Fair Share Scheduling in LINUX

Anmol Anubhai (121004) Pooja Shah (121035)
Rahul Patel (121040) Shashwat Sanghavi(121049)

May 4, 2015

1 Introduction to Scheduling

The main goal of fair share scheduling is to choose a scheduling algorithm that can increase the system utilization while not compromising on the wait time and response time of jobs. The process scheduler is in the kernel that controls allocation of the CPU to processes. The scheduler supports the concept of scheduling classes. Each class defines a scheduling policy that is used to schedule processes within the class. The default scheduler, in for example the Solaris Operating System tries to give every process relatively equal access to the available CPUs. However one might want to specify that certain processes be given more resources than others.

2 Linux Scheduler

The Scheduling history reported for Linux is as shown below:

Linux version	**Scheduler**
Linux pre 2.5	Multilevel Feedback Queue
Linux 2.5-2.6.23	O(1) scheduler
Linux post 2.6.23	Completely Fair Scheduler

Figure 1: Linux scheduler history

We have used the Linux kernel 2.6.38 and thus it follows the completely fair scheduling policy. We did not use the earlier kernel versions as they fail to compile without bugs.

2.1 The O(1) Scheduler

- The name was chosen because the scheduler's algorithm required constant time to make a scheduling decision, irrespective of the number of tasks.
- The algorithm used by the O(1) scheduler relies on active and expired arrays of processes to achieve constant scheduling time.

- Each process is given a fixed time quantum, after which it is preempted and moved to the expired array.
- Once all the tasks from the active array have exhausted their time quantum and have been moved to the expired array, an array switch takes place. This switch makes the active array the new empty expired array, while the expired array becomes the active array.

2.1.1 The Key Flaws of this algorithm are:

- The algorithm tries to identify interactive processes by analyzing average sleep time (the amount of time the process spends waiting for input).
- Processes that sleep for long periods of time probably are waiting for user input, so the scheduler assumes they're interactive.
- The scheduler then gives a priority bonus to interactive tasks (for better throughput) while penalizing non-interactive tasks by lowering their priorities. All the calculations to determine the interactivity of tasks are complex and subject to potential miscalculations, causing non-interactive behavior from an interactive process.

2.2 Completely Fair Scheduling (CFS)

- The main goal that CFS tries to achieve is to maintain balance (fairness) in providing processor time to tasks.
- To determine the balance, the CFS maintains the amount of time provided to a given task in virtual runtime.
- The smaller a task's virtual runtime, the higher its need for the processor.
- But rather than maintain the tasks in a run queue, as has been done in prior Linux schedulers, the CFS maintains a time-ordered red-black tree.
- A red-black tree is a tree with a couple of interesting and useful properties. First, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other. Second, operations on the tree occur in O(log n) time (where n is the number of nodes in the tree). This means that you can insert or delete a task quickly and efficiently.
- With tasks (represented by $sched_entity$ objects) stored in the time−ordered red−black tree, tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtime) are stored toward the right side of the tree.
- The scheduler then, to be fair, picks the left-most node of the red-black tree to schedule next to maintain fairness.

2.2.1 Priorities and CFS

- CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute.
- Lower-priority tasks have higher factors of decay, where higher-priority tasks have lower factors of delay.

2.3 Basic Approach Fairshare Scheduling

Fairshare algorithms use two aspects i.e. the share allocated and a dynamic priority formula to calculate a jobs dynamic priority. In addition to more shares, higher priorities can also be given to more important users. If two users with different priority are competing for resources then the resources can be allocated in the following ways:

1. The most important user can have all the resources.
2. The most important user can get more resources according to some proportional distribution.
3. The two users can get the same level of resources.

Now, since we are discussing the word share, let us understand the concept of CPU shares.

2.4 CPU Shares

The term share means the portion of the CPU resources that are allocated to a project. CPU shares are not equivalent to percentages of CPU resources. Shares are used to define the relative importance of workloads in relation to other workloads. When you assign CPU shares to a project, your primary concern is knowing how many shares the project has in comparison with other projects.
Next let us take a look at the Fairshare key Parameters in detail.

2.5 Fairshare Parameters

Fairshare scheduling allows shares users, groups, and classes. The number of shares allocated are based on the usage during windows of time. These shares can be configured at the system level, at the group level, and at the user level. Parameters for the calculation of the dynamic priority which forms another key feature are as follows:

- FS_INTERVAL: Duration of each fairshare window.
- FS_DEPTH: Number of fairshare windows factored into the current fairshare utilization calculation.
- FS_DECAY: Decay factor applied to weighting the contribution of each fairshare window in the past.

2.6 The constituents of FSS

2.6.1 Projects and Users

- Projects are the workload containers in the FSS scheduler.
- Groups of users who are assigned to a project are treated as controllable blocks.
- One can create a project with its own number of shares for an individual user.
- Users can be members of multiple projects that have different numbers of shares assigned.

