Technical Report

UCAM-CL-TR-2
ISSN 1476-2986

Number 2

# Scheduling for a share of the machine

J. Larmouth

October 1974

ISSN 1476-2986

## Summary

This paper describes the mechanism used to schedule jobs and control machine use on the IBM 370/165 at Cambridge University, England. The same algorithm is currently being used in part at the University of Bradford, and implementations are in progress or under study for a number of other British Universities.

The system provides computer management with a simple tool for controlling machine use. The managerial decision allocates a share of the total machine resources to each user of the system (in our case, about 3000 people), either directly, or via a hierarchial allocation scheme. The system then undertakes to vary the turnround of user jobs to ensure that those decisions are effective, no matter what sort of work the user is doing.

At the user end of the system we have great flexibility in the way in which he uses the resources he has received, allowing him to get a rapid turnround for those (large _or_ small) jobs which require it, and a slower turnround for other jobs. Provided he does not work at a rate exceeding that appropriate to his share of the machine, he can request, for every job he submits, the 'deadline' by which he wants it running, and the system will usually succeed in running his job at about the requested time - rarely later, and only occasionally sooner.

Every job in the machine has its own 'deadline', and the machine is not underloaded. Within limits, each user can request his jobs back when he wants them, and the system keeps his use to within the share of the machine he has been given. The approach is believed to be an original one, and to have a number of advantages over more conventional scheduling and controlling algorithms.

J. Larmouth
October 1974

## Contents

| | | |
|---|---|---|
| 1 | Principles and overview | 1 |
| 1.1 | Introduction to Section 1 | 1 |
| 1.2 | Normal scheduling procedures | 1 |
| 1.3 | Main features of the system | 2 |
| 1.4 | Odd points prior to the main argument | 3 |
| 1.4.1 | Allocation of shares | 3 |
| 1.4.2 | Machine-dependent scheduling | 3 |
| 1.4.3 | 'Work done' and 'cost' of jobs | 4 |
| 1.5 | A simple basic scheduling algorithm | 5 |
| 1.6 | Pseudo-time | 6 |
| 1.7 | The user interface | 6 |
| 1.8 | The actual algorithm | 7 |
| 1.9 | Real money | 8 |
| 1.10 | Alternative scheme | 9 |
| 1.11 | Conclusion to Section 1 | 10 |
| 2 | Implementation | 11 |
| 2.1 | Introduction to Section 2 | 11 |
| 2.2 | Machine-dependent scheduling | 11 |
| 2.3 | The $\tau$ to t mapping | 15 |
| 2.4 | Word sizes | 17 |
| 2.5 | Access to files | 17 |
| 2.6 | The TURNROUND statement | 18 |
| 2.7 | The range of P | 19 |
| 2.8 | Cost of the system | 19 |
| 2.9 | Conclusion to Section 2 | 20 |
| 3 | Analysis | 21 |
| 3.1 | Introduction to Section 3 | 21 |
| 3.2 | Assumptions | 21 |
| 3.3 | The Simple Algorithm | 22 |
| 3.4 | The Main Algorithm | 22 |
| 3.4.1 | Model 1 | 23 |
| 3.4.2 | Model 2 | 25 |
| 3.5 | Decaying U | 27 |
| 3.6 | Changes in the $\tau$ to t mapping | 27 |
| 3.7 | Conclusion to Section 3 | 29 |

## Section 1 - Principles and overview

### 1.1 Introduction to Section 1

In this section we develop the algorithm used to schedule jobs, and summarise the main features of the resulting system. Details of implementation appear in Section 2 and a mathematical analysis of some of the effects is given in Section 3.

### 1.2 Normal scheduling procedures

Most computer systems operate a job scheduling mechanism in two stages. In the first stage jobs are given a figure of merit, and placed on a queue ordered according to the figure of merit. In the second stage jobs are selected from the queue for running, giving strong preference to jobs near the head of the queue. A third process usually operates periodically to adjust the figures of merit for jobs currently on the queue. I will call these three procedures machine independent scheduling (also called high level scheduling), machine dependent scheduling (also called low level scheduling), and ageing.

The most common systems operate a combination of 'shortest job first' (figure of merit based on estimated length of job, no ageing) with 'oldest job first' (figure of merit constant, steady ageing). In most systems these are modified by some form of user priority, such that the priority modifies the figure of merit, or forms the only term in it. Ageing may or may not allow a job fed in at a lower priority to eventually have a higher queue position than a job fed in later at a higher priority. Wilson (1) gives a good survey of present approaches, with a discussion of queue length and wait times for various simple algorithms.

The system described here is implemented in the same three procedures as are described above, but the 'figure of merit' calculation takes into account the amount of work done by the user as well as job size and user priority, and the ageing process is dependent on the rate at which the machine is running jobs (in a sense which will become clear later). The combination allows a sharing of the machine resources, the ability for any user to request immediate turnround, and the provision of a fairly accurate statement of when any job in the machine will run.

## 1.3 Main features of the system

The following points are the features which the system was designed to contain. Section 3 - 'Analysis', contains certain derivations which show that, given reasonable assumptions, the system does implement these points.

1). The unit of control is the project. An individual user may have access to one or more projects, and a project may be used by one or more users. (The majority of users have a unique project.) Each project is allocated a certain share of the computer, and the scheduling aims to give it this proportion of the machine resources, averaged over some suitable period. Note that this means we must control the project's rate of working, not the total work done.

2). We control the rate of working of a project by varying the turnround given to jobs run on that project. We should never refuse to accept a job nor should the machine ever be left idle when there are jobs to be run. Thus an underused project will get a very good turnround - to its users the machine will appear lightly loaded. When the workload generated by that project increases, it will receive a steadily worse turnround - the users will think the whole machine is heavily loaded, and must accept the bad turnround (and hence slow rate of working) or must make out a case for more shares.

3). It should be possible for all users, in principle, to get an immediate turnround for a small part of their work, and for ther to specify that they will accept a worse than average turnround for some other parts. For the present, we will call this specification a priority. Jobs with a high priority should in general get a much better turnround than jobs on the same project with a low priority. They will in some sense be charged more for high priority, so that continuous running at any constant priority gives a project the turnround it 'deserves' (for its real work and shares), independent of that constant priority. Constant use of high priority must not enable a project to do better than a neighbouring project with the same shares running at constant low priority.

4). It should not be possible to 'play' the system. Two projects with equal share allocations working hard (in the sense that they each always have at least one job in the machine), should, on average, get exactly the same amount of work done no matter how the work is organised - one big job at a time, one small job at a time, or a lot of big jobs at a time.

5). It should be possible to give fairly reliable estimates of when a job will run, and to allow a user to choose his 'priority' by specifying the turnround he needs. [Although this did not start off as one of our 'principles', it is certainly one of the more attractive features of the system to the users, and much effort has gone into achieving this aim.] There have been attempts at so-called 'deadline scheduling' before, but these usually rely either on a very underloaded machine (idle much of the time), or on a large mass of non-deadline jobs to soak up the 'idle' time between deadline jobs. The present system makes no guarantee that a job will be returned in the requested/estimated time, but achieves good success barring major upheavals (e.g. machine broken for several hours!).

## 1.4 Odd points prior to the main argument

There are one or two asides which should be disposed of at this stage in the discussion.

