SCHEDULING MULTIPROGRAMMED COMPUTER SYSTEMS:

AN ANALYTICAL APPROACH

by

Robert Mahl

Computer Science

University of Utah

March, 1970

This research was sponsored by the Advanced Research Projects Agency, Department of Defense, and was monitored by AFSC, Research and Technology Division, Rome Air Development Center, Griffiss Air Force Base, New York 13440, under Contract AF30 (602) -4277.

SCHEDULING MULTIPROGRAMMED COMPUTER SYSTEMS:

AN ANALYTICAL APPROACH

Introduction

In a multiprogrammed computer system, several jobs are using the facilities of the system at the same time. However, a given facility (or resource) is generally only allocated to one user at a time. But, while working, jobs generate requests for some facilities and liberate other facilities; thus, conflicts may arise when several jobs request the same facility. The system's scheduler is a set of mechanisms to solve these conflicts.

The scheduler should try to maximize the use of the resources while giving higher priority to users having a higher urgency. This paper is an attempt to show that better schedulers could be built if some model of the interference between user's requests for facilities were available.

Of course, a solution to this problem depends on some information being available on the patterns of the user's requests for facilities. Haberman [4] has shown that such information can be useful in avoiding deadly embrace of processes in a time-shared environment. This information might be provided either by the user himself, or extrapolated from statistics collected by the system.

What information would be useful? It should be relevant to the scheduler by permitting computation, for instance, of a maximum possible "interference" between different jobs; it should be simple and condensed because the scheduler has to operate rapidly; and finally this information should be easily available and characteristic enough of a given program that it could be used without any modification for several runs of the same program with different input data. In this paper, the information used will be the proportion of usage of the various "non-memory" resources over a rather long period of time during which a job runs, and the total (maximum) amount of memory required by the job at various levels.

What exactly does the scheduler do?

In our terminology, scheduling means allocating resources or preparing an environment in which resources will be allocated. These resources are memory (drum, disk, core, fast registers), tape drives, C.P.U., busses, channels, etc..

Note that some scheduling is usually done by hardware, which for instance resolves conflicting requests for one memory bank. But even in these cases, it might be desirable to have a software scheduler which decides, for large intervals of time, which priorities the jobs (or busses, or CPU's) will, be assigned, for instance any memory access requested during this large interval of time.

The following decisions and strategies are completely independent from the scheduler, except that they may feed information or requests into the scheduler and they should not be confused with scheduling activities:

1) Page replacement algorithms in a computer with paged memory: which page should be extracted from memory, and should the size of the working set of pages be changed, and should there be any prepaging.

These decisions can be made by the programmer, or by the system, but in any case they concern program optimization and not system optimization (insofar as we can say that we don't improve the system by improving the user's programs running under it).

2) Deciding on the "external" priorities of jobs; some jobs are more urgent than others. This might be decided either by the system or by the user himself (who is willing to pay more to get his job executed soon). External priority can be reduced to an economic criterion, the price offered by the user per unit computation of his job, which is then fed into the scheduler, and will serve the scheduler in order to build its own optimization criterion.

3) Statistics to be used by the scheduler or by the paging algorithm or to compute external priorities, can theoretically be considered to be collected independantly of those decision-making processes.

We consider that a scheduler is a mechanism using the following information:

1) On each program: some information about the kind of service wanted by the program, either on a long term range, or because of a current request for a facility.

2) The economic "bid" of each job, characterizing its "external" urgency.

3) The resources available to the system.

We believe that some scheduling should be done for intervals of various duration of time. Microscheduling is being done for periods in milliseconds, macroscheduling for periods of hundreds of milliseconds, while scheduling of tapes

should be done for minutes, and some real time users don't want to use the system at all if they are not assured of getting some minimum guaranteed resource usage.

The scheduler decides to allocate some resources to some users, and chooses parameters to be fed into the "lower level" scheduler (which handles smaller time intervals).

The scheduler tries to optimize the economic criterion of the system, which is the sum of the economic criteria of the jobs being allocated. For any job, the economic criterion can be either under complete control of the user, or computed by the system according to some external urgencies. But these computations are external to the scheduler, contrarily to what happens on some current computer systems where restructuring the external priority queues is an essential function of the scheduler.

Microscheduling and Macroscheduling

Macroscheduling will refer to those scheduling operations which are related to the allocation of central memory. The time interval between two macrodecisions is rather large (greater than 10 milliseconds on most systems). Microscheduling concerns the allocation of the arithmetic and control units and of some fast busses, to programs which are already essentially present in central memory. For instance, the decision of what job is allocated access to the drum for page-in and page-out operations between drum and main core memory is a microscheduling decision in current computer systems where the programs are kept in core while paging takes place. In future computer systems, this kind of paging will most probably be replaced by a paging between two fast levels of memory, like on the 360/85.

	Current computer Systems	Future Computer Systems
Macroscheduling	Allocation of core memory	Allocation of core memory and fast registers
Time between macro-decisions	≈ 1 sec.	≈ 10 millisec.

	Current	Future
Microscheduling	Conflicts of accesses to drum and disk Allocation of CPU and fast registers	Allocation of CPU Conflicts in swapping between central core and fast registers

In future computer systems, the transfer rates of the channels and busses and the bandwidth of main memory are expected to grow faster than the speed of the memory. This opens an entire set of new strategies for scheduling; for instance, it might be smarter to swap out a program during every I/O completion delay encountered by that program, rather than to leave it in an expensive memory during that time. However, such swapping strategies will not be studied in this paper.

We prefer the words "microscheduling" and "macroscheduling" to "microqueuing" and "macroqueuing" [1], because the latter suggest the use of FIFO queues by the scheduling algorithm, which is a practice copied from real-life strategies, but which is not so easy to justify in a more sophisticated environment. Note that, in our terminology, "scheduling" and "allocating" are synonymous.

The models to investigate

In a first stage of this study, we are going to study microscheduling models where the programs are already in memory. A program will be considered as a sequence of calls to various resources: C.P.U., I/O,...such that one and only one resource is called at a time by a given program (no double buffering for instance); this limitation could be removed, but helps to simplify the presentation.

Various queueing strategies will be studied. Some assumptions will be made about the programs candidate to run, and the effect of these assumptions on the use of resources and progress rates of the jobs will be computed. There are several resources, each having its own microscheduler, which is independent of the other microschedulers, the strategy of each microscheduler being determined by the macroscheduler.

Some definitions

User: an entity which requests and seizes resources, and which might also give some information about its future resource requirements.

The words user, job, program & process describe the same concept in this study.

<u>Virtual time of a user</u>: a time reference in which the user's program is running at the maximum speed allowed by the hardware (a request never waits, but is always immediately served). If a user permanently has top priority for accessing all the resources, virtual time for this user and real time are equivalent, except for a change of origin in time.

<u>Virtual time diagram of a user</u>: a diagram in which the resource usage of the user is plotted as a function of his virtual time. The function has the value n if and only if resource #n is used. Remember that we made the assumption that a program uses one and only one resource at a time (it is a purely sequential process).

<u>Observables of the system</u>: any quantities which are relevant to our study. For instance, we are interested in:

The effective progress rate w_i of job #i (w like working). It is the portion of time user #i was working, divided by the total real time interval on which this was measured.

$$w_i = \frac{\text{total virtual time interval for \#i}}{\text{total corresponding real time interval}}$$

The performance u_j of resource #j (or its proportion of usage).

$$u_j = \frac{\text{time resource \#j is used by any job}}{\text{total real time interval}}$$

w_i and u_j are both dimensionless variables, which are observed over a certain interval of real time.

The economic criterion of the system is another observable; we assume that it takes the form:

$$E = \sum_i c_i w_i$$

where c_i characterizes the urgency of user #i.

