these are the shortest processes. On the other hand, the absolute waiting time is uniform, as is to be expected because scheduling is independent of service time. The figures show round robin using a quantum of one time unit. Except for the shortest processes, which execute in less than one quantum, round robin yields a normalized turnaround time of about five for all processes, treating all fairly. Shortest process next performs better than round robin, except for the shortest processes. Shortest remaining time, the preemptive version of SPN, performs better than SPN except for the longest 7% of all processes. We have seen that, among nonpreemptive policies, FCFS favors long processes, and SPN favors short ones. Highest response ratio next is intended to be a compromise between these two effects, and this is indeed confirmed in the figures. Finally, the figure shows feedback scheduling with fixed, uniform quanta in each priority queue. As expected, FB performs quite well for short processes.

## Fair-Share Scheduling

All of the scheduling algorithms discussed so far treat the collection of ready processes as a single pool of processes from which to select the next running process. This pool may be broken down by priority, but is otherwise homogeneous.

However, in a multiuser system, if individual user applications or jobs may be organized as multiple processes (or threads), then there is a structure to the collection of processes that is not recognized by a traditional scheduler. From the user's point of view, the concern is not how a particular process performs but rather how his or her set of processes (which constitute a single application) performs. Thus, it would be attractive to make scheduling decisions on the basis of these process sets. This approach is generally known as fair-share scheduling. Further, the concept can be extended to groups of users, even if each user is represented by a single process. For example, in a time-sharing system, we might wish to consider all of the users from a given department to be members of the same group. Scheduling decisions could then be made that attempt to give each group similar service. Thus, if a large number of people from one department log onto the system, we would like to see response time degradation primarily affect members of that department, rather than users from other departments.

The term *fair share* indicates the philosophy behind such a scheduler. Each user is assigned a weighting of some sort that defines that user's share of system resources as a fraction of the total usage of those resources. In particular, each user is assigned a share of the processor. Such a scheme should operate in a more or less linear fashion, so if user A has twice the weighting of user B, then in the long run, user A should be able to do twice as much work as user B. The objective of a fair-share scheduler is to monitor usage to give fewer resources to users who have had more than their fair share, and more to those who have had less than their fair share.

A number of proposals have been made for fair-share schedulers [HENR84, KAY88, WOOD86]. In this section, we describe the scheme proposed in [HENR84] and implemented on a number of UNIX systems. The scheme is simply referred to as the fair-share scheduler (FSS). FSS considers the execution history of a related group of processes, along with the individual execution history of each process in making scheduling decisions. The system divides the user community into a set of fair-share groups and allocates a fraction of the processor resource to each group. Thus, there might be four groups, each with 25% of the processor usage. In effect, each fair-share group is provided with a virtual system that runs proportionally slower than a full system.

Scheduling is done on the basis of priority, which takes into account the underlying priority of the process, its recent processor usage, and the recent processor usage of the group to which the process belongs. The higher the numerical value of the priority, the lower is the priority. The following formulas apply for process $j$ in group $k$:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k}$$