### 1.4.1 Allocation of shares

There are a variety of possible techniques for allocating shares to a project. Perhaps the three main ones are renting them for real money, direct one-level allocation by project vetting, and hierarchical allocation to departments who allocate sub-shares, etc. Use of real money is described in a later section. With both this and one-level allocation, there is continuous system-wide inflation in shares. New users are normally issued with relatively few shares, and receive more when their work requires it. This reduces the value of other people's shares. Thus if a project obtains a high allocation by 'screaming', the case has to be remade in due course when inflation has reduced the value of the (initially) large allocation. In an environment where users have a degree of permanency, but the work they do is very variable, this is a desirable feature. In a hierarchical system, there is no need for departments to hold a pool of 'reserved' resources to cover possible future needs - they simply issue the appropriate number of sub-shares, which are normalised by the system. In order to maintain given percentages at the top level, all share allocations in a department must be adjusted when a new project is added, so some flexibility, with a recalculation once per month say, is probably desirable. This aspect of the system will not be covered further, except to note that issue of 'pseudo-money' which 'buys' shares adds an extra dimension of flexibility, as will become clear in the section below on 'real money'.

### 1.4.2 Machine dependent scheduling

It is always necessary to have some machine-dependent scheduling which selects jobs for running based on keeping the machine operating at maximum efficiency, avoiding hold-ups for tape-drives, etc. The system we use is fairly straightforward, but for completeness is described in the 'Implementation' section.

Hopefully (and in practice) the machine dependent scheduling will find a suitable job to run from those very near the top of the queue (scheduled to run in under five minutes say). When this is not the case it is a sign that the job mix does not reflect the mix needed by the machine-dependent scheduling, or put another way, that the algorithm used to calculate the 'cost' of a job (see below) does not correctly reflect the cost to the machine of running certain jobs.

### 1.4.3 'Work done' and 'cost' of jobs

In this paper we frequently refer to the 'cost of a job', the 'charge for a job', and the 'work done' on a project. It is well to clarify these terms at this stage, lest confusion result. The 'cost of a job' is obtained by some algorithm applied when the job has been run, and which will normally take into account such things as the time spent in each step, and the memory requirements of the step, the computation time used, (sometimes called CPU time or mill time), the I/O activity of the job, the number of lines printed, the number of tapes used, and so on. The 'charge for a job' will become clear in the discussion below, but it can be defined as the effect the job will have on the turnround subsequent jobs on the same project will get. A possible value would be

cost of job x priority of job

The 'work done' on a project is taken to be the sum of the costs (not the charges) of all jobs run on that project in some period, or some suitable weighted average of the costs.

In addition, we require for each job an 'estimated cost' which can be calculated when the job is submitted, before it is selected for running. The 'fairness' of the system depends on the estimated cost being close to the actual cost, and it is important that the user should not be able to 'pretend' the job is a cheap one. In our case

estimated CPU time x maximum store size

is currently used for the estimate, and the job is aborted if either of these figures are exceeded. This is not an uncommon situation, and in any case is necessary for any effective machine-dependent scheduling.

Note that when we say a user should get exactly the same 'work done' whether he runs one large job or many small ones, this is to be interpreted in terms of cost. If he can split a job into two jobs whose total cost equals that of the original, he should neither gain nor lose, but in general, splitting one ten minute CPU time job into two five minute jobs will not have this property. Any sophisticated cost algorithm is likely to penalise both very short jobs (via a job overhead charge) and very long jobs (via a non-linearity).

## 1.5 A simple basic scheduling algorithm

For each project we have the shares S allocated to that project, and the usage U of that project. U is taken as the sum of the recent charges on that project - that is, for fixed priority, it is a measure of the rate of working on the project. Let C be the cost of a job run on the project, and P the priority (specified freely by the user, for the present). Then when the job has run we set

$$U := U + P \times C$$

we also allow U to decay exponentially by dividing it by the tenth root of ten every day

$$U := U / 10^{1/10}$$

thus U contains a weighted average (exponential weight function) of the work done each day times the 'average' priority. The U value will decay to one-tenth of its former value in a period of ten days.

When a job is read in, we use U and S for the project, and the priority P that the user has requested on the job, to produce a turnround, $\tau$ , for the job. This must be inversely proportional to S and P, and proportional to U. let us use

$$\tau = \frac{U}{S \times P}$$

thus the bigger U, the worse the turnround, the bigger S and P, the better the turnround.

[Note that P=0, priority zero, lowest priority, will give an infinite turnround, but no charge for the job. (Charge, the addition to U, being P x C in this algorithm). This is conveniently implemented by putting such jobs at the bottom of the queue, and not allowing them to 'age' into a non-zero priority. Thus they only run if the machine completely empties of non-zero priority jobs. On an underloaded machine this is a nice facility, although we have found that, when the machine filled, such jobs remained in the queue for weeks on end, and caused the job queue to become full (our limit was 700 jobs). We now allow P only in the range 1 to $P_{max}$. (integer values).]

Note that one of our principles is already being fulfilled. If the user runs continuously at any fixed P, then U will be P times some value, and the P will cancel in the $\tau$ calculation, so that the turnrounds are independent of the (constant) priority being used.

This turnround time $\tau$ is of necessity a pseudo-time, for we must take account of a lot of users asking for more or less immediate turnround in pseudo-time. A dimensioned constant (or in fact, a function) needs to be applied to map $\tau$ , the turnround in pseudo-time, into t, the turnround in hours, minutes and seconds.

## 1.6 Pseudo-time

At this point we look at the system in terms of a possible implementation (our original approach, but not the current one). We hold the job queue in order of increasing $\tau$ values, placing a job in the queue according to its value. Jobs are then run from the top of the queue, and we 'age' the queue by just sufficient to make the $\tau$ value for the job at the top of the queue zero.

Alternatively, we can see the jobs as being distributed along the pseudo-time axis, and we move along this axis running jobs as we go. New jobs are entered on the axis at a point $\tau$ ahead of our present position.

This makes it clear that the rate at which pseudo-time ticks over is a useful concept. By averaging U with a weight function, we have avoided any step functions which would result if U were, for example, zeroed each month, and we can make the following (pious) assumptions:

1). The rate at which pseudo-time will tick over for the next half-hour or so will not be very different from the present rate.

2). The rate at which pseudo-time will tick over in several hours time will not be very different from the rate at the same hour of the day in preceeding days.

In practice, of course there are sometimes quite large statistical fluctuations, and also disturbances arising from differences in the schedule from day to day. Nonetheless, these two assumptions are sufficiently well satisfied to form the basis of a mapping from pseudo-time $\tau$ to real time t. Full details of how this mapping is produced, and the steps which are taken to honour it, are given in Section 2 - 'Implementation'. For the present, suffice it to say that when we have a $\tau$ value we can predict, on the basis of present and previous loadings, when the job will run.