2.6.2 FSS and Processor Sets

- The FSS scheduler treats processor sets as entirely independent partitions, with each processor set controlled independently with respect to CPU allocations.
- The CPU allocations of projects running on one processor set are not affected by the CPU shares or activity of projects running on another processor set
- The number of shares allocated to a project is system wide. Regardless of which processor set it is running on, each portion of a project is given the same amount of shares.
- When processor sets are used, project CPU allocations are calculated for active projects that run within each processor set.

3 FSS Example

Project A is running only on processor set 1. Project B is running on processor sets 1 and 2. Project C is running on processor sets 1, 2, and 3.

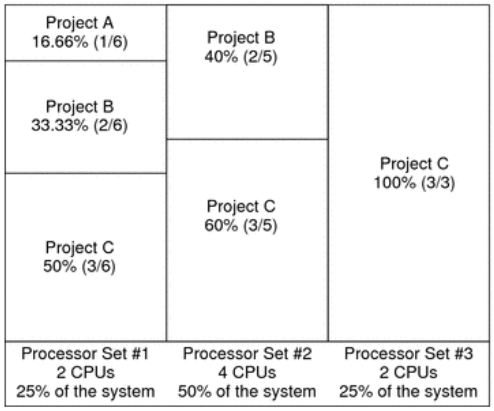

Figure 2: Scenario for FSS example

The total system-wide project CPU allocations on such a system are shown in the following table:

Project	Allocation
Project A	4% = $(1/6 \text{ X } 2/8)_{pset1}$
Project B	28% = $(2/6 \text{ X } 2/8)_{pset1} + (2/5 * 4/8)_{pset2}$
Project C	67% = $(3/6 \text{ X } 2/8)_{pset1} + (3/5 \text{ X } 4/8)_{pset2} + (3/3 \text{ X } 2/8)_{pset3}$

Figure 3: Allocation of resources in FSS

These percentages do not match the corresponding amounts of CPU shares that are given to projects. However, within each processor set, the per-project CPU allocation ratios are proportional to their respective shares.

Project	Allocation
Project A	16.66% = (1/6)
Project B	33.33% = (2/6)
Project C	50% = (3/6)

Figure 4: optimal resource allocation

4 Implementation

4.1 Background

Since the project was aimed towards changing the scheduling algorithm, we were supposed to work with /kernel/sched.c file which is solely responsible for linux 2.6.38 scheduling. Because of changed file structure and nomenclature of kernel from 3.X, we opted to go with kernel 2.6.38.
Whenever scheduler is called a function named $schedule()$ gets called which works as dispatcher. Except this function there are few other fuctions such as $sched\_fork()$ which gets called whenever any new child process is forked. This function is responsible for dividing resources held by parent with the child process. One other function is

important to look for this project which is $finish\_task\_switch()$. This function gets called whenever any process gets into zombie state. There are few other functions like $\_\_task\_cred()$ which provides credential like uid,euid of associated with that process. Beside these functions, some structures like $task\_struct$ are contributing in scheduling. This structure is responsible for keeping track of all details associated with process such as process state, timeslice, priority value etc.

4.2 implemented algorithm

For implementing our algorithm we have used a linked list with following structure of node.

```
struct process_track{
        int user_id;
        int count;
        int timeslice;
        struct process_track *next;
};
```

we can consider this two test scenarios for FSS.

- New process in forked.
- Existing process is terminated.

4.2.1 Block Diagrams

Figure 5 and Figure 6 are the blocked diagram for sharing timeslice between users and processes whenever new process gets forked and whenever any process gets terminated.

