A Fair Share Scheduler.

J. Kay
P. Lauder

University of Sydney

ABSTRACT

CPU schedulers are usually designed to allocate resources fairly among *processes*. In this paper, we describe Share, a scheduler that allocates resources so that *users* get their fair machine share over a long period.

We also describe an hierarchical form of Share that supports sharing, not only between individual users, but also between groups of users. In particular, it supports the sharing of a machine between organisational groups who are independently funded and have contributed a proportion of the machine cost. The hierarchical Share ensures that each group is allocated its defined machine share in the long term.

Keywords: scheduling, charging, resource allocation, fair sharing

1. Introduction

One of the critical requirements of a scheduler is that is *be fair*. Traditionally, this has been interpreted in the context of *processes*, and it has meant that schedulers were designed to share resources fairly between processes. More recently, it has become clear that schedulers need to be fair to *users* rather than just processes. This is reflected in work such as Larmouth (1975, 1978), Newbury (1982), Henry (1984) and Woodside (1986).

The context for developing Share was that of the main teaching machine in a computer science department. We had a large user community, dominated by 1000 undergraduates in several classes as well as a staff of about 100. Our load was almost exclusively interactive and had frequent, extreme peaks when a major assignment was due. On a typical day, there were 60-85 active terminals, mainly engaged in editing, compiling and (occasionally) running small to medium Pascal programs. All this activity was supported by a DEC VAX 11/780 running AUSAM, a local version of Unix that is oriented towards a student environment.

Unix provides a fairly typical scheduler (Bach, 1986). It was inadequate in our environment for a number of reasons:

1. it gave more of the machine to users with more processes, which meant that a user could easily increase their share of the machine simply by creating more processes;
2. it took no account of the long term history of a user's activity so that a student who used the machine heavily for, say, two hours, had the same machine share as a student who had not used the machine for some time;
3. when one class had an assignment due, and all the students in that class wanted to work very hard, everyone, including other students and staff, suffered with poor response, and
4. if someone needed good response for activities like practical examinations or project demonstrations, it was difficult to ensure that they could get that response without denying all other users access to the machine.

The first three of these are manifestations of unfairness in the way that the process scheduler affects users.

On many systems, these problems are partially addressed by the *charging* mechanism (Nielsen 1970, McKell et al. 1979). Typically, charging systems involve allocation of a budget to each user and as users consume resources, they are charged for them. We might call this the *fixed-budget* model, in that each user

has a fixed size budget allotted to them. Then, as they use resources, the budget is reduced and when it is empty, they cannot use the machine at all. A process can get a better machine share if the user who owns it is prepared to pay more for the resources used by that process. The fixed-budget model can be refined so that each user has several budgets, each associated with different resources.

We control allocation of some resources with a fixed-budget charging mechanism. In particular, we use this approach in these cases:

1. for resources like disc space, each user has a limit;[1]
2. resources like printer pages are allocated to each user as a budget that has a tight upper bound and is updated each day;[2]
3. daily connect limits are available to prevent individuals from hogging the machine within a single day;
4. weekly connect limits are sometimes used to prevent students from spending too much time on computing (compared to other subjects) and to encourage students to work steadily on assignments from the first week they are set, right through to the last week (but this commonly has the effect of denying students any machine access near the assignment deadline, even though the machine is lightly loaded and the students would like more machine access to finish and improve their programs).

There are also other utilities to help allocate resources, including a terminal booking program that allows students to reserve a terminal at particular times each week.

All these measures helped control consumption of resources but did not deal with the problems of CPU allocation we described earlier. It was for these that we developed the Share scheduler. Although Share was motivated by our particular problems in a student environment, it equally well serves the needs of any user community that shares a machine which is not run as a commercial bureau operating to make a financial profit. Indeed, Share has been implemented in a research environment with many users from different organisations that have chosen to share the capital and running costs of a machine.[3]

To date, Share has been used exclusively to allocate CPU time, though it takes account of the consumption of all resources as we describe below. We consider that Share is applicable to the scheduling of resources other than CPU, but for simplicity, this paper is written in terms of CPU scheduling.

We describe our work in terms of its design objectives. First we state those objectives and the underlying principles needed for a qualitative understanding of Share. Then we describe the Share scheduler, starting with the user's view of a single-level Share. This is followed by the detailed implementation. From there, we describe the motivation for the hierarchical Share and its implementation. The remainder of the paper is devoted to the evaluation of Share, including a description of the tools available for users and administrators to monitor the performance of Share.

2. Objectives of Share

Many systems link charging and scheduling only in that a user can specify processes for which they are prepared to be charged more in return for being given preference in scheduling. Indeed, Unix offers a mechanism rather like this in *nice*, an attribute of a process that a user can adjust to alter its scheduling priority. In a non-bureau environment, this approach is adequate. However, a more natural approach is to regard each user as having an entitlement to a fair share of the machine, relative to other users. Then the task of the scheduling and charging systems is to ensure that

---

1. If a user exceeds this limit, they are warned at the time and then at each login for three logins. After that, they are not allowed to login.
2. For example, a user might have a printer page bound of ten pages and a daily increment of two pages. This means that they start with a budget of ten pages and if they print, say, three pages in one day, their budget for the rest of the day is seven pages and provided they do no more printing that day, their budget at the beginning of the next day will be nine pages.
3. A Cray X-MP at AT&T Bell Laboratories.

