Multiprogram Scheduling / E. F. Codd / TR 00.716

IBM

Technical publication

April 29, 1960 TR 00.716

MULTIPROGRAM SCHEDULING

by

E. F. Codd

ABSTRACT

In order to fully exploit a fast computer which possesses simultaneous processing abilities, the computer should to a large extent schedule its own workload. The scheduling routine must be capable of extremely rapid execution if it is not to prove self-defeating.

The construction of a schedule entails determining which programs are to be run together and which sequentially with respect to each other. A concise scheduling algorithm is described which tends to minimize the time for executing the entire pending workload (or any, subset of it), subject to external constraints such as precedence, urgency, etc. The algorithm is applicable to a wide class of machines.

Product Development Laboratory, Data Systems Division
International Business Machines Corporation, Poughkeepsie, New York

MULTIPROGRAM SCHEDULING

by

E. F. Codd

PART I - INTRODUCTORY REMARKS

1.1 Scheduling Environment

Scheduling requirements are bound to vary a great deal from one computing installation to another. Factors which contribute to this variation are:

a. the nature of the jobs to be processed

b. the amount of surplus capacity which the equipment possesses in relation to the normal load of work

c. the pattern of arrival of workload (e.g., is it heavily peaked?)

d. operating and service policies

In spite of these variations, it seems worthwhile to attempt to define a scheduling algorithm which can form a common foundation upon which many installations may build by adding routines oriented toward their particular needs.

The reasonableness of such an objective is already recognized for large machines. It is doubly reinforced by the existence of machines capable of executing concurrently two or more (related or unrelated) programs. The complexity of scheduling work for such machines is far beyond the capabilities of the normal operating staff. Further, scheduling decisions must be made with great rapidity in order to be useful; human beings by themselves cannot make such decisions rapidly enough.

We consider first several concepts which appear to apply to numerous installations. Urgency is one of the most familiar of these concepts. At any particular time, it may be more urgent to process one job than some other. Each job would thus have an urgency rating generated in a manner dictated by a particular installation's own requirements. The interpretation, however, may be assumed to be fixed, providing the interpretation is sufficiently general. In the scheme proposed herein the urgency rating merely affects the order in which programs are considered by the scheduling algorithm. The time taken to process a given set of programs may be prolonged if it is stipulated that certain ones are to be given urgent attention. An additional way in which urgency can be handled will become apparent later.

Precedence is a second important concept. The completion of one program may be essential before some other can be started; results or output from the first become operands or input for the second. This kind of precedence which stems from causes external to the scheduling should be distinguished from precedence which is inevitably generated by the scheduling algorithm whenever a schedule is produced. Treatment of externally derived precedence will be described in Part 4 of this paper.

The arrival rate and consumption rate of work to be processed determine the extent to which a backlog of work builds up. This, in turn, places a limit on how far ahead one may schedule the machine. It is desirable that the scheduling algorithm be adaptable to extreme variations in the extent of the backlog, including the case of no backlog at all with in-flow of work at a mere trickle. For this reason, the scheduling algorithm can be executed in either of two modes. In the static mode, generation of a schedule is separated from its execution. Consequently, it is reasonable to consider generating a number of trial schedules, select the best one and then execute it. In the dynamic mode, the scheduling activity consists solely of augmenting an existing schedule; this activity is quite intimately interleaved with execution. The static mode is clearly applicable only when an adequate backlog of unscheduled work exists. The dynamic mode, on the other hand, may be applied in all situations.

Another kind of adaptability needed by a scheduling algorithm is the ability to cope with time information of varying degrees of reliability and, in the extreme case, with a complete absence of such information. In this connection, degraded versions of the algorithm will be discussed in Part 4.

1.2 Class of Machines

The algorithm is intended to be applicable to a wide class of machines, called multiple facility systems. Such a system consists of "a number of connected facilities, each capable of operating simultaneously (and, except for memory references, independently) on programs which need not be related to one another" (1). The sharing of time or space on a facility must be controllable in a flexible way to insure that the scheduling effort will be rewarding.

A multiple facility system is quite likely to contain several facilities of the same kind -- for example, two similar arithmetic units or several similar tape channels. A group of similar facilities is treated as a single composite facility by the scheduling algorithm. It is assumed that the work scheduled for a composite facility is subdivided by a separate and subsequent allocation algorithm. The special case of subdividing the work assigned to a set of tape channels and tape units is a sufficiently independent topic to warrant treatment in a separate paper.

