The Fair Share Scheduler Revisited

The Fair Share Scheduler was first conceptually described by G.J. Henry in 1984. As described by Henry, the Fair Share Scheduler is meant to guarantee a fair distribution of CPU resources not only between processes, but between users and predefined share groups of users. Henry's article omits a specific discussion of the implementation, providing only specification and final results. This paper uses the description provided by Henry to attempt to develop an implementation of the Fair Share Scheduler that can reproduce Henry's results. Given changes in computing architecture since the 1980s, a simulator is substituted to compensate. The simulation translates Henry's description into a functional program, and manages to reproduce the results. However, due the omission of the details of Henry's implementation, although the principle behind the algorithm can be understood and recaptured, Henry's minimal performance overhead cannot be reproduced.

I. Introduction

A typical examination of CPU scheduling algorithms will usually engage with various basic algorithms including first-come-first-serve, shortest job first, and round robin scheduling. However, all of these algorithms only function at a basic per-process level. No prioritization is made on the level of users or group of processes. This means that for any system with multiple users, as all processes are treated equally, a user with a larger share of active processes will naturally have a larger share of resources by virtue of the process scheduler's workings.

If instead, an operating system programmer's interest is in ensuring a reliable share of CPU time on a per-user or per-group basis, they will need a different scheduling solution. One of the earliest and most fundamental examples of such a scheduling algorithm is the Fair Share Scheduler, first described by G.J. Henry[1] and mentioned in the textbook *Operating Systems Essentials*[2], an algorithm which associates processes with so-called fair share groups. These groups are then assigned a certain share of resources. In Henry's example, this is a number of shares out of 100, with 100 representing total system resources. Processes are then allocated such that the group they are in has access to at least its specified share of CPU resources, more if other groups are using less than their assigned share, and less if its own demand is less than that of other groups. Henry claims that Fair Share groups maintain the resource rate they are given, or at least as well as possible given the particular workings of the UNIX scheduler he has adapted, and he claims it does this at only 1% performance overhead.

This particular scheduling algorithm is one of the earliest and most fundamental for a multi-user setting, and so it is important to examine and understand its workings. However, as Henry does not provide a detailed description of his implementation, but merely the specification

[1] Henry, G.J. "The Fair Share Scheduler." *AT&T Bell Lab. Tech. Journal.* 63, no 8 (Oct, 1984): 1845-1857.
[2] Silberschatz et al., *Operating Systems Essentials* (Hoboken, NJ: Wiley, 2014) . 320.

and the parameters by which it runs, it is a little difficult to understand the inner workings. Therefore, this paper will examine the idea behind Henry’s scheduler using a simulation.

II. The Simulation

For this paper, a simulation was written in Java which allows for any number of share groups, as long as they maintain a percentage (out of 100) share allotment between them. In order to bypass issues with the overhead involved in logging and performing administrative tasks, which might affect results, as much as possible was abstracted for the final build of the simulation. The simulation consists of a number of instantiable classes to represent a processor, a process, a ready queue, and a special process to represent idle time, as well as including a logging mechanism and a main testing program. The simulation as a whole contains two threads: the processor thread, and the tester thread which can be configured to add processes either before or after the processor starts operation, with optional time delays to simulate the unpredictable nature of process arrival. Because the simulation uses two threads, it heavily uses Java concurrency library data structures, to preserve thread-safety and prevent concurrency exceptions. Another reason Java library structures were used is to ensure the reliability and accuracy of results.

As for algorithms, the simulation specifically implements Henry’s descriptions of UNIX scheduler priority and the Fair Share Scheduler priority. Any given process within the simulation is allowed to track the CPU time it has received (in simulated milliseconds) and the time it was last running. Processes have a burst between 10 and 300 milliseconds, randomized on instantiation. Henry’s description of UNIX system priority is that it is determined by taking the process’s recent resource usage and dividing it by the real time to supply a rate of resource usage, and comparing this to similar processes. The formula Henry describes is $P = \frac{recent\ resource\ usage}{real\ time}$

In the simulation, this is done by taking the received CPU time (stored as an integer) and dividing it by the time waited since last run, yielding $P = \frac{count(ms)}{(currentTime - lastRunTime)}$, and new processes enter the queue at the lowest value (highest priority).

This functions very well to assure a fair spread of resources among processes, but obviously makes no prioritization at a user or group level. Here is where the fair share scheduling algorithm plays a key role. As defined by Henry, fair share priority includes the regular system priority describes above in its calculations, but adds the group dimension. Groups have been assigned a certain share of resources (in Henry's example, two groups are assigned 75 and 25 shares, respectively) and are meant to be prioritized such that their priority rises if they receive less than their fair share.