- no individual can get more than their fair share of the machine in the long term, and
- that the machine can be well utilised.

In addition, we extended the notion of fair shares to cover groups of individuals so that Share can allow the sharing of a machine between independent organisations.

To achieve fair sharing and be practicable, the objectives for Share were that at the level of the individual and independent groups which share the machine, it should

1. seem fair
2. be understandable
3. be predictable
4. accommodate special needs: where a user needs excellent response for a short time, it should permit that with minimum disruption to other users;
5. deal well with peak loads;
6. encourage load spreading;
7. give interactive users reasonable response;
8. give some resources to each process so that no process is postponed indefinitely;
9. be very cheap to run.

After we have described Share, we evaluate it in terms of these objectives.

3. User's View of Share

Essentially Share is based on the principle that

- everyone should be scheduled for some resources
- according to their entitlement
- as defined by their *shares*
- and their resource *usage history*.

This is illustrated in Figure 1, which shows that a user can expect poorer response if they have had their fair machine share. This, in turn, gives other users a chance to get their fair share.

To tighten this definition of the user's view of Share, we need to state what we mean by shares and usage.

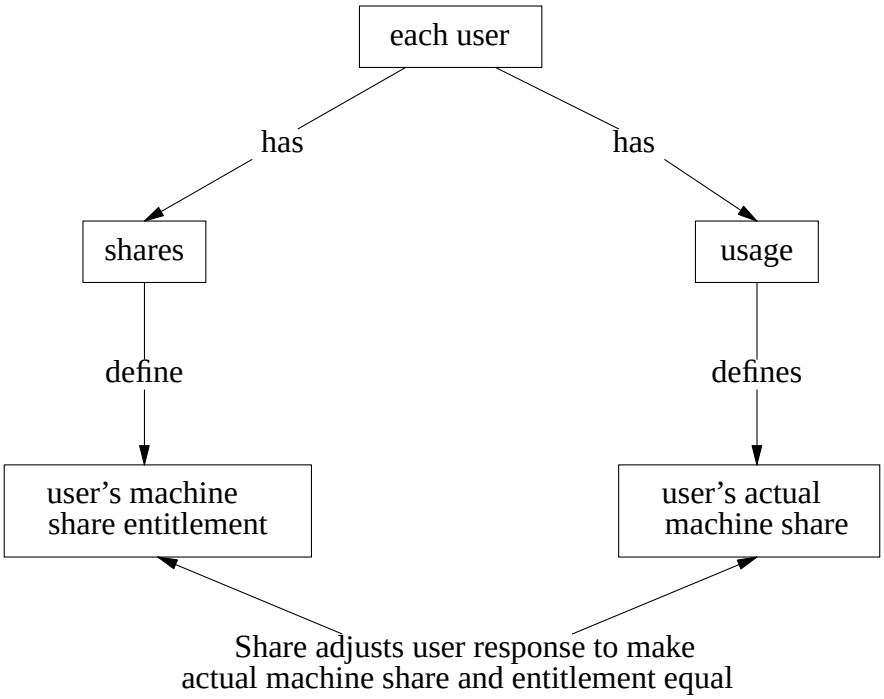
A user's *shares* indicate their entitlement to do work on the machine. The more shares a user has, the greater their entitlement. This should operate in a linear fashion so that if user A has twice as many shares as user B, then in the long term, user A should be able to do twice as much work as user B.

Every user has a *usage*, which is a *decayed* measure of the work that the user has done. The *decay rate* is defined by an administrative decision which determines how long a user's history of resource consumption affects their response. For example, in its first implementation in a student environment, the decay rate was set so that usage had a half-life of three days to encourage students to spread their machine use over the week.

While it is the norm for schedulers to use decayed CPU usages, Share's use of decayed resource usage in *charging* is a departure from traditional approaches. Where a machine is solely for in-house use, the only need for a raw (undecayed) resource consumption tally is in monitoring machine performance and throughput and to observe patterns of user behaviour.

The decayed usage is also *normalised* by the user's shares. One might view this as making the machine relatively cheaper to users with more shares. *In essence, Share attempts to keep the actual machine share defined by normalised usage the same as the machine entitlement defined by shares.* Looked at from the

Figure 1. User's View of Share



user's point of view, Share gives worse response to users who have had more than their fair share so that they cannot work as hard as users who have not had their fair share. *So users see that as their normalised usage increases, their response becomes worse.* (This assumes allowance is made for machine load.) Indeed, we provide a simple command that displays a user's profile which includes their usage and the machine share they can expect.

This approach contrasts strikingly with conventional charging and scheduling systems that schedule processes equally, provided the user who owns them has a non-empty budget. In the fixed-budget model, the users who consume their fair share, by emptying their budgets, get no resources, even if there happen to be plenty available. In the extreme case, there may be no users because everyone who wants to use the machine has empty budgets. For an in-house machine, this does not make sense and, worse still, we have observed that it can generate substantial administrative overheads as users seek extra allocations.