1.3 Form of Programs

Programs are assumed to be fully relocatable prior to execution -- with no dependence on specific assignments of space (including input-output units), or necessary association with a specific member of a set of interchangeable time-shared facilities. However, programs are not required to be relocatable during execution.

Programs and data are not assumed to be segmented or organized into modules of any predetermined size. An installation may, however, superimpose such a policy; some types of machines may dictate it.

1.4 Explanatory Note

The remainder of this paper is divided into three parts. Part 2 deals with a formulation of the scheduling problem, ignoring for the time being the dynamic mode and external constraints such as precedence, urgency, etc. From this formulation, four types of lower bounds are derived for the execution time of the pending workload. These bounds are used as a basis for the algorithm (essentially empirical) described in Part 3. External constraints are introduced in Part 4 and the scheduling algorithm is extended to handle them.

PART II - THEORETICAL CONSIDERATIONS

2.1 The Problem

Suppose we are given a multiple-facility system equipped with two kinds of facilities: those which may be time-shared and those which may be space-shared. A set of programs is to be executed so as to minimize the time for the whole set.

Each program i takes time t_i to be executed when run alone, and for this length of time uses fractions s_{ik}, r_{ij}, respectively, of the space and time available on space facility k and time facility j.

Let us further suppose that the adjacency constraint applies to every space facility: that is, for every program i and space facility k, it is necessary that the space fraction s_{ik} be allocated so that program i occupies a set of adjacent locations on facility k. Normally, this constraint applies to core and disk storage but not to tape unit allocation.

Note that any subset of programs selected to be run concurrently must meet the requirement that the total space fraction used by the subset on each space facility does not exceed unity. The scheduling problem does not require that an upper bound be imposed on the total time fractions. Neither does it preclude such action. As we shall see, the procedure described herein treats space fractions and time fractions in a near-symmetric way, and an upper bound is imposed on total time fractions.

2.2 Accuracy of Scheduling Data

We may assume the space fractions s_{ik} to be exact. On the other hand, the time fractions r_{ij} and expected elapsed time t_i are very likely to be approximate. In supplying time information, the programmer may expect assistance from the supervisory system. Such assistance may take the form of time studies (made during a previous run) of selected sections of his program. For many programs it would be reasonable to expect the time fractions to be accurate at least to the nearest quarter.

Programs will often deviate from their anticipated elapsed times. Because overrunning may seriously upset a schedule, a limit is placed on the extent of overrun. When this limit is exceeded, the program in question is set aside for external supervisory consideration or treatment by an installation-oriented routine.

2.3 The Space Fraction Diagram

Consider space facility k . Take space fraction for this facility as ordinate and elapsed time as abscissa. Then, the space-time provided by this facility is represented by the rectangular domain (0, 0), (0, 1), (∞, 1), (∞, 0) which is shaded in Figure 1.

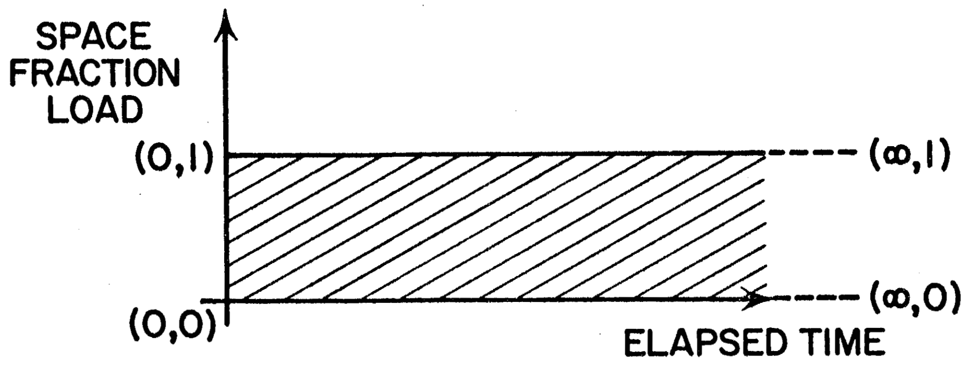

Fig. 1. Space Fraction Diagram.

The demand of program i for space-time on facility k is represented by a rectangle of height s_{ik} and base t_i . The corresponding rectangles for all programs of the given set are to be packed into the rectangular domain by a procedure which will be described later. An example of packing five programs on a single facility is illustrated in Figure 2.