## 1.7 The user interface

We are now in a position to see how the users view of the system relates to the internals. The sophisticated user has two statements in his armoury, a PRIORITY, and a TURNROUND statement. If he simply specifies a PRIORITY, he gives an integer in the range 1 to $P_{max}$ ($P_{max}$ = 225 in our implementation), and this value is used for P in the formula for $\tau$

$$\tau = \frac{U}{S \times P}$$

If he specifies a TURNROUND (in hours, minutes, or days), then the system will use the inverse of the $\tau$ to t mapping to discover the $\tau$ needed for that turnround, then will calculate the priority needed to achieve such a value with his U and S. If the appropriate P value is less than one, he is given P = 1, and a better turnround than he requested (his shares are very underused). If the appropriate P value is larger than $P_{lim}$ ($P_{lim}$ = 64 in our implementation) then he is given $P_{lim}$ and a correspondingly worse turnround - his shares won't stand the rate of working he is trying to achieve, given the load on the machine generated by all other users. If he supplies both a priority and a turnround, then the priority changes $P_{lim}$ (within the range 1 to $P_{max}$). Note that although the user can only specify integer priorities, the system is allowed to produce any value between 1 and $P_{lim}$. In all cases the user is told immediately what the turnround estimate for the job is (and can query the system for revised estimates at any time).

The simple user will normally use only TURNROUND. He is told to make 'reasonable' requests (he quickly learns from other users what is reasonable), and to watch the priority the system chooses. If this rises, he is either being unreasonable or needs more shares. If it reaches $P_{lim}$, he begins to get a worse turnround than he would like. The sophisticated user tends to watch his U value.

A beginner normally omits both statements, and is given a default turnround which gives fast turnround for small jobs, longer turnround for large ones. If he remains a small user, he will probably be running throughout at P = 1, getting better than the default turnround.

There is some further syntactic sugar on the TURNROUND statement, but this will be described later in Section 2 - 'Implementation'.

## 1.8 The actual algorithm

The simple algorithm given above unfortunately fails to satisfy the principles enunciated earlier. Section 3 - 'Analysis' shows that if we have two users A and B, with equal shares, and A always has four times as much work in the machine as B (either because A puts in four jobs at once, or because he runs jobs which cost four times as much as those of B), then A will only get twice as bad a turnround as B, and will therefore do twice as much work.

Section 3 gives the full details, but here we give an actual algorithm which can be proven to work for a reasonably general model, and which has given Cambridge good service for some time. [We progressed to it as we observed users who had found by experiment how to 'play' the simple algorithm!].

To produce a workable system, when we are about to schedule a job we need to have a measure of the amount of work (including the present job) that the project has in the machine, and to increase the turnround if this value is large. For this we need the 'estimated cost of the job', mentioned earlier.

We keep, for each project, a value V which holds the estimated charge (we use charge, but the algorithm would also work with cost - see 'Analysis') for all jobs not yet run (including the job currently being scheduled).

Let the estimated cost of a job be $C_E$, its priority P, its actual cost (calculated when run) C, shares on the project S, and U and V values held for the project.

When a job is read in

$$V := V + C_E \times P$$

$$\tau := \frac{V \times U^n}{P^{n+1} \times S^{n+1}} \qquad \text{[we use } n = 2\text{]}$$

When a job finishes running, or is cancelled

$$V := V - C_E \times P$$

$$U := U + C \times P$$

and each day, as before

$$U := U / 10^{1/10}$$

Note that V is not decayed. To be strictly fair, the contribution each job makes to V should decay to zero (linearly) as that job approaches the time when it runs. [The above algorithm is what we use (for simplicity of implementation), but it does have the undesirable effect that two identical jobs put in, A first then B, requesting identical fixed priorities, can be scheduled to run B first then A if a larger job on the same project finished between the two jobs going in. If the system is driven by TURNROUND (as ours largely is), then the effect is less noticeable, appearing as a higher charge for A than for B. This simplification gives some advantage to the user who only puts jobs in (one or more) when he has no other jobs in the machine - an unfairness we are prepared to live with.].

The effect of n above is to alter the effect of running with a spread of priorities. This is discussed in Section 3. If P is <u>not</u> used in calculating V, then $P^n$ instead of $P^{n+1}$ should be used in the $\tau$ calculation.

## 1.9 <u>Real money</u>

To avoid confusion, I shall reserve the term 'charged' for the P x C contribution to U, and use the American term 'billed' for that quantity of pounds (or dollars!) that the user is expected to produce in payment for his computer time.

It is clear that a user should be billed not only according to the total 'work done' on his project, but also according to the speed of response - his 'average turnround'. Put another way, if one merely bills for work done, on what basis do you allocate shares to the paying customer? Equally, if you bill a man for the shares he has been given, and he has done no work in the relevant period, he might reasonably feel aggrieved.

We have stated as a principle, and show it later for a suitable model, that for a man working hard, his turnround will deteriorate until

$$\text{average work done in a period} \propto \text{shares}$$

where the constant of proportionality is a (possibly time-dependent) system-wide constant.

A sensible billing algorithm would therefore be

$$\text{bill} = \sqrt{(\text{work done x shares})}$$

That is, the geometric mean of shares and work done. If either is zero, there is no bill, and if the man is working hard, the bill is proportional to the work he has done (which is proportional to the shares he got). Thus for a given amount of work to be done, the man can decide how many shares he would like, to give himself enough to do the work, and a margin (if he can afford it) for better response.

The figure could be normalised by the total number of shares issued, to take account of inflation.

In a non-real-money environment (such as most UK Universities) an alternative to direct allocation of shares is to allocate 'funny-money' which is used to pay the bill, and the department is given enough shares each month to bring its expected bill up to the funny-money available. Note that for departments working hard, the two systems result in the same share allocation, but the funny-money approach is fairer to a department which is felt politically to 'deserve' a certain share of resources, but doesn't have enough work to use up that proportion of machine work. (For example, allocation of 'funny-money' on a per capita basis might be sensible).

## 1.10 Alternative scheme

There is one major alternative available. Suppose we remove priority for the moment. Then whenever we want to run a job, we select a job for that project which currently has the smallest value of U/S. This is a system used in at least two 'university' installations where the installation is shared by a (small) number of Universities, and the mechanism is used to share out the machine between the Universities.

For this purpose it is, of course, admirably suited. However, when sharing between a (large) number of projects, there will often not be jobs for some users for several days, their U's will drop behind those of other users, so that they will go to the top of the queue whenever they run jobs. The body of users will be temporarily locked out, and if the share allocations were out of step with the demand for work, the effect will be somewhat extreme. This algorithm is equivalent to 'shortest U/S first with no ageing', compared with our algorithm which is effectively 'shortest U/S first with ageing'. Lack of ageing is known to produce some extreme wait times.

It is also difficult to see how to incorporate turnround estimates and the TURNROUND facility into such a scheme.

## 1.11 Conclusion to Section 1

The system described has been in use for two years at the University of Cambridge, and for about fifteen months in its current form. It has provided both the right degree of administrative control, and considerable user satisfaction (with some hilarity in the early days of bad turnround estimates!). Although implementation requires some effort, it is little more than that needed for a conventional scheduling system with controls on users allocations. The next two sections will discuss, respectively, implementation points, (particularly the machine-dependent scheduling and the $\tau$ to t mapping), and the analysis of certain models of user behaviour which are used to justify the assertion that the principles stated earlier are satisfied.

## Section 2 - Implementation

### 2.1 Introduction to Section 2

This section of the paper discusses a means of implementing the ideas of Section 1. The various points discussed form the 'Practice and Experience' gained at Cambridge in the last two years, and are areas which are likely to be relevant on any machine.