The number of shares allocated to a user is, essentially, an administrative decision. However, in a situation where independent organisations share a machine, the shares that should be allocated to individual users depend both upon the entitlement that their organisation has and to the individual's entitlement within the organisation. For simplicity, we describe Share first in terms of a simple situation where there are no independent organisations involved: all users' shares are simply defined to indicate their right to work compared to other users. We deal with the more complex situation where the combined usage of groups of users must be considered, in the description of hierarchical Share.

Another factor in scheduling is the individual users' rights to alter the relative scheduling priority of their processes. We have preserved the Unix *nice*, a number in the range zero to nineteen, which a user can associate with a process. When users assign a non-zero nice value to a process, they indicate that poorer response is acceptable. The larger the nice value, the poorer the response. The way that this affects charging is another administrative decision: the name, nice, suggests that users who do not need fast response for a process might be kind enough to use nice and get poorer response just out of generosity. In our environment, we felt that it was worthwhile to give users some incentive to use nice. So, we reduced the costs charged for processes with larger nice values.

Finally, the *charges* that Share uses are defined by the relative costs of different resources. So, for example,

we associate a charge with memory occupancy, another with systems calls, another with CPU use and so on. Note that this is another difference between Share and conventional schedulers which define a process's scheduling priority only on the process's consumption of CPU time. In Share, CPU scheduling priority is affected by total resource consumption.

In addition, we set charges at different levels at different times of the day. This is yet another administrative decision. For example, during the university's term time, we charge a peak rate during normal work hours, somewhat less for the hour or two around these, and much less at really off-peak hours.

We note that Share represents a radical departure from the traditional approaches to charging as described by Kleijnen (1968)

> prices should not be changed too frequently, since stability is one of the accepted requirements of a charging system.

We agree that users need to understand the charging system and see it as stable, but we argue that this does not require constant behaviour. It can equally be achieved by behaviour that changes steadily, as in Share where response steadily degrades as a user's resource consumption increases relative to other users.

At several points in this section, we have referred to *administrative decisions*. We emphasise that these administrative decisions are very important. In particular, they are critical to Share's fairness. For example, we have just noted that we charge less at off-peak times and this does seem to help spread the machine load. However, another important factor in setting this policy is that users consider it fair that they be charged less for the inconvenience of working out of normal hours. We also note that some of the administrative decisions are not easy. The fixed-budget model has the merit that one can easily top up empty budgets. So the initial size of a budget may not be so critical. By contrast, in Share, the shares allocated define the right to do work so that when we allocate each first year student half the shares given to a second year student, we are defining the relative amount of work we expect each to extract from the machine.

4. Overview of the Implementation

In this section, we describe Share at a conceptual level. As one might expect, there are two main components, one at the user level and the other at the process level. First, we describe the user level component.

User-level scheduling

- update usage for each user by adding charges incurred by all their processes since the last update and decaying by the appropriate constant
- update accumulated records of resource consumption for planning, monitoring, policy decisions

At this point, Share computes the charges due to a user for the resources they have consumed during the last cycle of the user-level scheduler. The charges are for all resources consumed and they are lower at off-peak periods. This part of the scheduler need not run very frequently because usages generally change fairly slowly.

Note that each user can get an estimate of their share of the machine by comparing their usage against that of all active users. Since this is a convenient and intuitive indication of the response that a user can expect, we provide an estimate of the user's machine share, expressed as a percentage, as part of the standard user profile information.

The remainder of Share operates at the process level. Before we describe it, we note that processes each have a priority and the *smaller* the priority value, the better the scheduling priority. We also introduce the term *active process* to describe any process that is ready to run and, at any point, the active process that actually has control of the CPU is called the *current process*. There are three types of activity at the process

level:

- that associated with the activation of a new process;
- the regular and frequent adjustment of the priority of the current process
- and the regular, but less frequent decaying of the priorities of all processes.

We begin with the first, which occurs in a number of situations, including times when a process relinquishes control of the CPU, times when the active process is interrupted for some reason, and at the regular times that the scheduler usurps the currently active process to hand control to the lowest priority process that is ready to run.

Process activation
update costs incurred by the current process
select the process with lowest priority and set it running

Next is the adjustment to the priority of the current process, which defines the resolution of the scheduler. This ensures that the CPU use of the current process increases (worsens) its priority.

Priority adjustment
^ increase the priority of current process in proportion to the user's usage, shares, and number of active processes

Finally, there is the regular decaying of all process priorities, which must be done frequently compared to the user-level scheduler but can be at a larger time interval than the scheduler's resolution.

Decay of process priorities
^ decay all process priorities, with slower decay for processes with non-zero *nice* values

5. Detailed Implementation

The implementation of Share is shown in the box below. The remainder of this section explains each component, including the setting of the various parameters (which can be altered as the system runs).

5.1 User-level Scheduling

The user-level scheduler is invoked every t1 seconds. The value of t1 defines the granularity of changes in a user's usage as they use the machine. Since usage is generally very large compared to the resources consumed in a second, t1 can be of the order of a few seconds without compromising the fairness of the scheduler. The merit in making t1 reasonably large is that we can afford relatively costly computations at this level without prejudicing the time-efficiency of Share. Our Vax implementation makes t1 four seconds, which is 240 times the scheduler's resolution. On the Cray, we have found that four seconds (400 times Share's CPU charging resolution) is also acceptable.

