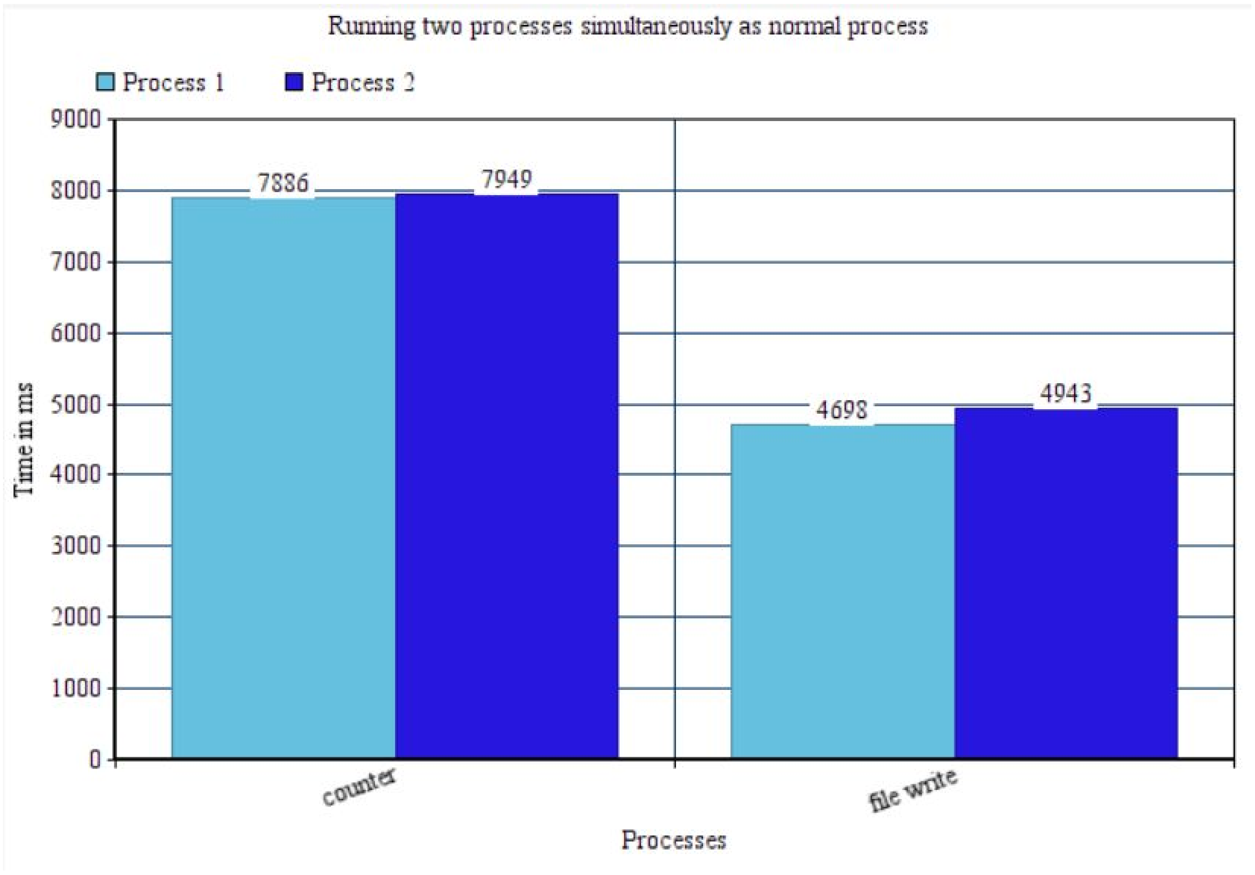
   This is a sample output on running counter:

   ```
   Total user time in milliseconds: 3048.190000
   Total system time in milliseconds: 400.025000
   Wallclock time in milliseconds: 4161.412842
   ```

2. Ran counter as SCHED_BACKGROUND process. It took 4421ms to complete execution.
   a. Used `chrt --background -p 0 $$` to set the process scheduling policy as SCHED_BACKGROUND.



3. Ran two counters simultaneously with *normal* scheduling policy. The two processed took 7886 ms and 7949 ms respectively for completion. We observed that the wall clock time was approximately double of that of the first condition. The two processes were scheduled by the Completely Fair Scheduler (CFS).



4. Ran two counters simultaneously, one scheduled normally (process A) while the other by SCHED_BACKGROUND policy (process B). Process A and B took 4173 ms and 4222 ms respectively for execution. Process B, being scheduled as a background process, is given lesser priority therefore taking more time for execution.