<u>Definition of the worst case</u>. The worst case relative to some assumtpions and a given microscheduling strategy is the lowest possible value of a chosen observable for all possible virtual time diagrams of the various users over a certain real time interval, subject to certain constraints. These constraints will generally tell which proportion of its virtual time a job is using a given resource.

A given set of observables defines a space. For a given microscheduling strategy and with some assumptions on the usage of resources by the jobs, we define the <u>attainable domain</u> (or "domain of certainty") as the set of points in the observable space which can <u>always</u> be reached if the system desires it, for any virtual time diagrams of the users which satisfy the given assumptions. This is a generalization of the worst case in a multidimensional space.

System of fixed priorities (with preemption) for each user and resource

In the first model which is being studied,

1) For each user we know, over a given period of time, which percentage of the various resources (for instance, C.P.U., drum, but <u>not</u> memory) he is going to use.

2) For each resource, there is a priority assigned to each user. For a given resource, these priorities are all different (the users are totally ordered with respect to each resource). This priority assignment will not be changed between two consecutive macrodecisions. If user #i requests the resource, he will get it either if the resource is currently idle, or if it is allocated to a user having lower priority for this particular resource (in which case the lower priority user will have to wait for further use of this resource).

Note that a user does not necessarily have the same priority for all resources.

Thus, we characterize the situation by two matrices nxm, where n is the number of users and m the number of resources:

a_{ij} = proportion of the virtual time of user i spent on resource j

p_{ij} = integer number representing the priority of user i for resource j.

$1 \leq i \leq n$

$1 \leq j \leq m$

$0 \leq a_{ij} \leq 1$

$\sum_j a_{ij} = 1$ (normalization of the a_{ij} for each user)

$p_{ij} = p_{kj} \iff i = k$

$p_{ij} < p_{kj} \iff$ user i has a <u>higher</u> priority than k for resource j (Attention!)

The assumptions are given over a real-time interval [0,T], which separates the two activations of the macroscheduling algorithm. If w_i is the progress rate of user i, this user will effectively get resource j allocated during a time

$$T\, w_i\, a_{ij}$$

Resource j will be running during a total amount of time

$$T\, u_j = \sum_i T\, w_i\, a_{ij}$$

thus:

$$u_j = \sum_i a_{ij}\, w_i \qquad (0 \leq u_j \leq 1) \tag{1}$$

and, as a consequence of $\sum_j a_{ij} = 1$, we have: $\sum_i w_i = \sum_j u_j$

<u>Fundamental equations and consequences</u>

We are now interested in finding the attainable domain in the worst possible cases, where the requests are synchronized in an order such as to get the least possible simultaneous use of the resources available. This is relevant to the general philosophy that the system should always expect the highest amount of conflict within certain computed bounds. It should not oversell itself to the users, guaranteeing them a service that it would eventually not be able to give. Even if the system would decide to take some chances for a greater expected profit, probabilistic models would be dangerous because they assume a randomnes and absence of correlation between users which are not true generally. Also, for a given user, the requests do not have a random length under some distribution, and are not uncorrelated with each other. Of course, the computer could compute Markov-chain coefficients for the various users, but this seems to exceed the allowable overhead of an allocator.

The following fundamental equations express that, in the worst possible case, a process will be waiting for a resource at any time when this resource is used by another user of higher priority:

$$\text{if} \quad \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj} w_k < 1 \quad \text{then} \quad 1-w_i \geq \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj} w_k \quad \text{else } w_i = 0$$

$$(0 \leq w_i \leq 1) \qquad (1 \leq i \leq n)$$

Note that $1 - w_i$ is the rate of waiting, for user i. The previous equations can be condensed:

$$(2) \qquad 1 - w_i \geq \min \left(\sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj} w_k, \quad 1 \right) \qquad \begin{matrix} (0 \leq w_i \leq 1) \\ (1 \leq i \leq n) \end{matrix}$$

Equations (2) define a domain of values for the w_i 's; any point within this domain can always be reached if the system wants it. We called it the attainable domain, or synonymously the domain of certainty. Note that equations (2) imply that:

$$(3) \qquad 0 \leq u_j = \sum_i a_{ij} w_i \leq 1 \qquad (0 \leq w_i \leq 1) \quad (1 \leq j \leq m)$$

(The reader will find this result easy to prove). Equations (2) can be rewritten in the following more practical form:

$$(4) \qquad \forall \; i, \; \underline{\text{either}} \;\; w_i = 0 \;\; \underline{\text{or}} \;\; 1 \geq w_i + \sum_{\substack{k,j \\ p_{kj} < p_{ij}}} a_{kj} w_k \qquad (w_i \geq 0)$$

We are <u>not</u> going to prove here that (4) defines the worst domain

If a point $(w_1, w_2, \ldots, w_n)$ satisfies equations (4), then the system is assured of getting the corresponding job progress rates by taking the following microscheduling strategies:

- If a job asks for resource j with a priority greater than the one of the job currently allocated, preemption takes place.

- If job i has already used resource j during a time greater than $T\ w_i\ a_{ij}$, its priority for resource j is reduced to something lower than the priority of any job which has not exceeded its estimates on resource j. This policy protects other users against those having made wrong estimates on their a_{ij}'s.

The macroscheduler now has the task of computing the w_i's and the p_{ij}'s, to satisfy equations (4) and to optimize some criterion, for instance:

$$\text{(5)} \qquad \text{maximize} \quad E = \sum_i c_i\ w_i$$

This criterion is equivalent to a criterion which would tend to maximize resource usage:

$$E = \sum_i c_i\ w_i = \sum_j d_j\ u_j$$

$$\text{with:} \quad c_i = \sum_j a_{ij}\ d_j$$

where the d_j's could be called <<weights>> or costs of the resources.

<u>Example 1</u>

$$(a_{ij}) = \begin{pmatrix} .8 & .2 \\ .4 & .6 \end{pmatrix}$$

In this array, a job is in a row and a resource is a column. Resource 1 is the C.P.U., resource 2 is the disk; job 1 is compute bound, job 2 is more I/O bound.

1) We give the maximum priority to job 1 for all resources:

$$(p_{ij}) = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}$$

Equations:

$$1 \geq w_1$$

$$1 \geq w_1 + w_2$$

The domain of certainty is defined by: $w_1 + w_2 \leq 1$, which shows that no parallelism in the use of the resources is obtained in the worst case.

2) The priority matrix is:

$$(p_{ij}) = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Equations defining the attainable domain:

$$\begin{cases} 1 \geq w_1 + .4\ w_2 & w_1 \geq 0 \\ 1 \geq .2\ w_1 + w_2 & w_2 \geq 0 \end{cases}$$

The attainable domain corresponds to a nearly optimal usage of the resources:

$$u_1 = .8\ w_1 + .4\ w_2$$

$$u_2 = .2\ w_1 + .6\ w_2$$

U_1 and u_2 are maximum at the point:

$$w_1 = .65,\ w_2 = .87 \Rightarrow u_1 = .87,\ u_2 = .65$$

In this case, both u_1 and u_2 are maximum at the same point. This, however, is not a general result, and we might get very complicated domains in the u_i space. Nevertheless, solving the equations:

$$\forall i, \quad 1-w_i = \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj}\ w_k$$

might give a good approximation of the use of resources in the attainable domain.

3) The next case has the priority matrix:

$$(p_{ij}) = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

$$1 \geq w_1 + .6\ w_2$$

$$1 \geq .8\ w_1 + w_2$$

Obviously, the attainable domain is worse than with the second priority assignment but better than with the first one.