The first component of the user-level scheduler decays each user's usage. This ensures that usages remain bounded and the value of the constant K1 in combination with t1 defines the rate of decay. We generally consider the effect of K1 in terms of the half-life of usage. In a student environment, we have used a half-life of three days. In other contexts, it has been much shorter but generally of the order of several hours. At a conceptual level, this step is performed for all users. In fact, the effect of the calculation is computed as each user logs in, and so the actual calculation need only be performed for active users.

The next part of the user level scheduling involves updating the usage of active users by the charges they have incurred in the last t1 seconds and resetting the charges tally.

Figure 2. Share implementation

Every t1 seconds: *user level scheduling*

for each user

Decay usage and update with costs incurred in last t1 seconds

$$\mathsf{usage}_{\mathsf{user}} = \mathsf{usage}_{\mathsf{user}} \times K1 + \mathsf{charges}_{\mathsf{user}}$$

Reset cost tally

$$\mathsf{charges}_{\mathsf{user}} = 0$$

Every t2 seconds: *decay of process priorities*

for each process

$$\mathsf{priority}_{\mathsf{process}} = \mathsf{priority}_{\mathsf{process}} \times K2 \times (\mathsf{nice}_{\mathsf{process}} + K3)$$

Every t3 seconds: *priority adjustment*

$$\mathsf{priority}_{\mathsf{current\_process}} = \mathsf{priority}_{\mathsf{current\_process}} + \frac{\mathsf{usage}_{\mathsf{current\_user}} \times \mathsf{active\_processes}_{\mathsf{current\_user}}}{\mathsf{shares}^2_{\mathsf{current\_user}}}$$

At each scheduling event: *current process selection*

$$\mathsf{charges}_{\mathsf{current\_user}} = \mathsf{charges}_{\mathsf{current\_user}} + \mathsf{cost}_{\mathsf{event}}$$

run process with lowest priority

5.2 Process-level Scheduling

From this point on, we discuss the low-level scheduler that deals with processes. It operates in terms of the *priority* of each process. As is common practice in process schedulers, the priority defines the order in which processes are entitled to be allocated CPU resources. Accordingly, it

- schedules CPU resources to the process with the smallest priority, which corresponds to the process being at the head of the queue;
- increases the priority of a process each time it is allocated CPU time, which can be viewed as putting the process further down the queue;
- decays all process priorities steadily so that one might view all processes as slowly drifting towards the front of the queue.

We now discuss how Share combines these activities with user level scheduling.

5.2.1 Decay of Process Priorities. The decay of process priorities ages processes so that those which have not had the CPU achieve better and better (smaller) priority values. The value of $t2$ combined with the value of $K2$ define the rate at which processes age. We need to make $t2$ small, compared to $t1$, because priority values change very quickly. In our Vax implementation, $t2$ is set at 1 second, which is sixty times the resolution of the scheduler. (On the Cray $t2$ is also 1 second.)

The rate at which processes age is affected by their *nice* value. We note that Share preserves the approach of the Unix scheduler to *nice*: it assumes that users normally want the best response possible (which corresponds to a *nice* value of zero) but there are also times when a user is happy to accept lesser response, which they indicate in terms of a *nice* value which is a small integer. (Its range is from zero, the default, to nineteen which gives the worst response.) We define the value of $K2$ as

$$K2 = \frac{K2'}{K3 + \mathsf{max\_nice}}$$

where $\mathsf{max\_nice}$ is the largest *nice* value (19). This ensures that the priority of processes with *nice* set to $\mathsf{max\_nice}$ is decayed by $K2'$ every $t2$ seconds and the priority of processes with *nice* set to zero is decayed somewhat faster. The values of $K2'$ and $K3$ must be sufficiently large to ensure that priorities are well spread and remembered long enough to prevent large numbers of processes from having zero priority.

5.2.2 Priority Adjustment. At the finest resolution of the scheduler, t3, the current process has its priority increased by the usage and active process count of the user who owns the process. (The scheduler resolution, t3, is a sixtieth of a second on the Vax version, one hundredth of a second on the Cray.) Typically, schedulers increase the priority by a constant. Intuitively, one might view the difference between Share and typical schedulers as follows:

- a typical scheduler adjusts the priority of the current process by pushing it down the queue of processes by a *constant* amount;
- Share pushes the current process down the queue by an amount proportional to the usage and number of active processes of the process's owner, and inversely proportional to the square of that user's shares, so that processes belonging to higher usage (more active) users are pushed further down the queue than processes belonging to lower usage (less active) users.

This means that a process belonging to a user with high usage takes longer to drift back up to the front of the queue. (The priority needs longer to decay to the point that it is the lowest.)

We also want users to be able to work at a rate proportional to their shares. This means that the charges they incur must be allowed to increase in proportion to the square of the shares (which gives a derivative, or *rate* of work done, proportional to the shares).

The formula also takes account of the number of active processes (processes on the priority queue) for the user who owns the current process. This is necessary since a priority increment that involved just usage and shares would push a single process down the queue far enough to ensure that the user gets no more that their fair share. If the user has more than one active process, we need to penalise each of them to ensure that the user's share is spread between them and we do this by multiplying the priority increment by the active process count. This is the crux of the Share mechanism for making long term usage, over all resources that attract charges, affect the user's response and rate of work.