The following main points are covered. First, a description of the machine-dependent scheduling, and how one arrives at the 'current point' for ageing the queue. Secondly, consideration of the mapping from $\tau$ to t (see Section 1), and of the ageing process. Thirdly, a brief discussion is given of the means of access to the file holding S, U and V for each project (see Section 1), and finally, some further details of the TURNROUND statement are given, and some comments on the cost of the system.

### 2.2 Machine-dependent scheduling

We operate by recognising a number of resources which we wish to use in a steady manner. Roughly speaking, a job will be acceptable for running if it does not cause the total (for all jobs running) of any particular resource to exceed some limit. The resources we use are peculiar to our machine, but for the sake of completeness, I list them:

1). Computation (CPU or mill) time for CPU bound jobs. Any job with an estimated computation time of less than 1 minute is assumed to require none of this resource, other jobs are assumed to require an amount equal to their estimate. We limit the total computation time of running jobs to 39 minutes, and of any one job to 20 minutes.

2). 9-track tape drives. We have four drives available. Any one job is limited to the number of drives currently not broken, and the sum over all jobs running has the same limit.

3). 7-track tape drives. Similar to 9-track drives.

4). Private disk packs. Limited as for 9-track drives.

5). Short term memory requirements. This is the maximum step size of a job. Any one job is limited to 400K bytes (we have about 1200K bytes available for jobs), and the sum over all jobs running is limited to 150% of available memory.

6). Long term memory requirements. This is zero if the job is under 1 minute comp, otherwise it is the maximum step size of the job. This sum is limited to the available memory minus 150K (to allow small jobs to continue to go through).

The procedure operates with a number of vectors, each component being a value of one of these resources. We use the following vectors:

| | |
|---|---|
| JOB | The values for the job currently being considered for running. |
| LIM | The limit for any job if no other jobs were running. This depends on what pieces of hardware are broken. |
| SUM | The limit for the sum of all jobs running. This again depends on available hardware. |
| CUR | The current sum of all jobs running. |
| MAX | For each resource, this gives the amount used by that currently running job which is using the largest amount. |
| RES | The amount of resources reserved for jobs which are unable to run yet, but are at the top of the queue. |

We make three passes down the job queue trying to find a job to run. The first pass ends when we find a job which is not OVERDUE (its $\tau$ or t value is still positive). If we found any job in this pass such that

$$\begin{array}{lll} & \text{JOB} < \text{LIM} & \\ \text{and} & \text{JOB} + \text{CUR} < \text{SUM} & \text{for all values} \end{array}$$

we will run that job with the shortest computation time. This is to minimise the size of the OVERDUE part of the queue, (if we ever get one!). [The exception is that if a job is more than one hour OVERDUE, and it is eligible to run, then it is taken, otherwise pass one ends]. If we found no such job, we commence pass two. Pass two differs from pass three only in that, in pass two, we will not run a job for a user if there is a job of his running. In pass three (to cope with an empty machine) we remove this restriction. Thus pass three rarely happens, and we ignore it here.

There are two facilities so far not mentioned. One allows the operators to HOLD a job to prevent it being run. The other allows users to specify jobs as part of a CHAIN. If a job has the CHAIN marker set, we guarantee not to run it at the same time as any other job for the user, (even in pass three) and also not to run a job lower in the queue before an earlier one. [This is a very useful facility in conjunction with job cancellation and the use of TURNROUND to insert a new job in the chain.]

If, in any pass, we find a job which cannot be run because it is HELD, or CHAINed to a running or earlier job, or (in pass one and two) for the same user as a running job, or cannot run because too much hardware is broken (e.g. a tape drive), then we wholly ignore the job. This is one way in which jobs can go OVERDUE.

We now zero RES and begin pass two.

```
        If        JOB < LIM
        and       JOB + CUR + RES  <  SUM
```

then we increment SUM, update MAX, and run the job. We also note the 'current time' as the $\tau$ value of this job, unless the 'current time' is already set beyond it.

```
        If        JOB < LIM
        but       JOB + CUR + RES  >  SUM
```

then the job cannot run yet, and we move on the 'current time' to this point, but continue our scan for a job to run, reserving resources for this job. If a job is found, 'current time' is not advanced beyond this first eligible job.

The reservation process is designed to ensure that a job requiring (e.g.) all four tape drives, is not locked out by a succession of jobs requiring only two or one. The process works by testing

$$JOB + CUR + RES > SUM$$

in any future test for selecting a job, where RES is the amount of reserved resources. We decide what resources should be reserved for this job, and each component of RES is then replaced by the larger of its original value and the reservation for the present job.

There are two points here. First, we regard the resources as being ordered, and the test considers each component from left to right. We reserve only resources to the left of and including the resource which was unavailable. Thus a job which is eligible to run on resource 1 (comp), but not on resource 2 (tapes) will reserve computation time and tapes, but will have a zero value in the reservation vector for the other (untested) resources. Only when its 9-track tape drives become available will we reserve other resources. [Clearly the ordering depends on where the bottlenecks lie in any particular system.]

The second point is the amount which we reserve of a resource. Let the MAX value (the amount used by the currently running job using the largest amount of this resource) be M. Let the SUM value (the available quantity) be S. Then we can plot the amount we reserve against the job's requirements in the two cases $M < S/2$ and $M > S/2$. For $M < S/2$ this is shown in Figure 1.

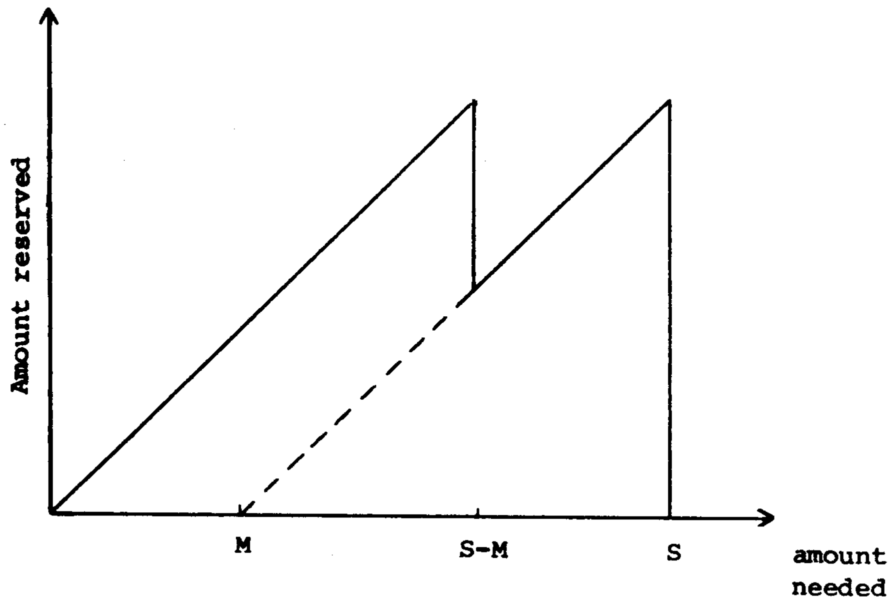

Figure 1 - Amount reserved when M < S/2

For M > S/2 we have the pattern shown in Figure 2.