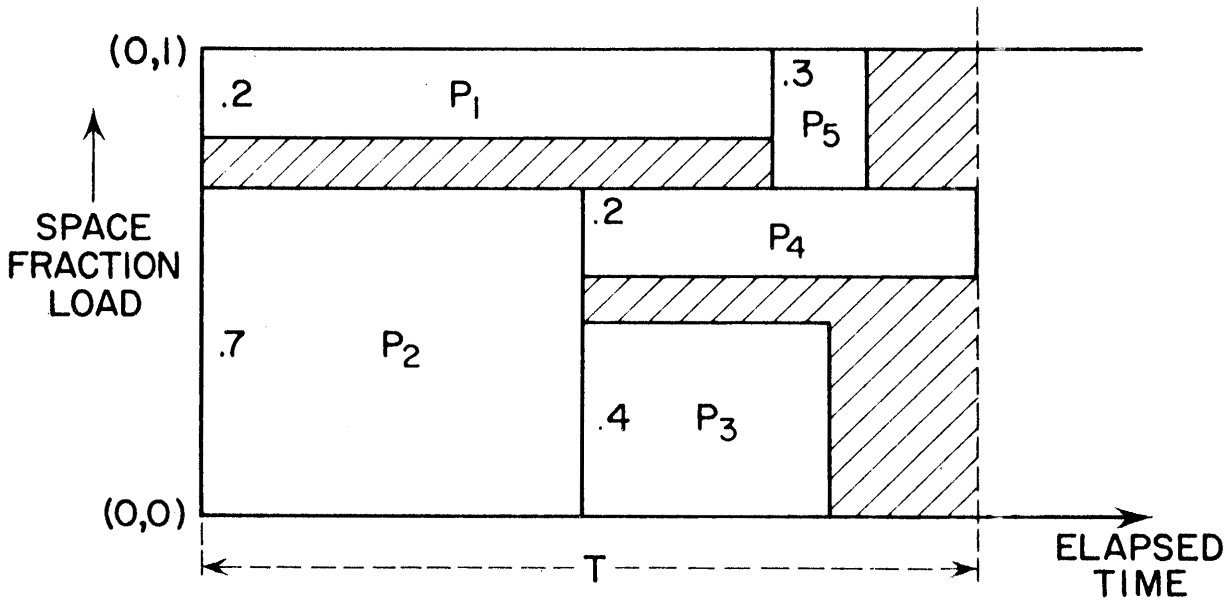

Fig. 2. Five Programs on a Space-Shared Facility.

Of course, the packing must be implemented simultaneously on all facilities. In other words, if space is allocated to program i on facility k between time t and $t + t_i$, then space must be allocated to this program on all other facilities between these same two times.

The abscissa T of the rightmost edge of the rectangle which extends farthest to the right (i.e., along the elapsed time axis) represents the least time in which the schedule represented by the diagram can be executed. Suppose that, instead of the loose packing indicated in the diagram, we have a perfectly tight packing into a rectangle of dimensions 1, v_k. Then, we may write, by summing areas,

$$v_k = \sum_i s_{ik} \; t_i$$

Clearly v_k is a lower bound for the execution of the given set of programs. There are as many lower bounds of this type as there are space facilities.

2.4 The Time Fraction Diagram

The time fraction loading of a time-shared facility may be represented by a diagram similar to the space fraction diagram for a space-shared facility. However, the total time fraction at any instant is limited only by the upper bound (if any) imposed by the scheduling procedure.

An example is given in Figure 3 of a time fraction diagram for 5 programs on facility j with start-stop times as indicated in the example of a space fraction diagram.