Although the model we have described may be adequate for some implementations where process priorities have a large range of values, on the machines where we have implemented Share, process priorities are small integers and so cannot be used directly. We need to normalise the *share* priorities into a range that is appropriate for real process priorities. In addition, where the range in priority values is quite small, we need to ensure that the normalisation procedure does not allow a single very large Share priority value to reduce all other normalised priorities to zero. To avoid this, we define a bound on the Share priority. This

Figure 3. Priority normalisation

Find greatest Share priority for normalisation
max_priority = 0
for each process
    if
        $\text{max\_priority} < \text{priority}_{\text{process}} \leq \text{priority\_bound}$
    then
        $\text{max\_priority} = \text{priority}_{\text{process}}$

for each process
    Scale priority to appropriate range
    if
        $\text{priority}_{\text{process}} \leq \text{max\_priority}$
    then
        $\text{normalised\_priority}_{\text{process}} = (K4 - 1) \times \frac{\text{priority}_{\text{process}}}{\text{max\_priority}}$
    else
        $\text{normalised\_priority}_{\text{process}} = K4$

is calculated in the process-level scheduler as shown in Figure 3 below. K4 is determined by the largest priority available to the low-level scheduler. Note that the Share priority bound does, somewhat unfairly, favour very heavy users. However, they still suffer the effects of their slowly decaying large usage and they

are still treated more severely than everyone else. On the other hand, it helps prevent marooning.

5.2.3 Process Activation. At each scheduling event, Share updates the current user's charges by the costs associated with the event and selects the lowest priority process to run. This aspect of Share is typical of CPU schedulers.

5.2.4 Multiple Processors. Multiple processor machines don't affect the implementation provided that the kernel still uses a single priority queue for processes. The only difference is that processes are selected to run from the front of the queue more often, and incur charges more frequently, than if only one processor were present.

5.2.5 Efficiency The implementation shown in Figure 2 should only be seen as a model of the actual code. For efficiency, some of the calculations that are shown at the level of the process scheduler are actually precalculated elsewhere.

5.3 Edge Effects

In general, it is important to avoid edge effects on scheduler behaviour. In particular, if a user enters the system with zero usage they could effectively consume 100% of the resources, at least for one cycle of the user-level scheduler. Since this is a comparatively long time (a few seconds), this would be quite unacceptable. We now examine why this undue favouritism could occur and how Share deals with the problem.

First, we define the relative proportion of the machine due to a user by virtue of their allocation of shares. This is:

$$\text{machine\_proportion\_due}_{\text{user}} = \frac{\text{shares}_{\text{user}}}{\sum_{u=1}^{\text{active\_users}} \text{shares}_u}$$

This defines the proportion of the machine due to a user in the short term. Now we can also predict the short term future proportion of the machine that a user should get by virtue of their usage.

$$\text{near\_future\_machine\_proportion}_{\text{user}} = \frac{\frac{\text{shares}_{\text{user}}^2}{\text{usage}_{\text{user}}}}{\sum_{u=1}^{\text{active\_users}} \frac{\text{shares}_u^2}{\text{usage}_u}}$$

If everyone is getting their fair share, these two formulae will give the same value for each active user. Indeed, Share works to push these two formulae to the same value for each user. In the case where a user has zero usage (or near zero usage), we need to interfere to prevent that user from being unduly favoured (while other users are ignored). We do this by altering the usage value in the user-level scheduler as shown

Figure 4. Avoiding edge effects

Every t1 seconds: *user level scheduler*

for each user

if

$$\text{near\_future\_machine\_proportion}_{\text{user}} > K5 \times \text{machine\_proportion\_due}_{\text{user}}$$

then

$$\text{usage}_{\text{user}} = \text{usage}_{\text{user}} \times \frac{\text{near\_future\_machine\_proportion}_{\text{user}}}{K5 \times \text{machine\_proportion\_due}_{\text{user}}}$$

in Figure 4. We have set K5 to 2.

5.3.1 System Processes. Processes that run in support of the operating system must be given all the resources that they need. In effect, system processes are given a 100% share of the resources, and it is assumed that they won't use it most of the time. Share is intended to arbitrate fairly between users, *after* the system has taken all the resources it needs.

5.3.2 Marooning. It is possible for a user to achieve a very large usage during a relatively idle period. Then, if new users become active, the original user's share becomes so small that they can do no effective work. This user's processes are effectively *marrooned* with insufficient CPU allocation even to exit. Marooning is avoided by the combination of bounds on the normalised form of priority, the process priority decay rate, and the granularity of the process-level scheduler.

6. Hierarchical Share

Although the simple version of Share that we have described served well for several years, it was inadequate for a machine that is shared between organisations or independent groups of users. Consider the situation where organisations need to share a machine and they want sharing not only between users, but also at the level of the organisation. Share as described above is fine for this situation provided that we can make the following assumptions

1. the total allocation of shares for each organisation is strictly maintained in the proportions that the machine split is made. E.g. if a machine is to be split equally between two organisations, the total shares for each organisation must be the same;
2. the users in each organisation are equally active;
3. $K1$ is acceptable at the organisational level and is constant for all users;
4. costs for resources are consistent for all users, and the other parameters of Share, including $K2$, $t1$, $t2$ and $t3$, are accepted for all users.