The formula for fair share priority is $FSP = P + \frac{recent\ usage\ of\ whole\ group}{real\ time}$, the latter part of which Henry describes as an "exponentially weighted sum … normalized to the allocated resource consumption rate"[3]. In the simulation, this is calculated by taking the sum total of received resources and comparing that to the total CPU busy time multiplied by the allocated resource rate. Transcribing Henry's example using a numerical ranking system where the least is the most important, the user process at the head of the queue will be G_1P_1, followed by G_1P_2 … G_1P_N, which is then followed by G_2P_1 … G_2P_N, and so on through G_NP_N. These groups are reordered every second, along with the processes within each group, but the priority is still determined mathematically at the process level. In the simulation, this is accomplished similarly, where a simple boolean value controls whether to use basic system priority or fair share scheduling priority. To be exact, whereas requesting system priority for a process returns a number corresponding to recent resource usage that can be compared to those of other processes

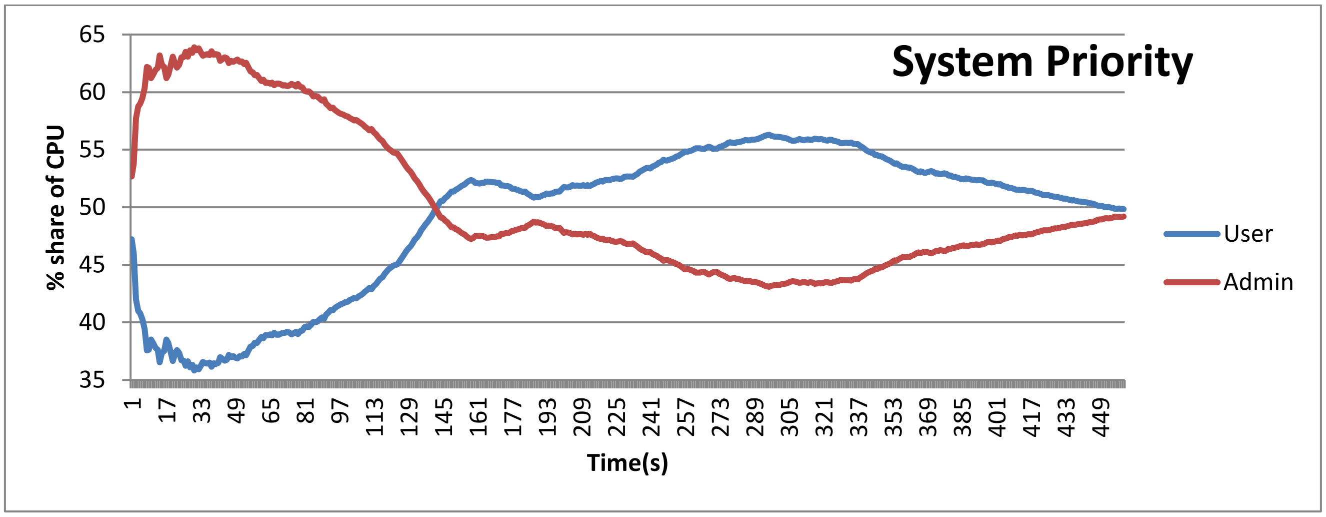
[3] Henry, 1855.

(with less usage giving higher priority), the fair share priority request returns the calculated differential between actual and expected usage rates and adds to that the process priority such that all processes in a given group are sorted amongst each other, then added to the ready queue before the next share group is sorted and added to the queue. This closely matches Henry's description.

Finally, the simulation can be set up with a variety of parameters. The tester program does the work of adding processes to the scheduling queue, with the randomized burst allowing for processes that may have a burst shorter than the supplied quantum, which is handled accordingly. Overall it maintains the priority round-robin architecture Henry describes.