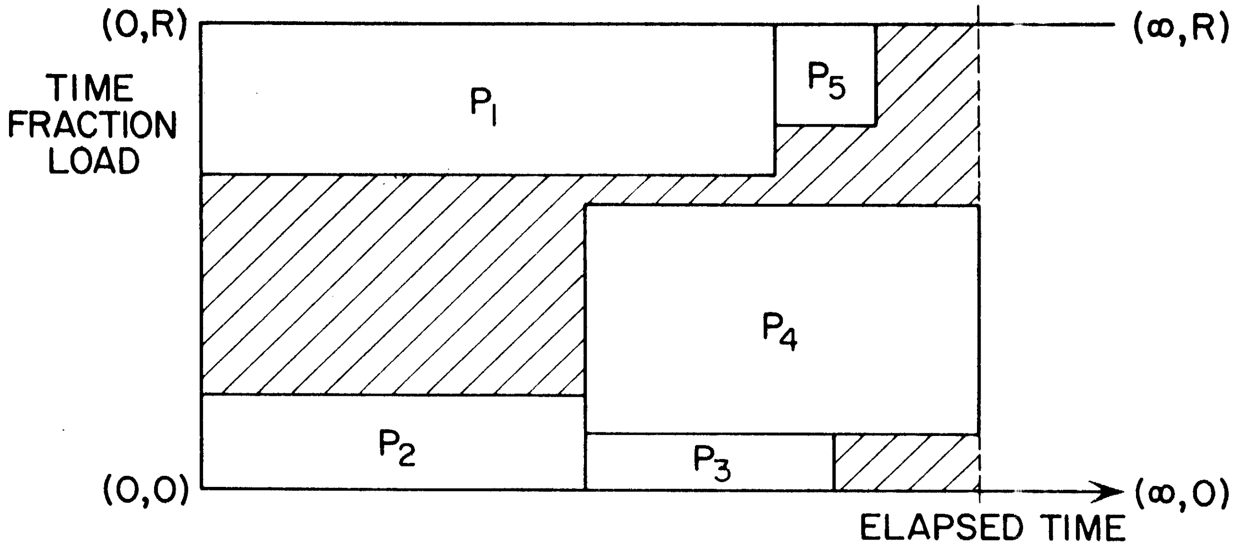

Fig. 3. Five Programs on a Time-Shared Facility.

Whenever the total time fraction on some facility exceeds unity, one or more of the programs using that facility will receive service from it at a reduced rate and the elapsed times of these programs will accordingly be extended.

Now program i requires of facility j an amount $r_{ij}t_i$ of service, regardless of the rate at which it receives this service. If this quantity is summed over all programs, we obtain a second type of lower bound for the execution time of the given programs:

$$u_j = \sum_i r_{ij} t_i$$

Again, there is a bound of this type for each time-shared facility. The space and time fraction diagrams are referred to collectively as the load diagrams.

2.5 Types of Mixes

The term mix is applied to a set of programs either planned to be in the execution phase together or actually being executed concurrently. A feasible mix is a set of programs for which the space fraction sum on each space-shared facility is less than or equal to unity.

The fact that a given mix is feasible does not imply that it can be executed efficiently -- that is, in less time than it would take to execute these programs one at a time. Consider, for example, a mix consisting of two programs, which for some time-shared facility have time fractions each equal to unity. Clearly, there is no time advantage in running these programs together; there may actually be a disadvantage due to the extra supervisory overhead incurred. A mix is said to be profitable if it is feasible and contains no pair of programs each of which demands a time fraction of unity on some common facility.

2.6 Pairwise Infeasible and Unprofitable Subsets

A pairwise infeasible subset consists entirely of programs, no two of which form a feasible mix. It is clear that programs belonging to a pairwise infeasible subset must be run one after the other. Hence, the sum of their elapsed times is a lower bound for the execution time of the whole set. Accordingly, there exists a lower bound of this type for every such subset.

An improved set of lower bounds is obtained by considering every pairwise unprofitable subset. Such a subset consists entirely of programs, no two of which form a profitable mix. For every such subset the sum of the elapsed times of the member programs is a lower bound. The greatest of these lower bounds clearly belongs to the longest pairwise unprofitable subset.

2.7 Summary of Lower Bounds

We have seen that there exist the following types of lower bounds for the execution time of a given set of programs:

a. $\sum_i r_{ij} t_i$ one for each j

b. $\sum_i s_{ik} t_i$ one for each k

c. execution time of each pairwise unprofitable subset.

In addition to these, the execution time t_i of each program represents a fourth type. All four types of bounds are independent: that is, any one of them may be the greatest lower bound of all.

These bounds may be readily computed and the value (T say) of the greatest determined. While other lower bounds exist, they are less readily computed and will not be discussed here.

2.8 The Principle of Uniform Loading

The greatest lower bound T for the execution time of a given set of programs may be used to determine for each facility the degree of loading which is necessary if this greatest lower bound is to be attained. Of course, this theoretical loading and the greatest lower bound itself may well be unattainable. For a time-shared facility j and a space-shared facility k the necessary degrees of loading are respectively:

$$\frac{\sum_i r_{ij} t_i}{T} \quad \text{and} \quad \frac{\sum_i s_{ik} t_i}{T}$$

We shall refer to these quantities as target loadings.

In generating the early part of a schedule, it is possible through poor placement of programs to underload a time-shared or space-shared facility to such a degree that the bound T cannot possibly be attained even if a perfect packing is assumed for the remaining programs. It is also possible to overload a time-shared facility to such an extent that the time for executing the given set of programs is again unnecessarily prolonged.