We now consider how the simple Share is adjusted to account for each of these factors.

6.1 Shares in an Hierarchical Share Scheduler

It would be impractical to require that the total shares for each organisation be maintained at a fixed value. This would mean that the arrival of a new user would require adjustments to the shares of all users in that organisation. This is a serious problem that could rule out organisational sharing with the simple form of Share.

To preserve the view that each organisation should appear to be operating their own machine, we allow that users be allocated shares as in the simple Share. However, we cannot directly compare such shares across organisations. We need to convert them to a comparable measure. The approach we take is to calculate each user's *machine-share*, the proportion of the machine that their allocation of shares make them eligible to receive. We start at the root of the Share hierarchy tree and convert the shares allocated to each child node into their machine share, using the formula:

$$\mathsf{m\_share}_{\mathsf{node}} = \mathsf{m\_share}_{\mathsf{parent}} \times \frac{\mathsf{shares}_{\mathsf{node}}}{\sum_{n=1}^{\mathsf{siblings}} \mathsf{shares}_n + \mathsf{shares}_{\mathsf{node}}}$$

This calculation is repeated recursively down the hierarchy tree until the $\mathsf{m\_share}$ of each node has been calculated and $\mathsf{m\_share}$ is then used instead of shares in the user level scheduler.

6.2 Varying Levels of Activity

One cannot reasonably assume that the users are equally active at all times. This means that as users log in and log out, they alter the $\mathsf{m\_share}$ value of all users in their scheduling group (and if they are the first user in their group to log in, or the last to log out, they alter the $\mathsf{m\_share}$ of all users who descend from their grandparent node in the hierarchy tree.)

In terms of the operation of Share, this means that some $\mathsf{m\_share}$ values will usually be recalculated at each log in or log out. This poses a small but acceptable overhead.

Share acts fairly under full load but a light load can distort it. Consider, for example, the situation depicted in Figure 5. This shows a case where there are two organisations A and B with an equal share of the resources, where organisation A has one active user A1 while organisation B has two users, B1 with a large share and doing nothing, and B2 with a small share running a CPU-bound process. The *effective* share of

Figure 5. Example of user activity that distorts group sharing

	m_share	description of user activity
Organisation A		
User A1	0.5	active
Organisation B		
User B1	0.45	logged in but inactive
User B2	0.05	CPU bound

the two active users, A1 and B2, differ by a factor of ten and yet the scheduler should divide the resources equally between the two groups, A and B.

First, we define the relative proportion of the machine due to a group by virtue of its allocation of shares. This is:

$$\text{machine\_proportion\_due}_{group} = \frac{\text{shares}_{group}}{\sum_{g=1}^{active\_groups} \text{shares}_g}$$

Now we can also calculate the actual share of resources consumed by a group for the most recent scheduling period.

$$\text{actual\_machine\_proportion}_{group} = \frac{\text{charges}_{group}}{\sum_{g=1}^{active\_groups} \text{charges}_g}$$

If each group is getting its fair share, these two formulae give the same value for each active group. In the case described above, we need to interfere if group B (and hence user B2) is to get its fair share. We do this in the user-level scheduler by reducing the costs of resources consumed by a group that is getting less than a certain amount of its share. This decreases the usage for active users in the group and allows them to increase their share and the group's share.

Figure 6. Group adjustment

for each group (descend hierarchy)
    if
        $\text{actual\_machine\_proportion}_{group} < K6 \times \text{machine\_proportion\_due}_{group}$
    then
        for each user in the group (descend hierarchy)
            $\text{charges}_{user} = \text{charges}_{user} \times \frac{\text{actual\_machine\_proportion}_{group}}{K6 \times \text{machine\_proportion\_due}_{group}}$

K6 is set to allow a group's allocated share to fall below its effective share by some small amount. We chose 10%.

6.3 Differential Decay Rates for Usage

We saw that the simple Share used the same rate of decay for all user's usages. It follows that users within an organisation should have the same usage decay rate. However, we need not do this between organisations. We can illustrate this is terms of the simple Share system operating in the university context where it is deemed appropriate to set a three day half-life for usage in the case of a machine used by undergraduates, but for the research support machine, an acceptable half-life value is twelve hours. When different organisations share a machine, the right to define different decay rates may be important.

In practice, we have not dealt with this problem. There is a simple administrative solution if the organisations can agree to a constant decay rate within each organisation and they negotiate the organisation machine share allocations to take account of this. An alternate, rather messy approach, is a dynamic correction for differential decay rates by keeping two forms of usage: one for each user as we currently do and another for each organisation with a common decay rate applied to all organisational usage values. Then we could make a further adjustment to each group's m_share value (and hence each user's)

to account for any imbalances in the group level usage value.

6.4 Other Parameters

We have not allowed for variability per group or per user in any of these.

7. Evaluation of Share

Some parts of the design we have described were evaluated (Brownie, 1984) before its implementation in 1985. This evaluation with synthetic loads was mainly intended to guide the development of a computational model for the scheduler before it was put into active service on a heavily used machine. This preliminary work smoothed the introduction of the scheduler.