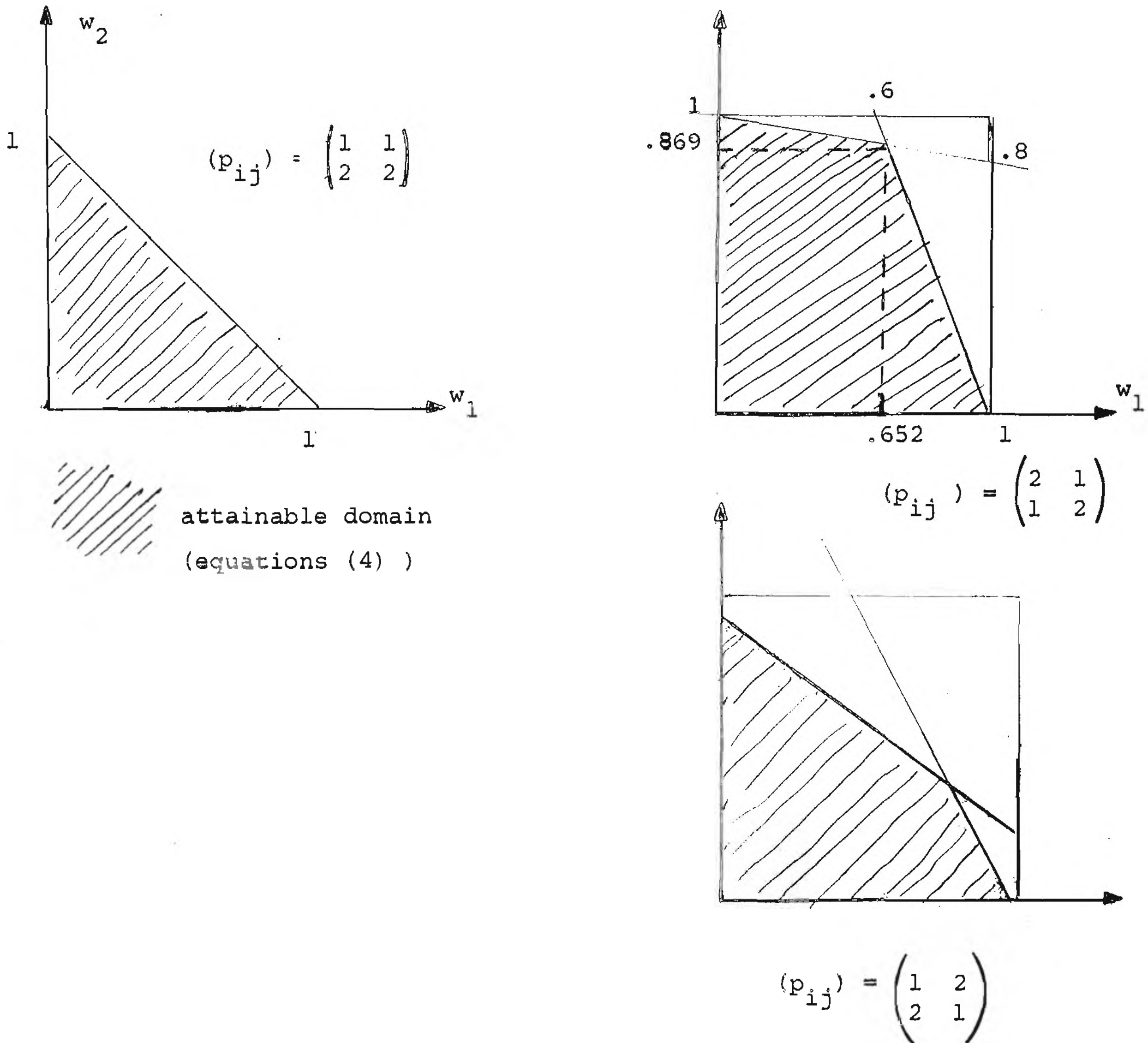

Fig. 1: Domains in the (w_1, w_2) space for example 1.

Example 2

Consider the case (2 jobs, 2 resources) where

$$(a_{ij}) = \begin{pmatrix} .5 & .5 \\ .5 & .5 \end{pmatrix}$$

Figure 2 shows the virtual time diagrams of the users, and how they combine into real-time diagrams for 2 different priority assignments:

$$(p_{ij}) = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} \Rightarrow \qquad 1 \geq w_1 + w_2$$

Figure 2.3 shows that this assignment really leads to almost sequential use of the resources, as expected from the model.

$$(p_{ij}) = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} = \begin{cases} 1 \geq w_1 + .5\, w_2 \\ 1 \geq .5\, w_1 + w_2 \end{cases}$$

The expected result $w_1 = .67$, $w_2 = .67$ is illustrated on figure 2.4.

Example 1 has a larger attainable domain than example 2 with the priority assignment $\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$. This is due to the fact that the jobs of example 1 are complementing each other (one needs more C.P.U., the other more I/O), while jobs of example 2 have identical average needs of resources.

We will now show that the assignment to each job of a set of the same priorities for all resources is a wrong choice, which can always be improved. For instance, we have the following theorem:

Theorem 1. If, under priority assignment #1 job i has a strictly higher priority than job i' on all resources, then, if for a particular resource j, there is no job having a priority higher than i' but smaller than i, $(p_{i'j} = p_{ij} + 1)$, then an exchange of the priorities of job i and job i' for resource j, without changing the other priorities, is a priority assignment #2 whose attainable domain strictly includes the attainable domain of priority assignment #1.

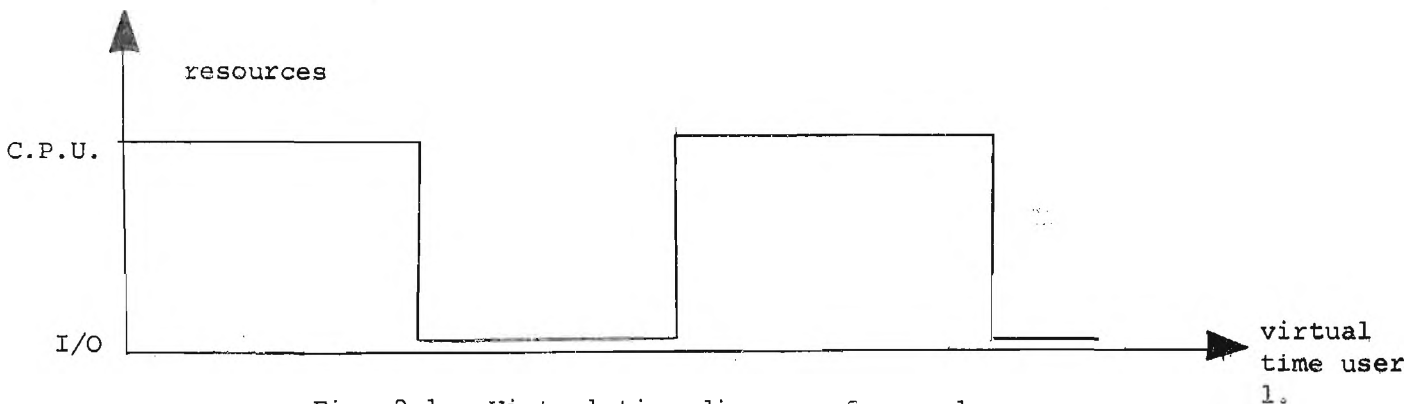

Fig. 2.1: Virtual time diagram of user 1.