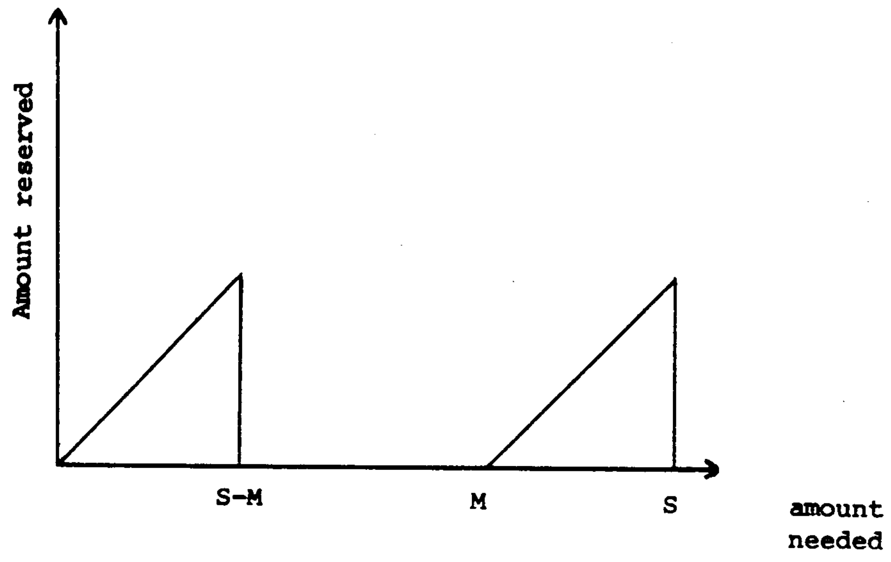

Figure 2 - Amount reserved for M > S/2

[The rationale behind this is that if the job will run in conjunction with the largest job running (that is, its requirements are less than S-M), then we reserve all its requirements. Otherwise it cannot run until a time when the largest job has finished, and M will then become available, so if it needs less than M we need reserve nothing, otherwise we must reserve the difference between its requirements and M.]

Note that if the machine has to be run-down (e.g. for scheduled maintenance), then this can be achieved by progressively reducing SUM, particularly the computation time field. [If we had a better measure of the real time needed by a job, we would use this as a resource. Unfortunately the IBM operating system is unaware of such a concept in any form which has a low variance from run to run of the same job.]

## 2.3 The $\tau$ to t mapping

Most of the discussion so far has assumed that the value held with each job in the queue is a $\tau$ value. When an estimate is needed of when the job should run, we apply the current $\tau$ to t mapping, and all ageing is done on $\tau$ values. This was our original system, and suffers from many disadvantages. Experience has very clearly shown that we should convert from $\tau$ to t when the job is read in, and hold the real time t value in the queue. This is what we now do, but it makes the explanation somewhat circular.

Assume, therefore, that we have a set of twenty-four values giving the pseudo-times covered in each hour of the day in the recent past. These values are updated at the end of each hour of the day, using again an exponential weight function

$$3/4 \text{ old} + 1/4 \text{ new}$$

to give a fairly rapid decay. [In fact, if the two are wide apart (more than one bit position difference) we take a value with a bit set midway between the two - an arithmetic mean for small differences, geometric for large. This gives a bias towards small values, a desirable feature.]. What they are updated with will become clear shortly. We also keep a count to decide whether the machine normally runs during this hour (the count is incremented by 1 if the machine ran during the hour, with a maximum of 8, and decremented by 1 with a minimum of one if it did not.) We then assume that any hour in which the count is more than 4 below the maximum is an hour of the day when the machine will not run. This takes account well of daily software development periods without building a schedule into the system.

Assume further that we have a 'current rate of pseudo-time' - the amount covered in the last twenty minute period. On a restart this is initialised from the history values to be the same as the corresponding period on other days.

Every two minutes we enter the ageing processor, and this maintains the maps from τ to t. Assume we have an existing old map. Then we take the 'current time' produced by the machine-dependent scheduler (a t value), and use the old map to convert this into a τ value. This value is now the increment for this two-minute period, and is used both for the average of the last twenty minutes (used to produce a new 'current rate of pseudo-time'), and for the contribution to the hours value which will be used to update the history. [The situation is slightly complicated by the need to cope with machine failure at any time followed by a restart an arbitrary time later, but this point is rather machine-dependent and will not be covered further].

We now produce a new τ to t mapping as a series of line segments covering t in the ranges 0 - 30 mins, 30 mins - 1 hour, 1 hour - 2 hours, and so on, up to 23 hours - 24 hours. These twenty-five values are assumed to repeat as necessary. The first segment is obtained entirely from the 'current rate of pseudo-time'. The remainder are found by interpolation and smoothing on those history values that have a sufficiently large count.

Finally, this process ages the queue. The 'current time' is 'aged' as if it were a job on the queue with that value of t. It is this following algorithm which shows the advantage of holding t rather than τ in the queue. If we have less than five different users OVERDUE, and if the 'current time' is less than or equal to two minutes (the period since we last aged - in equilibrium the 'current time' will be exactly two minutes), then the whole queue is aged by two minutes. If we have too many OVERDUE, we find a gap of at least (4 mins - 'current time') between two jobs, and age the queue above this point by two minutes, below this point not at all. If we are running fast ('current time' greater than two minutes), then we age thirty minutes worth of work at the front of the queue by 'current time', the rest of the queue by two minutes. Finally, after a restart when the machine has been down for t minutes, we look for a gap in the queue of t+2 minutes, and age all jobs above the gap by t minutes.

All this means that the longer a job's estimated turnround, the better its chances of continually decaying by two minutes every two minutes. The shorter its turnround estimate gets, the more likely it is to be affected by machine down time, overloading jams, or underloading.

It should now be clear how the history and rates adjust themselves when they are either too big or too small. The existence of an equilibrium and its stability is just about as problematical as in the economics field, but the maths goes through, and it seems to work.

There is one further point to make here. By applying τ to t on input, we can flatten or raise the mapping without affecting jobs already on the queue. The two minute ageing processor looks to see if any period of t is overloaded (in terms of number of jobs on the queue and total estimated computation time of those jobs), and progressively raises or lowers the $\tau$ to t mapping to prevent overloading or underloading from becoming critical. There is one constraint in any such manipulation. If a job with a certain $\tau$ value is fed in at time t, and a job with an equal $\tau$ value is fed in two minutes later, the second job must not end up in the queue ahead of the first. This limits the rate of change of the mapping in terms of the amount that portion of the queue has decayed. The details are given in Section 3.

## 2.4 Word sizes

This is an important point. The t values of jobs in the queue must be capable of holding sufficiently large values, and a sufficiently small grain. We use 58 days and centiminutes - thus requiring 24 bits. [As we are subtracting off (in the ageing process) small quantities, using floating point does not help]. The $\tau$ to t mapping must also be held as a set of integers, and getting the scale of $\tau$ right so that bits are not lost off the bottom nor overflow at the top is tricky. It clearly depends on the values of S which are issued, the number of users, and so on. We got it by trial and error, and I have no better advice! The rate at which $\tau$ ticks over is very much larger ($2^5$ times) at night than in peak periods. This aggravates the situation!

The values U, V, and S (and P) can sensibly be held floating point. This is essential for U, to cope with the decay, and for V when non-integer P is allowed. The $\tau$ calculation must be done floating point, and care taken with the mapping to t. What we do is find from the mapping the $\tau$ value for 24 hours, float it, then find the number of days in the estimate and only fix the remainder, thus keeping the integer $\tau$ value needed for the mapping to a sufficiently small value.