Once Share had been put into service, we used two forms of evaluation. Firstly, we used several monitoring tools to watch it in operation. These have also been useful for administration and for users. They indicate

Resource usage between groups
: Shows the effective share and actual resource consumption by group.

Resource usage between users
: Shows the actual resource consumption for every user.

Effective share distribution
: Plots a graph of users vs. normalised usages. A non-poisson distribution probably indicates problems, such as a class of users (not necessarily in the same group) that are consuming a disproportionately large amount of the resources

Resource event frequency
: Provides feedback on active resource consumptions.

Long term charges
: Provides details on the share of the resources between groups and users over a long time period.

In addition, we have run synthetic tests with pure CPU bound processes to check that Share preserves the proper relationships between users with different shares, usage and number of processes.

In view of the difficulties in creating valid simulation models and synthetic loads (Heidelberger and Lavenberg 1984), we consider that the most important evaluation of Share has been the users' reactions to it in real operation. We now return to the design objectives and report upon our evaluation of Share in terms of them.

7.1 Design goal: that it be fair

We aimed to achieve this goal in terms of a secondary goal: that users be allocated shares which defined their relative machine-share and that users getting more than their machine-share should be penalised with poorer response. On simple tests, with synthetic jobs, we observed that Share met this design goal (Brownie, 1984). More important, however, users deemed the scheduler to be treating them fairly.

Even with the simple, non-hierarchical Share, we have observed a number of situations in which Share has dealt with potentially disastrous situations to the satisfaction of most users. For example, in our student environment, we allocate shares to students on the basis of the relative machine share they should need. If a class is given an assignment that demands significantly more machine resources, the students in that class, and no other students, find the machine slow. With a conventional scheduler, everyone suffered in this situation. Share has proved useful for this problem in that the source of the problem is patently obvious, as is the identity of the person responsible for creating it.

A similar example, with the hierarchical Share system, involved a user who initiated a long running CPU bound process — Share ensured that users in other groups were unaffected by the problem.

7.2 Design goal: that it be understandable

Figure 1 indicates the user's view of Share. Our users appear to be able to appreciate this view and they interpret relatively poor response as an indication that they have exceeded their machine share.

They also become alert to the relative costs of various processes they create since it is directly reflected in their relative response from the machine.

7.3 Design goal: that it be predictable

Each user's personal profile lists their effective machine share and they quickly learn to interpret this is a meaningful way. Users speak of a certain machine share as being adequate to do one task but not another.

7.4 Design goal: the scheduler should accommodate special needs

Share caters for the situation where one needs to guarantee a user (or group) excellent response for a brief period. One simply allocates a relatively large number of shares to the relevant user's (or group's) account for the duration of the special needs. This is a simple procedure that the system administrator can set up to run at the required times.

Clearly, this sort of activity does disrupt other users in that they have to share a smaller part of the machine than usual. In fact, we observe that the favoured users may only make major demands of the machine for brief bursts during the period that they have a high machine-share account. Typically, other users suffer only small periods of reduced response. Although this facility is only necessary on odd but critical occasions,[4] it is an attractive benefit of Share.

7.5 Design goal: that it should deal well with peak loads

In our design environment, one of the classic causes of a peak load is the deadline for an assignment. Because we stagger the deadlines for different classes, this means that one class of students tries to work ever harder as the deadline approaches. In pre-Share days, everyone suffered and the machine ground to a halt. With Share, the individuals in the class that is working to the deadline are penalised as their usage grows. Meanwhile, other students get good response and are often unaware of the other class's deadline. In effect, under heavy load, heavy users suffer most.

7.6 Design goal: that it should encourage load spreading

The most direct observation of Share's load spreading effect is that users do give up when their response gets bad, and especially when it is bad relative to other users. We would like to report that our students now start their assignments early and work on them steadily; unfortunately this is not the case. However, the fact that one class deadline cannot disrupt another does allow students to plan their work and be able to predict that they will be able to get reasonable response if they do work steadily.

7.7 Design goal: that it should give interactive users reasonable response

We can ensure this goal by combining Share with a check at login time that only allows users to log in if they can get reasonable response. In practice, we have not enabled this facility unless there is a very large number of users (over 70 on the Vax). Those who have high usage do get poor response and if the machine is heavily loaded, the poor response may well be intolerable for tasks such as using a screen editor. We view this as an inevitable consequence of Share being fair to users whose fair share is really very small.

In general, Share does ensure good throughput for the small processes that typify interactive use.

7.8 Design goal: no process should be postponed indefinitely

Since Share allocates some resources to every process, this goal is also achieved.

7.9 Design goal: that it should be very cheap to run

Since most of the costly calculations are performed relatively infrequently (in the user-level scheduler), Share creates only a small overhead relative to the conventional scheduler.

---

4. These include demonstrations of software to funding agencies and, in the teaching context, practical examinations.

8. The essential Share

Our description mirrors Share as we have implemented it. The aspect that is essential to Share is that it shares resources fairly between users, rather than just processes. Other aspects can be altered within the Share framework.