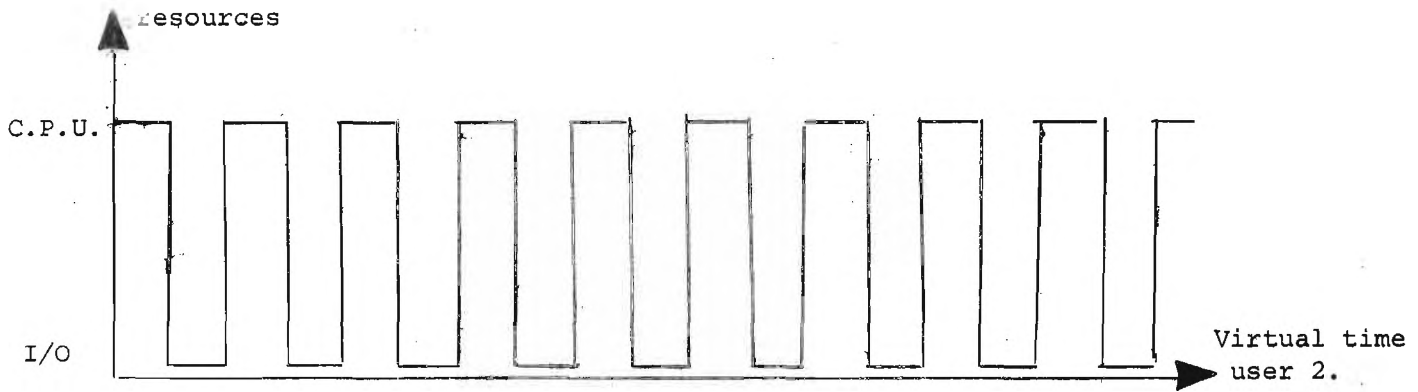

Fig. 2.2: Virtual time diagram of user 2.

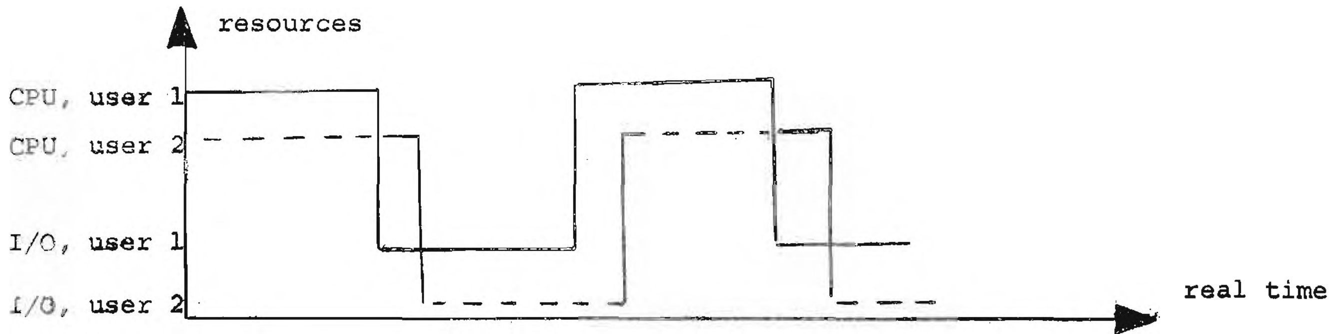

Fig. 2.3: Real time diagram; $(p_{ij}) = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}$

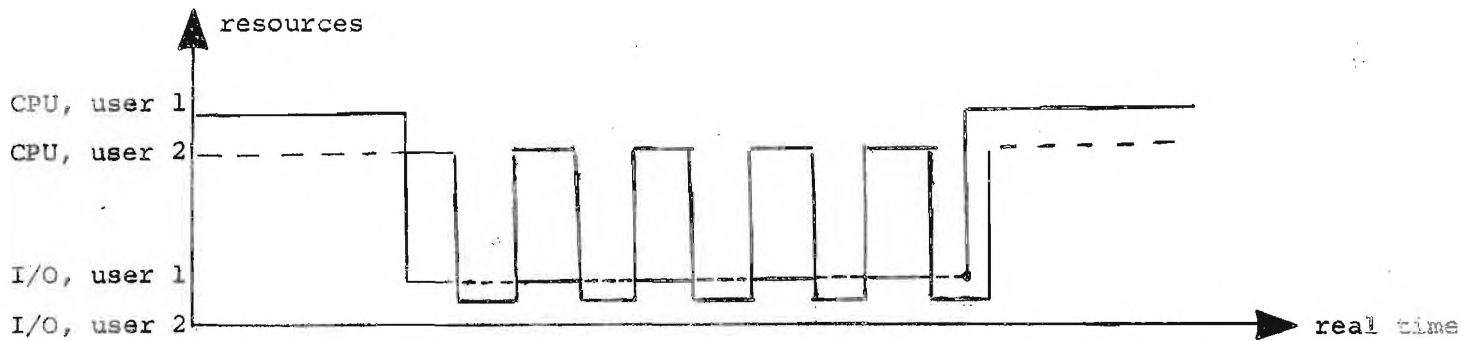

Fig. 2.4: Real time diagram; $(p_{ij}) = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$

------ user waiting for the resource

______ users seizing the resource

In other words, any point in the space of observables which satisfies equations (4) under assignment #1, will satisfy (4) under priority assignment #2.

The proof is given in appendix A. The theorem is quite weak, but at least it shows that a priority assignment like the following can certainly be improved:

$$(p_{ij}) = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{pmatrix}$$

How to assign priorities.

If we want to optimize $\sum_i c_i w_i$, we get a good priority assignment by ordering the priorities for resource j in the order of the quantities $\frac{a_{ij}}{c_i}$ (the smallest $\frac{a_{ij}}{c_i}$ gets the highest priority).

$$p_{ij} < p_{i'j} \Leftrightarrow \frac{a_{ij}}{c_i} < \frac{a_{i'j}}{c_{i'}}$$

This is however not an optimal priority assignment, as might be shown by the following counter-example:

Example 3: 3 resources, 2 jobs.

$$(a_{ij}) = \begin{pmatrix} .3 & .3 & .4 \\ .31 & .6 & .09 \end{pmatrix}$$

We want to optimize $w_1 + w_2$; our method leads to the following assignment of priorities:

$$(p_{ij}) = \begin{pmatrix} 1 & 1 & 2 \\ 2 & 2 & 1 \end{pmatrix}$$

The optimum is $w_1 + w_2 = 1.384$. However, with the priority assignment:

$$(p_{ij}) = \begin{pmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

the optimum would be: $w_1 + w_2 = 1.477$

This example clearly shows that our "good" priority assignment is not always optimal; one of its major advantages is simplicity.

We shall restrict ourselves in the following to the optimization criteria

$$E = w_1 + \dots + w_n = u_1 + \dots + u_m.$$

This is not a restrictive hypothesis for the validity of the theory, but simplifies the equations.

Multiprocessor case.

So far we have only considered the case where the resources are not interchangeable, and are only susceptible to one activation at a time. We are now going to study how the previous model can be extended to the case where some resources may have more than one activation at a time. For instance, there might be several identical CPU's or identical channels.

The fundamental "worst case" equations are quite complicated. We give them here without further justification.

Define q_{rj} by: $r = p_{ij} \Leftrightarrow i = q_{rj}$ $\quad r \in [1,n],\ j \in [1,m]$

q_{rj} is the number of the user having the r-th priority for resource j.

If resource j has R_j processors (possible simultaneous activations), the maximum time that user i would spend waiting for resource j in time interval [0,T] is:

$$\eta_{ij} = \begin{cases} \min\limits_{k \in [1,R_j]} \left(\dfrac{1}{R_j - k + 1} \sum\limits_{m \in [k,r-1]} a_{q_{mj}j} w_{q_{mj}} \right) & \text{if } r = q_{ij} > R_j \\ 0 \quad \text{if } q_{ij} \leq R_j \end{cases}$$

so that the equations are: $\forall i \in [1,n],\ 1 - w_i \geq \sum\limits_{j \in [1,m]} \eta_{ij}$

Theorem 2. A smaller domain than the attainable domain defined by the previous equations can be defined by equations (4) where a_{ij} has been replaced by $\alpha_{ij} = \dfrac{a_{ij}}{R_j}$

Proof: by choosing the first of the quantities whose minimum is η_{ij}, we have:

$$\eta_{ij} \leq \frac{1}{R_j} \sum_{i \in [1, r-1]} a_{ij} w_i$$

hence the theorem.