Some care must be taken (for projects with a lot of users working hard) to ensure that the floating point addition of P x C to U does actually make a change in the U value. We have not hit this problem (with 7 decimal floating point), but we have come near to it.

## 2.5 Access to files

A problem which is common to the ICL 1900 series and the IBM OS system is how the machine-independent scheduler can access the files containing the U, V, and S values for a project (and incidentally the passwords in scrambled form, the valid users of the project, etc., etc.). In our case we could not let the operating system halt the relevant process (HASP) while it performs I/O, and this may be a common problem. There is also the problem of security of the files. Our solution may be of general interest, so I mention it here. We set up a job, (called JOB 0) which is the first job started whenever the system comes up. It runs forever, and amongst other things looks after the daily decay. A new SVC (supervisor call, extracode) was introduced which allowed requests or data to be transferred between HASP and JOB 0 (with due authorisation). To read data, HASP would request from JOB 0 the record it needs, and JOB 0 sends a return code saying WAIT. HASP then arranges to retry JOB 0 some time later (one centisecond for us) to get its data. The record is accessed by a block number which is obtained by a hashing of the project number, so the number of transfers is small, and JOB 0 maintains a small (twenty item) circular buffer.

If HASP wishes to write into a record (to update U or V), then life is more interesting, for there may be several activations of the reader in progress at once, and an interlock is needed which will not cause halting, and which is safe against administrative programs updating (e.g.) from abending after seizing a semaphore. The arrangement is that whenever a WRITE is requested, the original record (as HASP thinks) is also presented. JOB 0 checks this against the record in the file, and if there is agreement the update proceeds. If there is not, then another process has done an update since the first one read the record, and JOB 0 returns a code saying 'try again'.

The JOB 0 mechanism is also used for other purposes beyond the scope of this paper (file space allocation, privilege authorisation, etc.) and on some systems (probably even on ours!) a simpler mechanism will work. Much depends on the structure of the particular operating system. This problem represented the only real technical difficulty we had to overcome in implementing the system - the rest was logical problems and sheer slog!

## 2.6 The TURNROUND statement

This has some features beyond those described in Section I. The user can say

TURNROUND 50

(meaning 50 minutes), or specify MINS, HOURS, or DAYS. He can also say, e.g.

TURNROUND 1 HOUR OR OVERNIGHT

meaning 'if 1 hour is not achievable with my priority limit, do not give me the best possible, give me OVERNIGHT (or any other value) instead'. OVERNIGHT is a synonym for BY 0800, the keyword BY being available with any time to get the system to calculate the number of minutes to that time.

TURNROUND ... OR REJECT

allows the user to have the job rejected if the requested value is not attainable.

The default values for turnround (no PRIORITY or TURNROUND specified or TURNROUND DEFAULT) are proportional to the cost of the job, and thus correspond, for users with nothing in the machine, to a priority independent of the job cost.

## 2.7 The range of P

For historical reasons, our users actually specify P in the range 1 to 15 (integer) and this value is squared before being used. With n=2 in the main algorithm, this gives a potential range of $2^{24}$ in $\tau$. This is excessive and if we were starting again, I would rethink this area. Further, P would be better specified on a log scale - for equal effect, the user must use P = 1, 2, 4, 8, and so on, each one then dividing his $\tau$ by 64. This means that the 'grain' of P is too coarse at the lower end, too fine at the upper. In order to protect the user from himself (and to avoid jobs sitting on the job queue for an excessive time), we effectively reduce the range of P by rescheduling any job with a t estimate of over four days as if it were submitted requesting four days and a priority limit of $P_{max}$.

## 2.8 Cost of the system

A frequent question is the cost of such a system as this. A great deal depends on what the system is compared with, but the cost is not as great as might be supposed. There are three points at which costs are entailed. Firstly, when a job is read in, a single disk transfer is needed to read down U, V, S, and so on, and a second is needed to write back the new V value. In a more conventional system only one transfer is probably needed to check that the users time allocation has not run out.

At the end of a job two more transfers are needed to read and update U and V. These will be needed in any system controlling use of the machine.

Two transfers will be a small addition to the disk traffic produced by almost all jobs, and the computation time requirements at these two points are completely negligible when priority is specified. If TURNROUND is specified, there is a slightly greater computation requirement to interpolate to find the priority, but this is still a small fraction of a second.

The third point is the ageing process, which we conduct every two minutes, and the checkpointing (for restart purposes) of history values. The load from this source, although not completely negligible is still well under a second of computation and a couple of disk transfers every two minutes.

There is one other cost. That part of the job queue which is resident must contain the t value for the job (24 bits say), and in a more conventional system, only the priority (4 bits say) might be held.

This imposes a memory demand. Further, the coding must either be resident, or some of it read in every two minutes and at the start and end of a job. Either way there is a further load here. Total coding for the scheduling system is of the order of 8 or 9k bytes, and compared with the code brought in to run a job, is negligible.

## 2.9 Conclusion to Section 2

This section has aimed to provide some of the details useful to an implementor, in particular, it has covered most of the points where our initial approach was inadequate or in error and required later refinement.

## Section 3 - Analysis

### 3.1 Introduction to Section 3

This section of the paper contains four main sections. First we give an analysis to show that the simple algorithm

$$\tau = \frac{U}{S \times P}$$

is inadequate, and secondly we give an analysis of the

$$\tau = \frac{V \times U^n}{P^{n+1} \times S^{n+1}}$$

algorithm.

Thirdly, we look at the effects of the time constant in the decay of U, and finally we analyse the permitted changes in the τ to t mapping.

### 3.2 Assumptions

In the first two sections we will be making three main assumptions.

1). That the system is in a steady state. We assume that the U value is such that the decay in it is exactly balanced by the increments due to charges for jobs being run. Section three takes a brief look at a non-steady U value.

2). That the τ to t mapping is a simple constant multiplier, that is, that pseudo-time ticks over at a constant rate. This assumption is certainly false if one considers night time and day time (by a very large margin). This means that if we write

Work done = k x shares

the k is only constant for those users working at the same time of day. The night-time user will have a very much larger k. This automatically reflects the antisocial nature of the CPU time available at night, giving suitable encouragement to people to work at this time, so what at first appears to be a problem can be viewed as an advantage. The analysis proceeds as if the night/day effect were absent.

3). That the estimated cost of jobs equals their actual cost. This is only relevant to analysis involving V.

3.3 <u>The simple algorithm</u>

$$\tau = \frac{U}{S \times P}$$

Assume constant priority, and consider a user running jobs one after the other, always putting one job in the moment the previous job has run. Then the work done, W per unit time, if each job gets a turnround t, and the cost of the job is C will be

$$W = \frac{C}{t} \qquad (1)$$

and (assuming P=1), the steady U value will be

$$U \propto \frac{C}{t} \qquad (2)$$

But the algorithm gives us

$$t \propto \tau = \frac{U}{S}$$

so from (2)

$$t^2 \propto \frac{C}{S}$$

and using (1)

$$W \propto \sqrt{(SC)}$$

This means that the larger the jobs the man is running, the greater his rate of working. A similar effect occurs if he puts several jobs in at once.

This demonstrates that the simple algorithm does not satisfy our requirements in this special case, and hence cannot in general.