5. Ran the counter along with compilation of kernel via `make-kpkg`. The execution times for the counter (scheduled by SCHED_BACKGROUND policy) are as follows:

```
total user time in milliseconds: 3068.191000
total system time in milliseconds: 44.002000
Wallclock time in milliseconds: 75484.290039
```

6. Ran two counters, one at lowest priority (set using `nice -n 19`) and the other by SCHED_BACKGROUND policy. The other process, being a background process, had lower priority than even the first process. The wallclock time taken by the

background process for completion was 7875 ms (significantly larger than that in condition 4).

7. Ran two counters, one at highest priority (set using `nice -n -20`) and the other by SCHED_BACKGROUND policy. In this case we observed that both the processes had approximately the same wall clock time that is 4132ms and 4149ms for the counter executed using nice and counter executed by SCHED_BACKGROUND respectively.

   But the actual execution of background process finished after around 8200ms. This discrepancy occurs because on setting nice value = -20 for normal process, the background process does not even get to initiate and its timer does not initialise until high priority normal process has ended.

8. In this, we repeated steps 1-3 for read/write to a large file. The output observed was as follows:

```
1. Run disk write as a normal process. Record how long it takes to
write.
Total user time in milliseconds: 12.000000
Total system time in milliseconds: 1604.100000
Wallclock time in milliseconds: 3358.211914
^^ Program: write_to_disk || scheduling: normal || pid: 29849


2. Run disk write as a scheduled process. Record how long it takes to
write.
Total user time in milliseconds: 56.003000
Total system time in milliseconds: 1608.100000
Wallclock time in milliseconds: 3038.084961
^^ Program: write_to_disk || scheduling: background || pid: 29851


3. Run disk write simultaneously with another disk write, both as normal
processes.

First normal write queued
Second normal write queued
Total user time in milliseconds: 20.001000
Total system time in milliseconds: 1956.122000
Wallclock time in milliseconds: 4479.694092
```

```
^^ Program: write_to_disk || scheduling: normal || pid: 29871

Total user time in milliseconds: 52.003000
Total system time in milliseconds: 2392.149000
Wallclock time in milliseconds: 5859.659912
^^ Program: write_to_disk || scheduling: normal || pid: 29869
```

getrusage() vs gettimeofday()

The function getrusage() returns resource usage measures. This system call retrieves process statistics from the kernel. It can be used to obtain statistics for the current process by passing RUSAGE_SELF as the first argument. 'rusage' is a struct with many fields, however, here we make use of just 2 fields:

1. ru_utime—A 'struct timeval field' containing the amount of user time that the process has used. User time is CPU time for which the process runs in the user mode.
2. ru_stime—A struct timeval field containing the amount of system time that the process has used. System time is the CPU time for which the process runs in kernel mode

The function gettimeofday() obtains the current time, expressed as seconds and microseconds and stores it in a timeval structure. gettimeofday will report the time that you would see on your wrist-watch, whatever your task is doing.

The difference in the time reported by both the calls is due to the difference in their precision and accuracy. It seems that getrusage uses only the kernel tick (usually 1ms long) and as a consequence can't be more accurate than the ms. Then the getimeofday() function seems to use the most accurate underlying hardware available. As a consequence its accuracy is usually the microsecond (can't be more because of the API) on recent hardware

● ‘nice’ command

Nice is a command in Unix and Linux operating systems that allows for the adjustment of the “Niceness” value of processes. In order to prevent a process from stealing CPU time from high priority processes, we increase the processes niceness value. Adjusting the “niceness” value of processes allows for setting an advised CPU priority that the kernel’s scheduler will use to determine which processes get more or less CPU time. Niceness values range from -20 (the highest priority, lowest niceness) and 19 (the lowest priority, highest niceness).

References:

- https://web.cs.wpi.edu/~claypool/courses/3013-A05/projects/proj1/index.html
- https://askubuntu.com/questions/91815/how-to-install-software-or-upgrade-from-an-old-unsupported-release
- https://www.howtoforge.com/kernel_compilation_ubuntu_p2
- https://www.kernel.org/pub/linux/kernel/v2.6/
- http://man7.org/linux/man-pages/man2/getrusage.2.html
- http://man7.org/linux/man-pages/man2/gettimeofday.2.html
- https://www.howtogeek.com/howto/ubuntu/how-to-customize-your-ubuntu-kernel/
- https://www.youtube.com/watch?v=scfDOof9pww