By replacing the a_{ij}'s by the α_{ij}'s, we have reduced the multiprocessor to

a monoprocessor case. There is another way to achieve this: suppose resource j to be a single resource (one activation only), but which works at R_j times its initial speed. The reader will easily verify that the w_i's would satisfy equations (4), where the a_{ij}'s would have been replaced by the β_{ij}'s given by:

$$\beta_{ij} = \frac{\frac{a_{ij}}{R_j}}{\left(\sum_k a_{ij} R_k\right)\left(\sum_k \frac{a_{ik}}{R_k}\right)}$$

The estimate given by the β_{ij}'s is slightly more optimistic than the bound obtained with the α_{ij}'s.

Example 4.

a_{ij}	CPU	Bus
job 1	.8	.2
job 2	.4	.6
job 3	.2	.8

Suppose that 2 C.P.U.'s are available. We assign the following priorities:

$$(p_{ij}) = \begin{pmatrix} 3 & 1 \\ 2 & 2 \\ 1 & 3 \end{pmatrix}$$

1) An exact treatment gives the fundamental equations:

$$1 \geq w_1 + \min (.2\ w_2 + .1\ w_3)$$

$$1 \geq .2\ w_1 + w_2$$

$$1 \geq .2\ w_1 + .6\ w_2 + w_3$$

yielding the solution (with equalities) :

$$\begin{cases} w_1 = .94 \qquad w_2 = [illegible] \qquad w_3 = .32 \\ \frac{1}{2} u_1 = .57 \qquad u_2 = .93 \end{cases}$$

2) The lower bound method:

α_{ij}	CPU	Bus
job 1	.4	.2
job 2	.2	.6
job 3	.1	.8

$w_1 = .80 \quad w_2\ .80 \quad w_3 = .36$

$\frac{1}{2}\ u_1 = .52 \quad u_2 = .93$

is the "best" point of the worst case.

3) The "optimistic" approximation:

β_{ij}	CPU	Bus
job 1	.372	.186
job 2	.18	.54
job 3	.093	.75

$W_1 = .81 \quad w_2 = .82 \quad w_3 = .40$

Note that the treatment with the β_{ij}'s does not in this example yield a higher value of $\sum_i w_i$ than the exact treatment with the a_{ij}'s. In other words, it is not always as efficient globally to have a processor working at speed n, as it is to have n identical processors working at speed 1. Of course, the processor at speed n has other advantages under other assumptions, if the memory size is limited.

Example 4 was intended to show that the estimate of the w_i's by the α_{ij}'s can be quite close to the exact computation results. This gets even more obvious with high numbers of jobs.

A first scheduling algorithm based on the preceding theory

Suppose that we know the coefficients a_{ij} for the various jobs. Those can be obtained either by extrapolating measurements conducted on each job since it started to run, or statistics on previous runs, or by using programmers' estimates We suppose that there is only one level of central memory, of which job i needs an amount m_i to run.

The jobs are ordered by external priorities. We then have to decide about the w_i's for the jobs; for instance, we might allow $w_i = .5$ for the high priority jobs. A w_i too close to 1 might strongly degrade the possible service for other jobs, by obliging the system to give a high internal priority for all resources to the job which has a high external priority. This would lead, as we saw, to a poor utilization of the resources.

If a set S of jobs is allocated in memory, it has to satisfy:

$$1 \geq w_i + \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj} w_k \qquad \forall i \in S$$

$$\sum_{i \in S} m_i \leq M \quad \text{(memory available)}$$

A procedure to find the maximal set of users fitting into available resources would take the following steps:

Step 1: Take the highest priority user; put him in set S'.
Step 2: Check whether S' is an allowable set: first assign the priorities p_{ij} $\forall i \in S'$, according to the rule

$$p_{ij} < p_{kj} \iff \frac{a_{ij}}{c_i} < \frac{a_{kj}}{c_k}$$

where the c_i's are in the same order as the external priorities. Then check equations (6) for set S'. If they all check, go to step 3 else go to step 4.

Step 3: $S \leftarrow S'$; go to step 4.
Step 4: Define S' as including all users of S, plus the highest priority user not yet handled. If there are no more users to handle, the algorithm stops, else go to step 1.

Using the above procedure we have found a maximum allowable set of users, each of which has a requested guaranteed service. The computations can be done so that the time required by the algorithm is: $t = A\,m\,n + B\,m\,n \log(n)$ the nxlog(n) term expresses the time to sort the quantities $\frac{a_{ij}}{c_i}$.

Check for inaccurate estimation of the resources needed

Having done our allocation for a period of time T, we have to check during

that period what amount of resource j job i is using. If he uses this resource during more than $w_i \; a_{ij} \; T$

job i is punished in the sense that its priority p_{ij} for this resource is changed to a priority lower than any job which has not exceeded its quantum on the resource. This method assures that a job which accurately estimated its needs will be served at least as well as promised.

Set of users with non-guaranteed service.

Assume that there is still some central memory available after having applied the previous macroscheduling algorithm. We might then put some other users in memory, with priorities lower (for each resource) than the lowest priority of the users of set S.The guaranteed service of users of set S will not be affected by these additional ("Marginal") users. M will be the set of marginal users.

We affect priorities in set M according to the same criteria as in set S. Of course, the resource usage will not be as good as if the priorities had been affected optimally for the entire set M + S. Our solution is a compromise between respecting the external priorities of the users, and increasing the system's efficiency.

Example 5. There are 2 CPU's but only one bus (or channel).

a_{ij}	CPU	I/O	w_i decided if allocated
job 1	.4	.6	.5
job 2	.3	.7	.5
job 3	.5	.5	.5
job 4	.9	.1	.5
job 5	.8	.2	.5
job 6	.6	.4	.5

These 6 candidates are in the order of their external priorities; there is no limitation because of memory in this example.

The reader can verify that the algorithm (with $c_i = 1, \forall i$) will accept jobs 1 and 2, reject 3, and accept 4 and 5. This is intuitively a good choice because 1, 2, 3 are I/O bound while the others are compute bound. $S = \{1,2,4,5\}$ $M = \{3,6\}$

and the priority assignment:

$$(p_{ij}) = \begin{pmatrix} 2 & 3 \\ 1 & 4 \\ 5 & 6 \\ 4 & 1 \\ 3 & 2 \\ 6 & 5 \end{pmatrix}$$

Note that job 3 would have been accepted if $w_3 \leq .4$. Our assignment gives the resource usage $u_{CPU} = .6$, $u_{I/O} = .8$, which could be improved by solving equations (4) (equalities) for the w_i's with the p_{ij}'s that we just computed.

Comments:

1) We only took into account the external priorities of the jobs, and not a more precise quantitative measure of their urgencies.

2) Clearly, if the w_i's were computed instead of just being arbitrarily decided before the algorithm started we could get to a more optimal solution.

We will now study a few models of allocation before returning to specific algorithms.

No-priority case.

In the "no-priority case", a user seizing a resource will never be preempted and will not lose the resource until he decides to release it. The situation is even worse than if users have the same priority for all resources, and may lead to almost no parallelism in the computations.

The worst case equations are:

$$(7) \quad \forall i \in [1,n] \qquad 1 \geq w_i + \sum_{\substack{k \neq i \\ j}} a_{kj} w_k$$

In the situation of example #4, this gives the following progress rates:

$$w_1 = .16, \; w_2 = .32, \; w_3 = .64$$

so that the overlap of activity is small:

$$\text{overlap} = w_1 + w_2 + w_3 - 1 \approx 12\%$$

Therefore the no-priority case is uninteresting, and should be avoided in any actual system design.