In particular, several parameters are defined by administrative decisions and need to be set according to the particular requirements of each machine. For example, we set the constant K1 to make usage decay quite slowly: its half life has ranged from a few hours to three days. It could equally well be of the same order as the process priority decay rate. Since the function of usage is to ensure that process priorities reflect the total activity of the user who owns them, it can do that equally well with a short half life if that is what is required. To date, we have used Share in environments where a long usage half life has been regarded as fair.

Other such parameters that can be altered include the various constants, the frequency with which the user level and process schedulers run and the way that charges are calculated. On the last of these, charges should be selected to reflect the administrators view of the costs of each resource. This may well change in the light of monitoring information or with changes in the hardware configuration. Similarly, the time variance of charges could be altered. In our experience, it seems best to have fixed costs at particular times of day so that users can plan their work in terms of these. In other situations, it may be appropriate to take some other approach: one could dynamically alter costs on the basis of load so that the machine becomes more costly to use at peak times whenever they happen to occur, or one could have fixed costs at all times. Such changes should be taken with care. For example, the suggestions that costs change dynamically may, at first glance, seem attractive and sensible. However, it violates the principle of predictability, a sacrifice that should not be taken lightly.

9. Conclusion

Users perceive the scheduler as fair in practice, and tend to blame poor response more on their past usage, rather than on system overloading. The strengths of Share are that it:

- is fair to *users* and to *groups* in that users cannot cheat the system and groups of users are protected from each other
- gives a good prediction of the response that a user might expect
- gives meaningful feedback to users on the cost of various services
- helps spread load

Share has proved useful in practice, both in teaching and research contexts. Other contexts are possible, such as sharing access to a file server to prevent any one client from monopolising the service.

Acknowledgements

This work was the product of much discussion over a long period. In typical academic tradition, many people had a lot to say about the changes that Share brought to their lives. Many of those comments were very useful. In addition, flaws in the initial design were identified by several people to whom we are grateful.

The first versions of Share grew from the ideas described by Larmouth (1975, 1978) and the basic work by Andrew Hume (1980). Chris Maltby played a critical role in the implementation and monitoring of the first version. Sandy Tollasepp helped to analyse the performance of the first version and John Brownie, the second. Carroll Morgan's suggestions were the basis for revising the whole approach of the first version of Share and they made for the simplicity of the current version. Rob Pike and Allan Bromley independently identified an error in an earlier form of the share normalisation procedure. Glenn Trewitt suggested the current form of taking account of the user supplied nice value.

References

Bach, M.J. "The Design of the UNIX Operating System", Prentice Hall, 1986.

Brownie, J. ‘‘Analysis and Simulation of Share Systems’’, Unpublished Honours Thesis, 1984.

Coffman, E.G. and Kleinrock, L. ‘‘Computer Scheduling Methods and Their Countermeasures’’, Proc. AFIPS SJCC vol 32, 1968, pp11-21.

Heidelberger, P. and Lavenberg, S. S. ‘‘Computer Performance Evaluation Methodology’’, IEEE Trans. on Comp. vol C-33, no 12, December, 1984.

Henry, G. J. ‘‘The Fair Share Scheduler’’, Bell System Technical Journal, October, 1984.

Hume, A. ‘‘A Share Scheduler for Unix’’, AUUG Newsletter, 1979.

Kay, J., Lauder P., Maltby C. and Tollasepp S. ‘‘The Share Charging and Scheduling System’’, Basser Dept. of Comp. Sc. Tech. Rep. 174. May, 1982.

Kleijnen, ‘‘Principles of Computer Charging in a University-Type Organisation’’, Comm. ACM, vol 26, no 11, 1983.

Kleinrock, L. ‘‘A Continuum of Time-Sharing Scheduling Algorithms’’, Proc AFIPS SJCC, vol 36, 1970, pp453-348.

Lampson, B.W. ‘‘A Scheduling Philosophy for Multiprocessor Systems’’, Comm. ACM, vol 11, no 5, 1968.

Larmouth, J. ‘‘Scheduling for a Share of the Machine’’, Software Practice and Experience, vol 5, 1975, pp29-49.

Larmouth, J. ‘‘Scheduling for Immediate Turnaround’’, Software Practice and Experience, vol 8, 1978, pp559-578.

Lauder, P. ‘‘Share Scheduling Works!’’, AUUG Newsletter, 1980.

McKell, L.J., Hansen, J.V. and Heitger, L.E. ‘‘Charging for Computing Resources’’, ACM Computing Surveys, vol 11, no. 2, 1979.

Newbury, J.P. ‘‘Immediate Turnround - An Elusive Goal’’, Software Practice and Experience, vol 8, 1982, pp897-906.

Nielsen, N.R. ‘‘The Allocation of Computing Resources - Is Pricing the Answer?’’, Comm. ACM, vol 13, no 8, 1970, pp467-474.

Ritchie, D.M. and Thompson, K. ‘‘The UNIX timesharing System’’, Bell System Technical Journal, July 1978.

Tollasepp, S. ‘‘The SHARE Resource Allocation System’’, Unpublished Honours Thesis, 1981.

Woodside, C. M. ‘‘Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers’’, IEEE Trans. on Software Engineering, vol. SE-12, no. 10, October, 1986.