3.4 <u>The main algorithm</u>

$$\tau = \frac{V \times U^n}{P^{n+\alpha} \times S^{n+1}}$$

where V is incremented by $P^{\alpha} C_E$ ($C_E$ the estimated cost of the job) and $\alpha = 0$, or 1 and $n = 1, 2, 3, \ldots$

In the last section we had merely to prove incorrectness, and we did this by a special case. In this section we would like to prove correctness, so that a completely general model should be analysed. I give below an analysis for two important models, which demonstrate the technique. A somewhat more sophisticated model can be analysed, and gives similar results.

### 3.4.1 Model 1

Let the user run m jobs at once; whenever a job finishes, he puts in another job of the same type at the same priority.

Thus we have m streams of jobs, jobs in the i th stream having cost $C_i$ and running at priority $P_i$

Let the turnround in the i th stream be $T_i$. Then as before

$$V = \sum_i P_i^{\alpha} C_i \ , \quad W = \sum_i \frac{C_i}{T_i} \qquad (3),\ (4)$$

and

$$U \propto \sum \frac{C_i \times P_i}{T_i} \qquad (5)$$

and

$$T_i \propto \frac{V \times U^n}{P_i^{n+\alpha} \times S^{n+1}} \qquad (6)$$

We now solve these equations.

from (6), $$\frac{C_i \ P_i}{T_i} = \frac{P_i^{n+\alpha+1} \times C_i \times S^{n+1}}{V \times U^n}$$

from (5), $$U \propto \sum \frac{C_i \ P_i}{T_i} = \frac{S^{n+1}}{V \times U^n} \ \sum_i P_i^{n+\alpha+1} \ C_i$$

therefore $$U^{n+1} \propto \frac{S^{n+1}}{V} F \qquad (7)$$

where $$F = \sum_i P_i^{n+\alpha+1} \ C_i \qquad (8)$$

from (5), $$T_i \propto \frac{V \times S^n \times F^{n/(n+1)}}{P_i^{n+\alpha} \times V^{n/(n+1)} \times S^{n+1}}$$

therefore $$\frac{C_i}{T_i} \propto \frac{C_i \ P_i^{n+\alpha} \times V^{n/(n+1)} \ S^{n+1}}{V \times S^n \times F^{n/(n+1)}}$$

from (4), $$W = \sum \frac{C_i}{T_i} \propto \frac{\sum C_i \ P_i^{n+\alpha} \times S}{F^{n/(n+1)} \times V^{1/(n+1)}}$$

i.e.
from (3), (8), $$W \propto S \left\{ \frac{\sum C_i \ P_i^{n+\alpha} \qquad i}{(\sum P_i^{n+\alpha+1} \ C_i)^{n/(n+1)} \times (\sum P_i^{\alpha} \ C_i)^{1/(n+1)}} \right\}$$

<u>**This shows that $W \propto S$, provided the function in braces is close to 1 for all reasonable values of $C_i$ and $P_i$.**</u>

1). Consider $P_i$ constant equal to P.

Then $$\frac{P^{n+\alpha}}{(P^{n+\alpha+1})^{n/(n+1)} \times (P^{\alpha})^{1/(n+1)}} = \frac{P^{n} \times P^{\alpha}}{(P^{\alpha})^{n/(n+1)} \times (P^{\alpha})^{1/(n+1)} \times P^{n(n+1)/(n+1)}} = 1$$

and
$$\frac{\sum_i C_i}{(\sum_i C_i)^{n/(n+1)} \times (\sum_i C_i)^{1/(n+1)}} = 1$$

<u>So the function gives 1.</u>

2). Consider $m = 2$, $P_1 = 1$, $P_2 = P$ (two streams, a high priority one and a low priority one). Then the function gives

$$\frac{C_1 + P^{n+\alpha} C_2}{(C_1 + P^{n+\alpha+1} C_2)^{n/(n+1)} \times (C_1 + P^{\alpha} C_2)^{1/(n+1)}}$$

Now let us assume a turnround variation of y between fast and slow, and a proportion of x of total work done at high priority. Then

$$\frac{C_2}{C_1} = \frac{x}{y} \quad \text{(as y times as many high priority jobs are run)}$$

further $y = P^{n+\alpha}$

so we have $$\frac{1 + x}{(1 + xy^{1/(n+\alpha)})^{n/(n+1)} \times (1 + xy^{-n/(n+\alpha)})^{1/(n+1)}}$$

[This is 1 at both $x = 0$ and $x = \infty$]

If we take $x = 0.1$, $y = 256$, and consider values for $\alpha = 0$ and 1 and $n = 1$ and 2, we obtain

| | | |
|---|---|---|
| $\alpha = 0$, | $n = 1$ | 0.2 |
| $\alpha = 1$, | $n = 1$ | 0.62 |
| $\alpha = 0$, | $n = 2$ | 0.53 |
| $\alpha = 1$, | $n = 2$ | 0.72 |

This clearly shows the advantage of $\alpha = 1$, $n = 2$.

### 3.4.2 Model 2

Let the user run one job at a time, cycling through m jobs of computation time $C_i$, each run at priority $P_i$. Again, a job is put in as soon as an earlier job finishes.

Let the turnround for the ith job be $T_i$, then

$$V_i = P_i^{\alpha} C_i, \qquad W = \frac{\sum C_i}{\sum T_i} \tag*{(9), (10)}$$

and

$$U \propto \frac{\sum C_i P_i}{\sum T_i} \tag*{(11)}$$

and

$$T_i \propto \frac{V_i \times U^n}{P_i^{n+\alpha} \times S^{n+1}} \tag*{(12)}$$

$$= \frac{P_i^{\alpha} C_i \times U^n}{P_i^{n+\alpha} \times S^{n+1}} \qquad \text{from (9)}$$

$$= \frac{C_i}{P_i^n} \times \frac{U^n}{S^{n+1}}$$

Solving these equations,

$$U \propto \frac{\sum C_i P_i \times S^{n+1}}{U^n \sum \frac{C_i}{P_i^n}} \qquad \text{from (11)}$$

or

$$U \propto \frac{S\left(\sum C_i P_i\right)^{1/(n+1)}}{\left(\sum \frac{C_i}{P_i^n}\right)^{1/(n+1)}}$$

therefore

$$T_i \propto \frac{C_i}{P_i^n} \times \frac{1}{S} \times \frac{\left(\sum C_i P_i\right)^{n/(n+1)}}{\left(\sum \frac{C_i}{P_i^n}\right)^{n/(n+1)}} \qquad \text{from (9), (12)}$$

$$W \propto S \left\{ \frac{\sum C_i}{\left(\sum C_i\ P_i\right)^{n/(n+1)} \times \left(\sum \frac{C_i}{P_i^{\ n}}\right)^{1/(n+1)}} \right\}$$ from (10)

Again, W ∝ S provided the function is sufficiently close to 1.

1). P constant.
The function becomes

$$\frac{\sum C_i}{\left(\sum C_i\right)^{n/(n+1)} \times \left(\sum C_i\right)^{1/(n+1)} \times P^{n/(n+1)} \times 1/P^{n/(n+1)}} = 1$$

so the function gives 1.

2). Consider two jobs only, $P_1 = 1$, $P_2 = P$.