"Randomly turning" priorities

We will now investigate the following microscheduling algorithm:

The time is divided into very short intervals, and the priority of the users for the various resources is changing from one interval to the other, cycling so that each user spends the same amount of time in each priority level. Typically, the time between two priority changes might be 100 microseconds, and is small compared to the interval between two allocation requests of jobs to the microscheduler. Nevertheless, we assume that this method does not introduce any

additional overhead.

We might, for instance, use a random number generator, at the beginning of each time interval, to generate the job priorities during this interval; this would insure that there is no regular pattern of one job spending most of the time at a higher priority than another, as happens with a circular permutation.

The idea of such a microscheduling algorithm has the following justifications:

1) The hardware could allow time-sharing of a C.P.U. or a channel on very short time-slices. However, we don't know whether this would be a good practice.

2) We want to assure a user of a certain percentage of use of some resources, under any circumstances. Time-slicing on a very short time basis might seem a natural way to do it; if user i is assured of having the top priority on resource j during a portion of time $p_j\ \Delta T$ where ΔT is some small interval of time, then, with the a_{ij}'s defined previously, his progress rate will be at least

$$w_i = \min \left(\frac{p_j}{a_{ij}}\right)$$

However, we can compute a much higher "lower bound" estimate for the w_i's. After having done it, we will compare these new "worst case" equations to equations (4) and show that under some assumptions the "turning priorities microscheduling" performs poorer than a fixed priority algorithm with the p_{ij}'s well chosen. This result has been checked by simulation, and the following discussion attempts to establish a theoretical justification.

Under our new model, if k users compete for some resource, each one will get it during a portion of the time 1/k. Consider resource j. User i will seize it during a period $T\ a_{ij} w_i$. In the worst possible case, the maximum overlap of requests occurs on resource j. Thus, the time spent by user i waiting for resource j is less than or equal to

$$\sum_{k \neq i} \min\ (a_{kj} w_k\ T,\ a_{ij}\ w_i\ T)$$

This points out that if a job k asks for less time on resource j than job i, the maximum time spent by job i waiting for resource j because of job k will be

$T\,a_{kj}\,w_k$. If, on the other hand, $a_{ij}w_iT \leqslant a_{kj}w_kT$, job i will wait for resource j because of job k at most during a time $a_{ij}w_iT$.

The worst case equations are thus:

$$(8) \quad 1 \geq w_i + \sum_{\substack{k \neq i \\ j}} \min\,(a_{kj}w_k,\ a_{ij}w_i) \qquad \forall\, i \in [1,n]$$

These equations define the attainable domain with turning priorities.

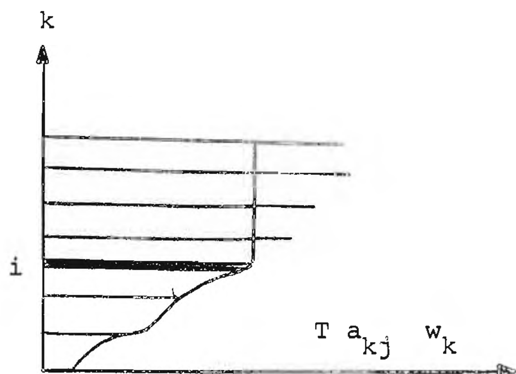

Fig. 3: Time spent by the jobs on resource j
and maximum interference of job i with other jobs.

Theorem 3. For every point in the attainable domain defined by equations (8), there exists a priority system in which this point is attainable according to equations (4).

Proof: We define this priority system by:

$$p_{ij} < p_{kj} \quad \Leftrightarrow \quad a_{ij}\,w_i < a_{kj}\,w_k$$

We assume that, for a given j, the a_{ij}'s are all different. Then obviously, equations (8) imply equations (4) for this system, which are:

$$\forall\, i \in [1,n] \qquad 1 \geq w_i + \sum_{\substack{k \neq i \\ a_{kj}w_k < a_{ij}w_i}} a_{kj}\,w_k$$

This theorem is reassuring because it tells us that whatever we are assured of doing under a turning priority system, we are also assured of doing under a fixed priority system.

However, we can prove the following theorem under some restrictive assumptions:

Theorem 4.

If one of the following is true:

1) There are only 2 jobs (and any number of resources).
2) There is any number of jobs, but competition is limited to one resource only.

Then there exists a priority system whose attainable domain includes the domain defined by:

$$p_{ij} < p_{kj} \Leftrightarrow a_{ij} < a_{kj}$$

The proof is shown in appendix B.

Theorems 3 and 4 show that a fixed priority system should, to a certain extent, be preferred to a random priority system (which is itself better than no priority at all). If a resource has the property that it can be preempted without any other additional future loss of time, then the available information on the jobs can be used to assign priorities for the resources, and a "good" choice is to assign the resource to the job which has the least need for it (after having weighted these needs by the external urgencies of the jobs, which leads to the quantities $\frac{a_{ij}}{c_i}$).

What about macroscheduling?

Do we really need macroscheduling separated from microscheduling? In a recent paper [2], the author examines what was wrong with the Chippewa Operating System; he concludes that there were two flaws. First, the absence of a macro-scheduler: the Chippewa system allocated resources for an indefinite period of time, without taking into account the global demand of each job. Thus, there was no guarantee when a job was allocated, that the job would not ask later for more memory than was available, and in this case the Chippewa scheduler did not take back the resources (CPU,...) already allocated. The second problem of Chippewa was that I/O bound jobs, or compute bound jobs, were not recognized as such by the scheduler, and so this information was not taken into account in assigning priorities for the resources. We have seen that bether simultaneity in resource usage is achieved by assigning a high priority for a resource to a job which makes little use of this resource.

Denning [3] has attacked the macroscheduling problem by assuming that each user's processor demand and memory demand are known. Then the macroscheduler has only to solve the Knapsack problem (deciding, for each job, whether to allocate it or not, so that the sum of the demands of each job for some resource will not exceed a quantity which is slightly smaller than the total available amount of this resource.

Our approach has some similarities to Denning's, however we do not only determine for a job whether it will be allocated or not, but also what will be its w_i (progress rate). By doing so, we hope to get better resource usage than if the users were deciding individually about their own progress rates (or if the "external" scheduler was deciding them).

<u>Sketch of another algorithm for macroscheduling</u>

There are two kinds of resources: the macroresources $\mathcal{B}$ (which are allocated by the macroscheduler), and the microresources $\mathcal{a}$ (allocated by the microscheduler) We call b_{ij} the <u>absolute</u> amount of macroresource j desired by user i. By opposition, a_{ij} is the <u>relative</u> amount (per unit of time) of microresource j needed by user i.

Now, the mathematical problem can be expressed in the following way:

Find a set of users S which maximizes the economic criterion

$$E = \sum_{i \in S} c_i \, w_i$$

subject to the following constraints:

$$(9) \begin{cases} \forall_j \in \mathcal{B}, \quad \sum_{i \in S} b_{ij} \, \delta_i \leq B_j \\ \qquad \text{where} \begin{cases} \delta_i = 0 & \text{if } w_i = 0 \\ \delta_i = 1 & \text{if } w_i > 0 \end{cases} \\ \forall_i \in S, \quad w_i + \sum_{\substack{j \in a \\ p_{i'j} < p_{ij}}} a_{i'j} w_{i'} \leq 1 \end{cases}$$

where p_{ij} ia a priority system to be chosen.

The first equations for $j \in \mathcal{B}$, express that any user which has a non-zero progress rate (and thus who is allocated in memory), can find some space in memory level #j.

The other equations are just equations (4).