III. Results for a pre-supplied set of processes

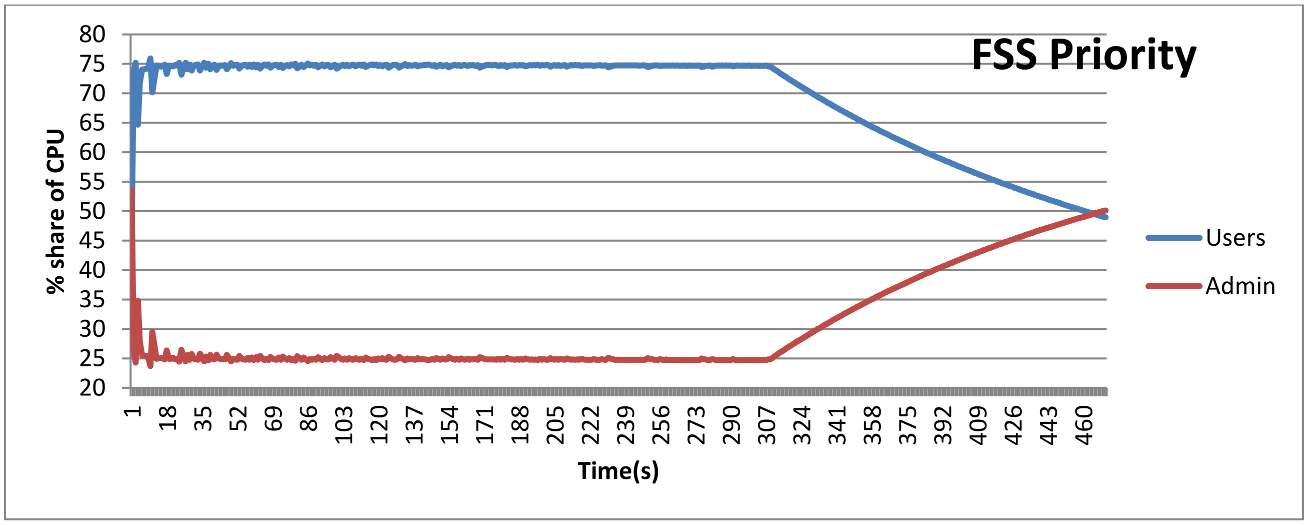
The purest indicator of the algorithm's speed and reliability would be to queue up a number of processes such that there is only thread active at a time to reduce thread blocking overhead. For the following results, two share groups were used: a user group, and an admin group. The user group was assigned a 75% share, and the admin group the remaining 25%. Then, 4500 processes were assigned to the groups such that the user group received one third of the active processes, and the admin group two-thirds. With fair share scheduling off and solely system priority active, the results are as in the following graph:



It is self-evident that the administrative group receives a larger share of CPU resources by simple nature of their larger share of active processes, which is exactly what Henry cites as a key weakness of the standard UNIX scheduler. Only once

If instead of categorized groups these were two users or two groups of users and their associated processes, the share of resources and thus performance any user or group received would be directly proportional to their share of active processes, which is not productive in a multi-user setting.

However, using the same dataset and fair share scheduling, the following results are obtained:

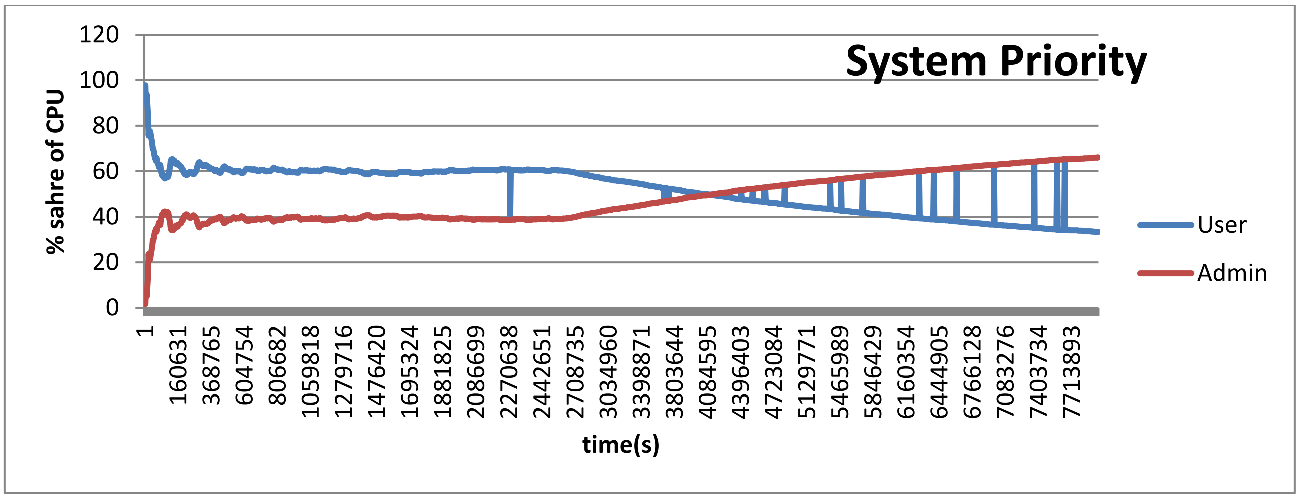


As expected, the fair share scheduling algorithm maintains the assigned share for each group, regardless of number of active processes. The initial imbalance can be attributed to the fact that, just as in Henry's scheduling algorithm, Only after the Users group has emptied its queue of active processes does the admin group's resource usage rise.