Then the function gives

$$\frac{C_1 + C_2}{(C_1 + PC_2)^{n/(n+1)} \times (C_1 + \frac{C_2}{P^n})^{1/(n+1)}}$$

Let the ratio $C_2/C_1$ be x (as an equal number of $C_1$ and $C_2$ jobs are run) and let $y = P^{n+\alpha}$ as before.

Then

$$\frac{1 + x}{(1 + xy^{1/(n+\alpha)})^{n/(n+1)} \times (1 + xy^{-n/(n+\alpha)})^{1/(n+1)}}$$

This is the identical formula to that obtained in model 1. Thus we have reasonable assurance that, at least for small quantities of work done at high priority,

$$W \propto S$$

no matter what the mode of working.

This analysis may tend to encourage higher values of n. The disadvantage of large n can be seen if we consider two users who run at a constant priority of 1, and do the same amount of work, but have different quantities of shares.

We have

$$\tau \propto \frac{1}{S^{n+1}}$$

So for n = 2, a man with twice as many shares, doing the same work, will get eight times as good a turnround. For higher n the effect is even more extreme. Thus n = 2 represents a compromise.

## 3.5 Decaying U

If we consider only the n = 2 case, we have that

$$t \propto U^2 = k_1 U^2 \text{ say}$$

Now if a user waits a time $t_1$, before putting his job into the machine, it will come back at time

$$t_1 + k_1 U(t_1)^2$$

But $\qquad U(t_1) = U_0 e^{-k_2 t_1}$

So $\qquad t_1 + k_1 U_0^2 e^{-2k_2 t_1}$

is the time the job comes back.

If the user tries to minimise this, we have

$$1 - 2k_2 k_1 U_0^2 e^{-2k_2 t_1} = 0$$

That is, provided $U_0$ is sufficiently small, so that

$$2k_2 k_1 U_0^2 < 1$$

he does best to put his job straight on. Otherwise he should wait for $U_0$ to decay.

Now $k_1 U_0^2$ is the time estimate he would receive if he put the job on now, and $1 / k_2$ is the time it takes for U to decay by a factor of e.

So the user should wait before submitting if the time estimate he would get is greater than half the time it takes for U to decay by a factor of e.

Now e is approximately the square root of ten, and if we decay by ${}^{n}\!\sqrt{10}$ each day, the maximum sensible time estimate for the user to receive is n/4 days.

It is probable that the decay time on U should be adjusted to suit the load on the machine, such that very few jobs get turnround estimates in excess of n/4 days.

## 3.6 Changes in the τ to t mapping

We hold the τ to t mapping as 25 integer values giving the increment in τ for t ≐ 0 to 30 mins, 30 mins to 1 hour, 1 hour to 2 hours, 2 hours to 3 hours, etc. Let us call the old mapping $\tau_i^{old}$ (i = 1 to 25), and the new mapping $\tau_i^{new}$.

For simplicity we will assume the whole queue has been aged by $\delta t$. (It is left to the reader to modify the algorithm where this is not the case!).

If we consider two of the line segments involved in the $\tau$ to t mapping, we have the situation shown in figure 3.

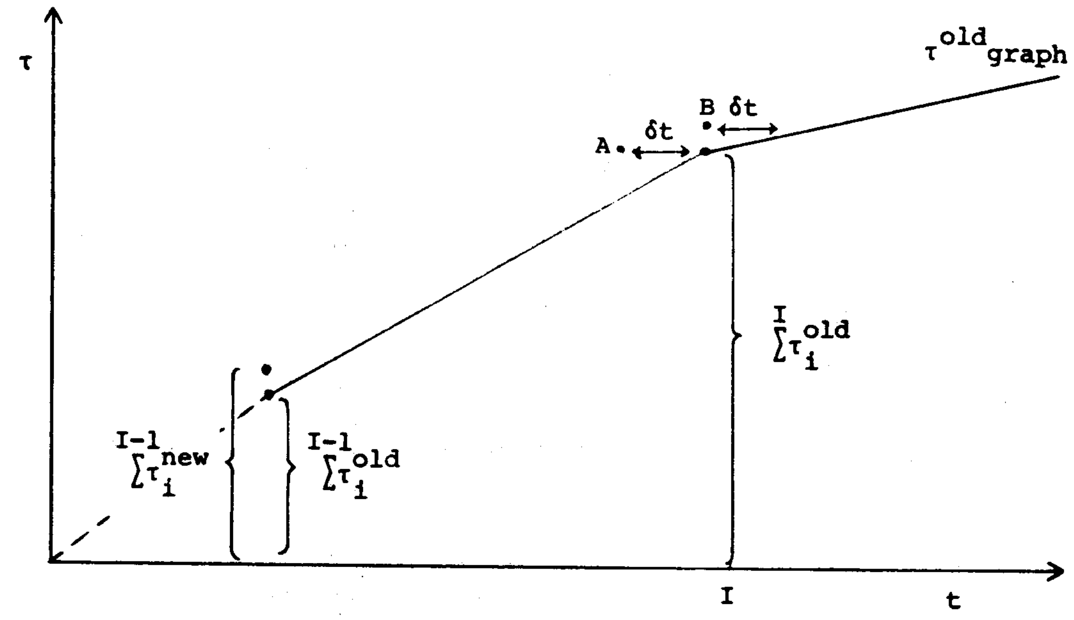

Figure 3 - the $\tau$ to t mapping.

Assume we have set $\tau_i^{new}$ for i = 1 to I-1. Assume further we have values $\tau_i^{aim}$ which is the new mapping we would like to put in. Then

$$\sum^{I} \tau_i^{aim} - \sum^{I-1} \tau_i^{new}$$

is the value of $\tau_I^{new}$ which we would like to set, but we must ensure that the $\tau^{new}$ graph lies wholly below the graph representing a job read in and decayed by $\delta t$. That is, it must be wholly below the old graph shifted left by $\delta t$. This is achieved by keeping it below the points A and B in the diagram, for each point I. Thus we get

$$\sum_i^{I} \tau_i^{new} - \sum_i^{I} \tau_i^{old} \leqslant \frac{\delta t}{60} \tau_I^{new} \quad \text{for point A}$$

and $$\sum_{i}^{I} \tau_i^{new} \leqslant \frac{\delta t}{60} \tau_{I+1}^{old} + \sum_{i}^{I} \tau_i^{old}$$ for point B

That is, $$\tau_I^{new} \leqslant \frac{60}{60-\delta t}\left(\sum_{i}^{I} \tau_i^{old} - \sum_{i}^{I-1} \tau_i^{new}\right)$$

and $$\tau_I^{new} \leqslant \sum_{i}^{I}\tau_i^{old} - \sum_{i}^{I-1} \tau_i^{new} + \frac{\delta t}{60} \tau_{I+1}^{old}$$

This needs modifying for the first two intervals (where widths are 30 minutes and not 60), and also needs rethinking if $\delta t > 60$, but the above formulae are the basic rules.

## 3.7. Conclusion to Section 3

This section of the paper has covered some of the more mathematical points associated with the algorithm, and has hopefully produced the justifications for the form it takes and the way it is used.

## Reference

(1) R. Wilson, Investigation of Operating System Techniques: A survey of techniques used in Job Scheduling, Resource Allocation and Dispatching. The Department of Computer Science, The Queen's University of Belfast. Belfast Report 11.