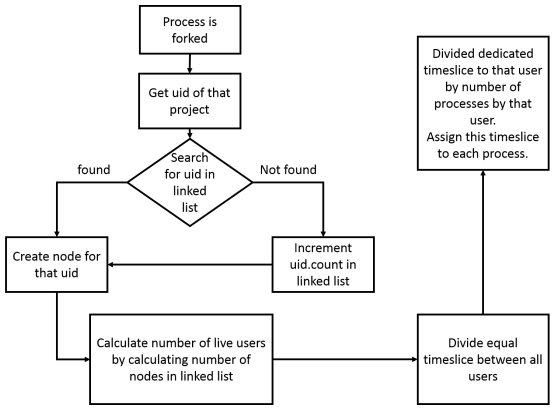

Figure 5: Block diagram to update timeslice division when new process is forked

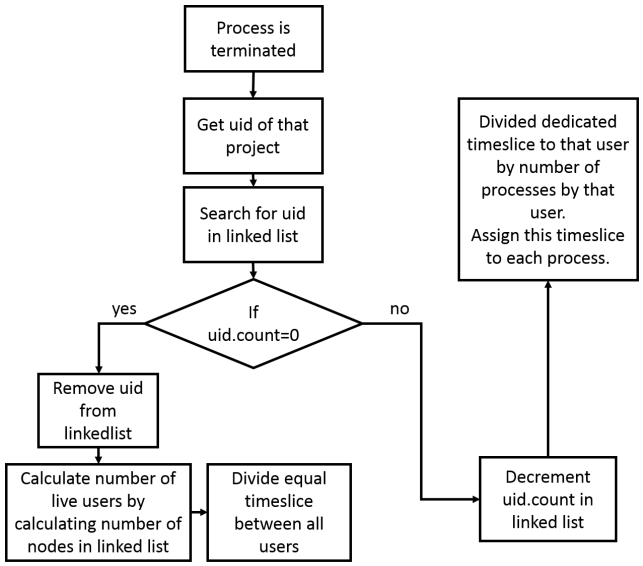

Figure 6: Block diagram to update timeslice division when new process is forked

4.3 Snapshots of execution

Here in the test computer, root is responsible for init process and few other processes like schedule, IO init etc. Apart from that host user is also responsible for running background processes for GUI and other drivers, so test results give many processes running by uid 0 and 1000.
Because of this reason, we will consider only UID ¿ 1000 which are the users that have logged in with terminal.
Here below snapshots shows the division of timeslice depending on the number of users and number of processes held by particular user over uniprocessor machine. Below snapshots are of kernel messages using dmesg. One can incorporate the new scheduler to original scheduler but since linux 2.6.38 is using CFS, we can not see the effect of FSS over uniprocessor machine because FSS is a subpart of CFS.

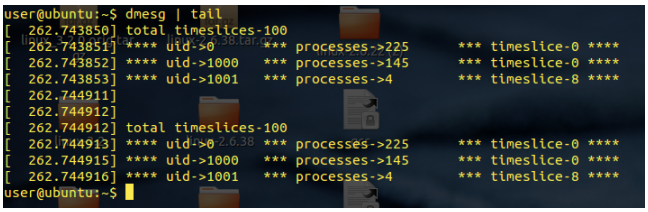

Figure 7: Total timeslice: 100, number of users is 3, each user will get 33 time slices, uid 1001 has 4 processes so it is getting 8 timeslices per process

.

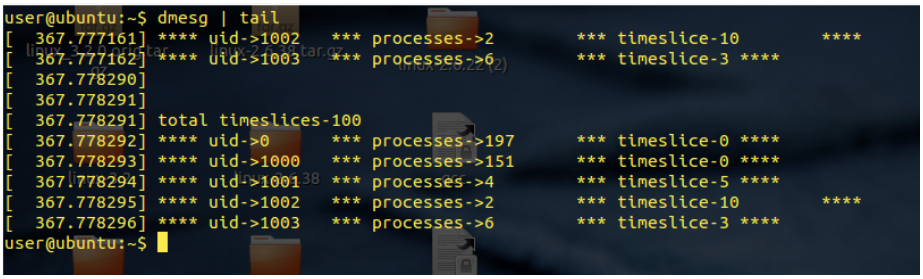

Figure 8: Total timeslice: 100, number of users is 5, each user will get 20 time slices

In the figure above,
timeslice for process by uid 1001=20/5=4
timeslice for process by uid 1002=20/2=10
timeslice for process by uid 1001=20/6=3

5 Conclusion

Our algorithm for FFS checks for new forks. After this if new fork has been created then it creates a new node for the same process. It also increments the process count to keep a track.Then it simply divides the shares equally among all and allocates. One aspect though is that our algorithm's complexity is O(n). Thus presently it cannot be integrated in the kernel as the kernel presently just takes O(1) complexity. Thus we look forward and are keenly interested to work on that in future.

References

[1] Scheduling in Linux `http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/`

[2] Process Scheduling in Linux `http://www.ittc.ku.edu/~kulkarni/teaching/EECS678/projects/scheduling/materials/scheduling.pdf`

[3] Implementing a new real-time scheduling policy for Linux: Part 1 http://www.embedded.com/design/operating-systems/4204929/Real-Time-Linux-Scheduling-Part-1

[4] Background Process Scheduling http://web.cs.wpi.edu/ claypool/courses/3013-A05/projects/proj1/

[5] Linux Kernel Map `http://www.makelinux.net/books/lkd2/ch04lev1sec2`

[6] Modify the Linux Scheduler to limit the CPU usage of a process family http://www.csd.uoc.gr/~hy345/assignments/2013/cs345_front4.pdf

[7] LinSched: The Linux Scheduler Simulator http://www.cs.unc.edu/~jmc/linsched/

[8] Understanding the Linux 2.6.8.1 Process Scheduler http://cs.boisestate.edu/~amit/teaching/597/scheduling.pdf

[9] CFS Scheduler https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

[10] Fair-Share Scheduling http://dasl.mem.drexel.edu/~ducNguyen/courses/cs370-operating-systems/cs370-p4-fair-share-scheduling/

[11] Linux Kernel 2.6.22.19 Scheduler http://dasl.mem.drexel.edu/~ducNguyen/2013/08/15/linux-kernel-2-6-22-19-scheduler/