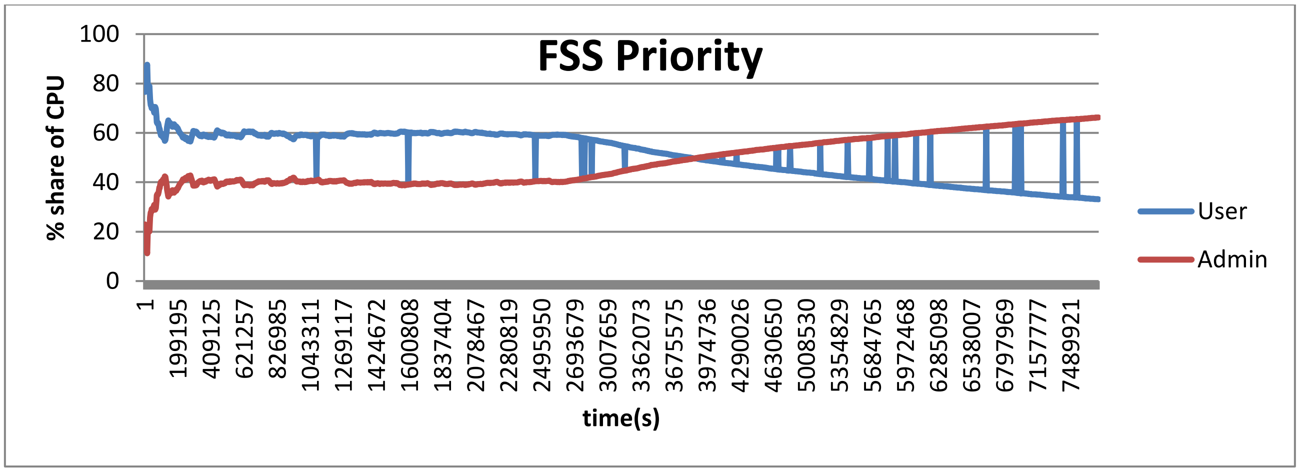
Using these results, we can conclude that the fair share scheduling algorithm does an excellent job of maintaining the specified rate of resources until any given group demands less resources than it is assigned, at which point the algorithm can dynamically reassign those resources. In this regard, the simulation performs exactly to Henry's specification. However, it bears mentioning that while these results were accurate, the fair share algorithm slowed the simulation down immensely in this specific situation.

IV. Results for a dynamically supplied set of processes

Although the previous section demonstrated the basic principle and functionality behind the fair share scheduling system, simply setting the algorithm to work on pre-supplied processes cannot accurately model the performance of the scheduler in a real-time environment, where processes are added to the queue unpredictably. For the following set of results, the same data set (User group: 1500 processes, Admin group: 3000 processes) was used. However, the testing program added processes to the queue at pseudo-random time intervals and in pseudo-random amounts. Functionality was added to randomly add a spike of user processes as well. The results were as follows:



System priority behaved strangely, which may be down the tester's process addition pattern. Overall though, spikes are visible and administrative tasks do end up receiving a large share by the end, presumably as these processes are added later.



FSS Priority similarly behaved strangely, but it maintained a decent resource rate and rises in admin performance were associated with dips in user performance.

V. Conclusion

Having explored the results of a simulation implementing Henry's fair share scheduling system, it is clear that the algorithm has value in an environment where certain groups of processes, whether they are organized by user, by type, or otherwise, must be guaranteed a certain minimum share of CPU resources. Share groups can be reliably guaranteed the minimum share at peak demand, and if demand drops for any particular share group, the resource share for other groups is increased accordingly. In principle, Henry’s algorithm is perfectly sound. However, as Henry provided no details on his implementation, the simulation’s implementation most likely does not resemble Henry’s at all.

Performance in particular can suffer heavily in the simulation. Henry claims an overhead of 1%, but the simulation was incomparably slower, where pre-supplied system priority simulations ran in less than one second, while the fair share version could take a minute or more. The dynamically supplied version is harder to measure due to its dynamic, randomized nature, however, the ratio between the time the process-queueing tester took and the final time of the process is relevant. Results actually indicated these ratios were the same. Evidently the algorithm performs better in a non-heavy load situation.

To conclude, Henry’s fair share algorithm provides excellent functionality, but depending on implementation and situation performance can lag. Had Henry provided more specifics on his implementation, it might have helped those of us reading his article today implement a similar solution. Although there is no shortage of literature on the fair share scheduler, and information on implementations is no doubt abundant, having a foundational document on the algorithm provide a few more details on implementation would have been helpful, but it does not detract from the validity of the algorithm, which is very useful for this specific application.

Bibliography

Henry, G.J. "The Fair Share Scheduler." *AT&T Bell Lab. Tech. Journal.* 63, no 8 (Oct, 1984): 1845-1857.

Silberschatz, Abraham, Peter Baer Galvin and Greg Gagne, *Operating Systems Essentials* (Hoboken, NJ: Wiley, 2014)