The following algorithm finds a nearly optimal solution to the problem. It works in two steps:

1) Get an approximate solution by optimizing the economic criterion with the following constraints:

$$(10) \quad \begin{cases} \forall j \in \mathcal{B} \quad \sum_{i \in S} \frac{b_{ij}}{B_j} \leq 1 \\ \forall j \in \mathcal{A} \quad \sum_{i \in S} a_{ij} \, w_i \leq 1 \end{cases}$$

The constraints for the resources of set $\mathcal{B}$ are identical in equations (9) and (10); the constraints concerning the resources of set A are, however, weaker in equations (9) than in equations (10); the latter just express the best possible case (where no unnecessary interference between jobs would happen); however, we use this method because equations (10) are easier to manipulate than equations (9) and we shall later get a more refined solution. This first step is essentially intended to eliminate from further consideration the jobs which should certainly not be scheduled (for which $w_i = 0$).

To get a good approximate solution of this mixed linear programming problem (10), it is not necessary to use the Simplex method, nor an enumerative method of search. A faster method which gives a good approximate solution works as follows:

Assign an initial weight K_j to resource j. Assign an initial w_i to job i. Compute the desirability for each job:

$$d_i = \frac{c_i}{\sum_{j \in \mathcal{B}} b_{ij} K_j + \sum_{j \in \mathcal{A}} a_{ij} w_i K_j}$$

Sort the jobs according to their desirabilities. Starting with the one of highest desirability, compute whether the job can be allocated or not, that is,

if equations (10) can be satisfied with S consisting of the jobs which we already decided to allocate and of the job we are trying to allocate. Whether the job has been allocated or not, try the next one.

When all the jobs have been examined, compute a new weight assignment and the new w_i's according to the principle that a job having larger d_i should have a larger w_i, and that a resource for which the corresponding equation (10) had its left side much smaller than 1, should have its weight decreased.

This entire process can be repeated 2 or 3 times.

2) Having determined the set S, we get a better approximation of the w_i's by solving equations (11), with $p_{ij} < p_{i'j} \Leftrightarrow a_{ij}/c_i < a_{i'j}/c_{i'}$

$$(11) \qquad 1 = w_i + \sum_{\substack{j \in \alpha \\ p_{i'j} < p_{ij}}} a_{i'j} w_{i'} \qquad \forall\, i \in S$$

If any of the w_i's of the solution is negative, this w_i is removed from set S, and equations (11) are solved again. As we have seen, equations (10) were giving a set of users to allocate which could be somewhat too large. By eliminating some users from this set in some cases, we get a nearly optimal set to satisfy equations (9) while optimizing equation (5).

Pricing

The determination of prices is, to a large extent, a consequence of the scheduling strategy. In our approach, a user agreed to pay at most a price $c_i w_i$ to get a progress rate w_i, and if he was proposing a larger c_i he did get a higher priority.

However, the system should charge more or less uniformly the various jobs being allocated; it should not just charge the maximum possible to each job, because otherwise the jobs would increase their c_i's slowly until they were scheduled, thus leading to a greater overhead. The marginal theory of pricing requires us theoretically to charge to user i exactly $c'_i w_i$, where $c'_i < c_i$ is the lowest bid that the user would have had to offer to get allocated; unfortunately, this definition would lead to very complicated computations. We suggest here a few alternative methods.

1) If l is the first job which was skipped (not allocated) when we scanned the jobs in order of decreasing desirabilities in the first step of our macroscheduling algorithm, and if w_i is the effective progress rate for job i, charge job i:

$$p_i = \min(c_l, c_i) \times w_i \times T$$

2) If we want to penalize jobs having estimated their a_{ij}'s incorrectly, and if job i has effectively used an amount r_{ij} of resource j, we charge him:

$$p_i = \min(c_l, c_i) \times \max\left(\frac{r_{ij}}{a_{ij}}\right)$$

3) We could have also computed a unit cost for resource j:

$\mu_j = K_j d_l$, where l is the first job not allocated and K_j the weight of resource j, as computed by the macroscheduling algorithm. If job i uses resource j during a time r_{ij}, we could let him pay:

$$p_i = \min(c_i w_i, \sum_j r_{ij}\mu_j)$$

It is useful to have some prices for resources, so that

1) A new coming user can by immediate inspection of the prices determine whether he wants to get in the system or not.

2) On the long range, the computer center staff might determine the needs to install or remove facilities (see [5])

The variations of the μ_j,s in time should probably be smoothed for those purposes.

Problem for further research:

1) Continuous macroscheduling: instead of applying the macroalgorithm at regular time intervals, find a simplified macroalgorithm to be applied each time a job previously running deactivates itself voluntarily, or when a job is changing its external priority, or even when the swapping channel is idle. We might just schedule or unschedule jobs using the desirabilities which have been already computed, but we might also want to recompute the p_{ij}'s, the K_j's, the d_i's and the w_i's.

2) Extend the models to include processes using more than one resource at a time. For instance, Fig. 4 shows the virtual time diagram of a user

which initiates I/O and swapping at the same time:

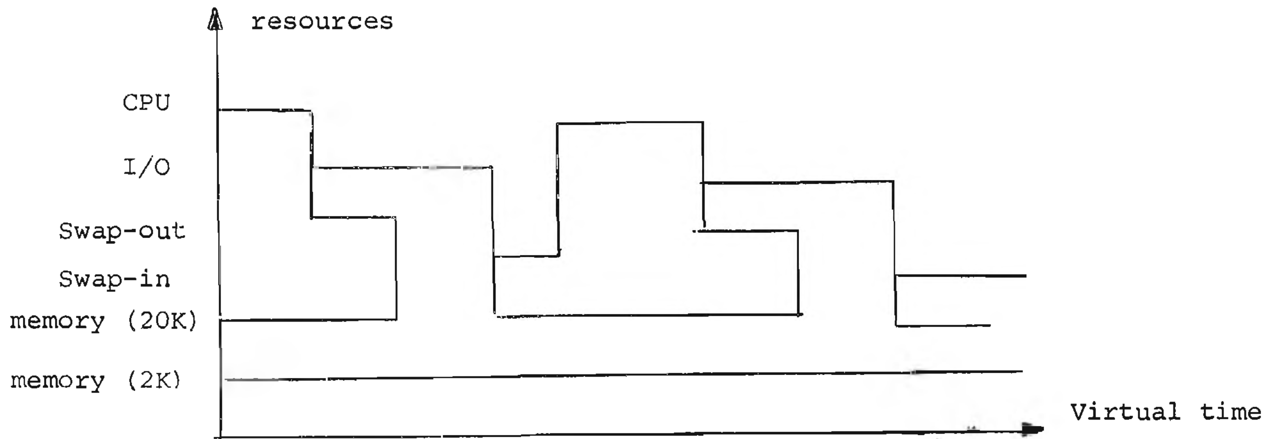

Figure 4

Another characteristic of our hypothetical job is that it does not need all its memory resource continuously (a buffer of 2K is enough during I/O completion). Could we take care of this knowledge?

Solving this problem would be especially useful for future computer systems where the cost of arithmetic and control units is expected to decrease much more than the cost of central memories.

3) Find models of "probable" performance as well as "worst case" models.

4) Which information other than a_{ij}'s or the b_{ij}'s on the jobs would be relevant to an allocation algorithm?

For instance, the exact virtual time at which a job will place a request might be available for some jobs while being completely out of the question for others.

5) How much would the results of the model be affected by slight errors in the predictions?

Conclusion

Our initial effort was applied to separate problems which are usually handled together in a very intricate manner: 1) Scheculing 2) Paging algorithms 3) Deciding external priorities of users 4) Collecting information about the average probable needs for resources of a specific job. Pricing, at the opposite, should not be a question separated from scheduling. The problems of protection

and of deadly embrace had already been separated from the others in previous works. By partitioning the difficulty, we believe that the way to better scientific understanding of shared computer systems stands open.