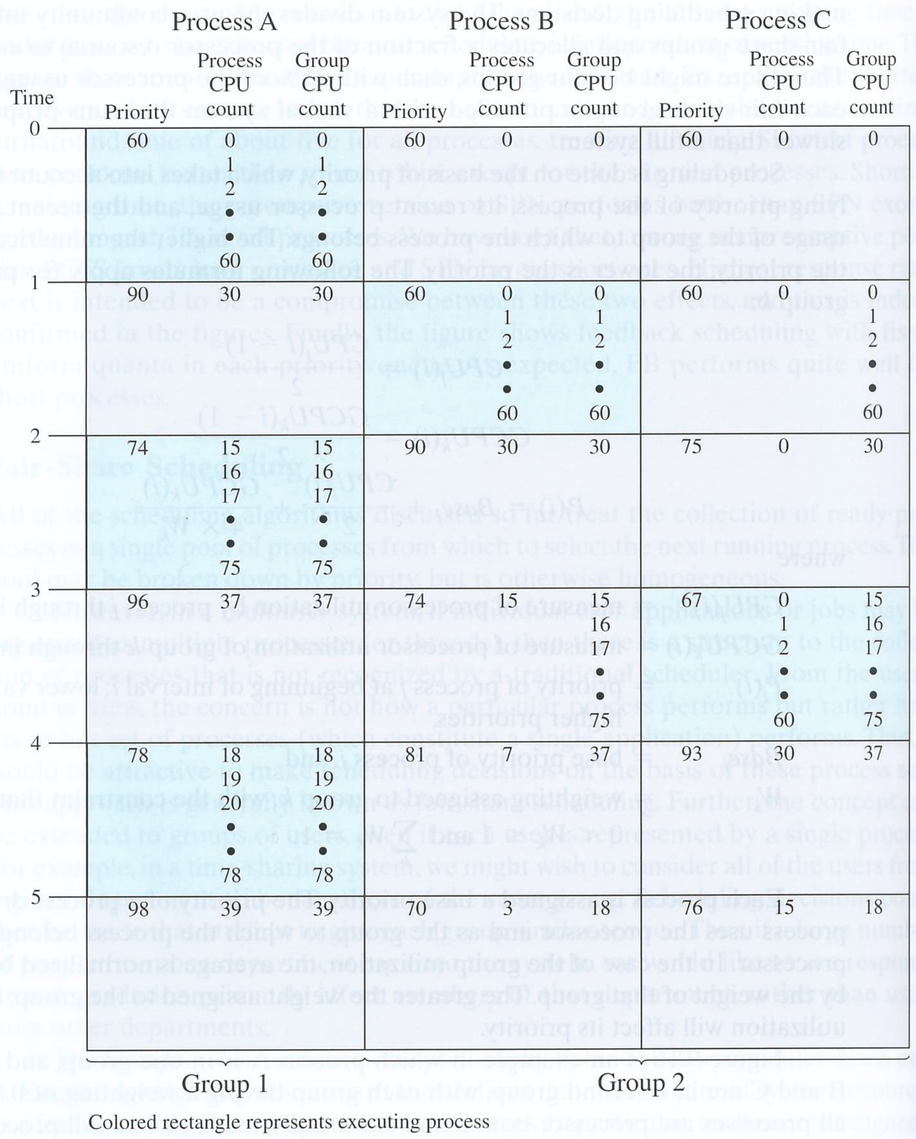
where

| | |
|---|---|
| $CPU_j(i)$ | = measure of processor utilization by process $j$ through interval $i$, |
| $GCPU_k(i)$ | = measure of processor utilization of group $k$ through interval $i$, |
| $P_j(i)$ | = priority of process $j$ at beginning of interval $i$; lower values equal higher priorities, |
| $Base_j$ | = base priority of process $j$, and |
| $W_k$ | = weighting assigned to group $k$, with the constraint that and $0 < W_k \leq 1$ and $\sum_k W_k = 1$. |

Each process is assigned a base priority. The priority of a process drops as the process uses the processor and as the group to which the process belongs uses the processor. In the case of the group utilization, the average is normalized by dividing by the weight of that group. The greater the weight assigned to the group, the less its utilization will affect its priority.

Figure 9.16 is an example in which process A is in one group, and processes B and C are in a second group, with each group having a weighting of 0.5. Assume all processes are processor bound and are usually ready to run. All processes have a base priority of 60. Processor utilization is measured as follows: The processor is interrupted 60 times per second; during each interrupt, the processor usage field of the currently running process is incremented, as is the corresponding group processor field. Once per second, priorities are recalculated.

In the figure, process A is scheduled first. At the end of one second, it is preempted. Processes B and C now have the higher priority, and process B is scheduled. At the end of the second time unit, process A has the highest priority. Note the pattern repeats: The kernel schedules the processes in order: A, B, A, C, A, B, and so on. Thus, 50% of the processor is allocated to process A, which constitutes one group, and 50% to processes B and C, which constitute another group.



| | Process A | | | Process B | | | Process C | | |
|---|---|---|---|---|---|---|---|---|---|
| Time | Priority | Process CPU count | Group CPU count | Priority | Process CPU count | Group CPU count | Priority | Process CPU count | Group CPU count |
| 0 | 60 | 0 1 2 • • 60 | 0 1 2 • • 60 | 60 | 0 | 0 | 60 | 0 | 0 |
| 1 | 90 | 30 | 30 | 60 | 0 1 2 • • 60 | 0 1 2 • • 60 | 60 | 0 | 0 1 2 • • 60 |
| 2 | 74 | 15 16 17 • • 75 | 15 16 17 • • 75 | 90 | 30 | 30 | 75 | 0 | 30 |
| 3 | 96 | 37 | 37 | 74 | 15 | 15 16 17 • • 75 | 67 | 0 1 2 • • 60 | 15 16 17 • • 75 |
| 4 | 78 | 18 19 20 • • 78 | 18 19 20 • • 78 | 81 | 7 | 37 | 93 | 30 | 37 |
| 5 | 98 | 39 | 39 | 70 | 3 | 18 | 76 | 15 | 18 |

**Figure 9.16 Example of Fair-Share Scheduler—Three Processes, Two Groups**

## 9.3 TRADITIONAL UNIX SCHEDULING

In this section, we examine traditional UNIX scheduling, which is used in both SVR3 and 4.3 BSD UNIX. These systems are primarily targeted at the time-sharing interactive environment. The scheduling algorithm is designed to provide good response time for interactive users while ensuring that low-priority background jobs do not starve. Although this algorithm has been replaced in modern