The previous scheduling algorithms and models apply in computer systems where the shared facilities can either be preempted with very little overhead (CPU, busses between two levels of fast memory), or cannot be reallocated without a great amount of overhead (memory). They do not apply, however, in cases where a resource can be preempted but the delay imposed on the preempted job is greater than the time during which the preemption occured. This would be the case if, for instance, a job is swapped from the drum into memory, but if at a certain moment he can't get one of the pages because another job has a higher priority to get a page from this sector of the drum, then our preempted job will have to wait an entire revolution of the drum before the opportunity to get the missing page is repeated, and the cost of having a set of pages idling in memory during all that time is of course important. In such a case, the right strategy might be to avoid preemption, and to decide what to do by computing a "desirability ratio" for each possible scheduling operation (ratio of the urgency by the total cost of the resources involved).

It is our belief that the scheduling techniques described in this paper will be especially useful for scheduling of real-time users, who want to have the assurance of getting a certain percentage of usage of the resources of the machine before they start working.

Other investigations of multileveled scheduling are still necessary. We believe that queuing theory is getting too rapidly enormously complicated when the number of servers and the complexity of the queueing strategy increase. Simulation is a fast way of testing whether some algorithm is workable, but is not more than a predictive technique: it does not seem to be likely in the future that a scheduler will first simulate the situation before making a decision. Analytical approaches are almost all that are left to prove schedulers in the future with the certainty that the designed algorithm will work almost optimally in all cases.

Acknowledgements.

The author thanks specially Dr. David C. Evans, of the University of Utah, who has patiently directed his research. He also thanks Denis D. Seror for many enlightning discussions, and Duane B. Call, who helped to edit the paper. And, last but not least, Marcella Sanchez who typed it.

References

[1] LASSER, Daniel J. - Productivity of Multiprogrammed Computers - Progress in developing an Analytic Prediction Method - CACM 12,12 (Dec. 1969) PP 678-684

[2] STEVENS, David F. - On overcoming High-Priority Paralysis in Multiprogramming Systems, a Case History. CACM 11,8 (August 1968), pp 539 - 541.

[3] DENNING, Peter J. - Resource Allocation in Multiprocess Computer Systems - Ph.D. Thesis, Project MAC, M.I.T., May 1968.

[4] HABERMAN, A. N. - On the Harmonious Co-operation of Abstract Machines. Technische Hogeschool, Eindhoven, October 1967.

[5] NIELSEN, Norman R. - Flexible Pricing: An approach to the allocation of computer resources, AFIPS F.J.C.C. 1968/ pp 521-531.

Appendix A. Proof of Theorem 1.

Equations (4) can be written, under priority assignment #1:

(12) $$\forall k \in [1,n], \text{ either } \sum_{k'=1}^{n} \gamma_{kk'} w_{k'} \leq 1 \text{ or } w_k = 0$$

$$\text{with } \gamma_{kk'} = \sum_{\substack{\rho \in [1,m] \\ p_{k'\rho} < p_{k\rho}}} a_{k'\rho} \quad \text{for } k \neq k', \text{ and } \gamma_{kk} = 1$$

Under priority assignment #2:

(13) $$\forall k \in [1,n], \text{ either } \sum_{k'=1}^{n} \gamma'_{kk'} w_{k'} \leq 1 \text{ or } w_k = 0$$

$$\text{with } \gamma'_{kk'} = \sum_{\substack{\rho \in [1,m] \\ p'_{k'\rho} < p'_{k\rho}}} a_{k'\rho} \quad \text{for } k \neq k', \text{ and } \gamma'_{kk} = 1$$

We shall prove that equation #k of (12) implies equation #k of (13), except for k = i, and that equation #i' of (12) implies equation #i of (13).

1) Note that for $k \neq i$ and $k \neq i'$, we have obviously $\gamma_{kk'} = \gamma'_{kk'}$, because the priorities of these jobs relative to either job i or job i' have not been modified on any resource. This proves equation #k of (13) for $k \neq i$ and $k \neq i'$.

2) We have: $\gamma_{ii} = \gamma_{i'i'} = \gamma'_{ii} = \gamma'_{i'i'} = 1$

$$\gamma_{ii'} = 0, \quad \gamma_{i'i} = 1, \quad \gamma'_{ii'} = a_{ij} \leq 1, \quad \gamma'_{i'i} = a_{ij} \leq 1$$

thus: $\gamma'_{i'k'} \leq \gamma_{i'k'}$ for any k'; this proves equation #i' of (13).

3) Finally, we show that equation #i of (13) is implied by equation #i' of (12). For $k' \neq i$ and $k' \neq i'$, $\gamma'_{ik'} \leq \gamma_{i'k'}$ because $p_{k'j} < p_{ij} \Rightarrow p'_{k'j} < p'_{i'j}$

$$\gamma'_{ii'} < 1 = \gamma_{i'i'}$$

$$\gamma'_{ii} = 1 = \gamma_{i'i}$$

QED.

Appendix B. *Proof of Theorem 4.*

1) There are only 2 jobs.

Before proving the theorem, let us prove the following lemma:

Lemma: if λ_{12}, λ_{21}, w_1 and w_2 are positive numbers less than 1, then equations

(14) $$(1 + \lambda_{12})\, w_1 + \lambda_{21}\, w_2 \leq 1$$

(15) $$\lambda_{12}\, w_1 + (1 + \lambda_{21})\, w_2 \leq 1$$

imply

(16) $$w_1 + (\lambda_{12} + \lambda_{21})\, w_2 \leq 1$$

Proof: equation (16) is achieved by multiplying equation (14) by $(1-\lambda_{12})$, equation (15) by λ_{12}, and adding.

Proof of the theorem: We assume that $\forall j,\ i \neq k \Leftrightarrow a_{ij} \neq a_{kj}$. We have to show that equations (8) imply equations (17):

(17) $$1 \geq w_i + \sum_{a_{kj} < a_{ij}} a_{kj} w_k$$

We can rewrite equations (8) as equations (14) and (15) with :

$$\lambda_{ij} = \sum_{\substack{k \\ a_{ik} w_i < a_{jk} w_j}} a_{ik}$$

According to the lemma, this implies equation (16); now note that

$$\sum_{\substack{k \\ a_{ik} < a_{jk}}} a_{ik} \leq \alpha_{ij} + \alpha_{ji}$$

so that equations (17) are verified.

2) There is only one resource.

We have to show that equations (18) imply equations (19):

(18) $$i \in [1,n] \quad w_i + \sum_{k \neq i} \min\,(a_i w_i,\ a_k w_k) \leq 1$$

(19) $$i \in [1,n] \quad w_i + \sum_{a_k < a_i} a_k\, w_k \leq 1$$

Consider the set S of jobs such that $k \in S \Leftrightarrow a_k w_k > a_i w_i$.

Does there exist a job $k' \in S$ such that $w_{k'} > w_i$?

a) yes, there does. Choose then k' such that there is no job in S whose progress ratio is less than $w_{k'}$ and greater than w_i. Then equation #k' of (18) implies equation #i of (19).

b) there is no such k'. This means that $\forall k \in S$, $w_k < w_i$, and $a_k > a_i$. Thus equation #i of (18) implies equation #i of (19).

3) Note that the theorem is not valid for any number of jobs and any number of resources, as shown by the following counter-example:

$$(a_{ij}) = \begin{pmatrix} .25 & .25 & .25 & .25 & & & & \\ .24 & & & & .76 & & & \\ & .24 & & & & .76 & & \\ & & .24 & & & & .76 & \\ & & & .24 & & & & .76 \end{pmatrix}$$

The progress rates $w_1 = .5$, $w_2 = w_3 = w_4 = w_5$ .875, satisfy equ. (8), but not equ. (4).

However, we suspect that the optimum of $\sum_i w_i$ is always higher with equ. (4) than with the constraints of equ. (8).