It is clear that, if the degree of loading of each facility remained at its target value defined above, the shortest possible schedule would be obtained. However, it is also clear that in practice such an ideally uniform loading will normally be unattainable. A less stringent goal must be accepted.

Accordingly, we set lower and upper limits upon the loading of each facility. The upper limit for a space-shared facility should always be set to unity, since any greater value is useless and any lesser value may lengthen the schedule unnecessarily. For the same reason, the upper limit for a time-shared facility should not be set to a value less than unity, but it certainly may exceed unity. Within these constraints it is desirable that for each facility the mean of the lower and upper limits be as close as possible to the corresponding target loading. The remaining freedom in specifying values for these limits may be used to generate alternative schedules on a trial basis. These schedules may then be evaluated as described in Section 2.10.

2.9 Load and Limit Vectors

For convenience, we treat the space and time fractions of a program as components of a single vector, known as the load vector for that program. The load vector for a given mix is then simply the sum of the load vectors of all the programs participating in that mix.

The lower and upper limits for each facility may be treated similarly as components of a lower limit vector $\underline{b}$ and an upper limit vector $\underline{B}$, respectively.

For every mix we now require that each component of its load vector be less than or equal to the corresponding component of the upper limit vector $\underline{B}$. If at any time during the generation of a mix every component of the mix load vector equals or exceeds the corresponding component of the lower limit vector $\underline{b}$, no more programs are added to that mix.

2.10 Evaluation of a Schedule

We may expect that the execution time of any given program will be extended when run concurrently with other programs due to delays in getting service from time-shared facilities. These delays become significant when the time fraction load on one or more facilities exceeds unity.

For the sake of simplicity, schedules may be generated ignoring these delays. However, in comparing two schedules, allowance should be made for them; otherwise, an erroneous result may be obtained. Such allowance cannot be anything but approximate due to inadequacies in the information available concerning the behavior of each program.

Consider the time interval between successive mix changes (a mix change occurs whenever a program begins or ends). Let this time interval be of magnitude Δt before allowance is made for extension. Let λ be the maximum time fraction load which the pertinent mix places on the time-shared facilities. Then an approximate value for the extended time interval for the mix is:

$$\Delta t \cdot \max\ (1, \lambda)$$

By summing the extended time intervals for all mixes, we obtain an approximate execution time for the schedule.

PART III - THE SCHEDULING ALGORITHM

The scheduling algorithm examines the programs to be scheduled one by one and places their component rectangles in the corresponding load diagrams according to a set of placement rules.

The sequence in which programs are selected for placement is determined by their position in a list of unscheduled programs known as the U list. In the absence of external constraints, the preferred ordering for this list is by elapsed time, the longest program being first on the list. The reason for this is that delay in placing long programs in the schedule can cause one or more of these programs to project unoverlapped at the end of the schedule, thus yielding an unnecessarily long schedule.

3.1 The Placement Rules

The placement rules are designed with a view to generating short, compact schedules and making the scheduling program simple, concise, and capable of rapid execution.

Rule 1: Fitting Criteria

A program P is said to fit in a given position in the schedule if all three of the following criteria are satisfied:

1. Prior to the addition of P's load vector, none of the affected mixes has a load vector which equals or exceeds the lower limit vector $\underline{b}$ (see Section 2.9).

2. After the addition of P's load vector, none of the affected mixes has a load vector which exceeds the upper limit vector $\underline{B}$.

3. Its elapsed time is sufficiently short that none of its component rectangles intersects any rectangles already placed in positions with later starting times.

Rule 2: Left Justification

Each program is placed in the schedule in the leftmost eligible position in which it will fit. One consequence of this rule is that no program may start at a time other than the termination time of some other program. Expressed in terms of mix changes, this means that there are only two types: the dropout, in which one or more programs leave the execution phase and no new program enters; and the pickup, in which one or more programs leave the execution phase and one or more enter. A third type in which one or more programs enter and none leaves is prohibited by this rule.

An important objective of the left justification rule is to make schedules shorter. It also tends to make the early portion of the schedule more densely packed than the later -- a result which is clearly desirable in situations in which schedules are subject to augmentation by newly arrived workload or to outright abandonment. Further advantages of this rule will become apparent when we deal with precedence and urgency.

Rule 3: Single-Mix Test

Generally, the placement of a single program entails testing to see whether it fits in first one position, then another, until a position is found in which a successful fitting is achieved. Two important measures to reduce the amount of testing are:

1. The maintenance of a list of eligible positions -- the step list.

2. The adoption of a rule that no position is eligible which would require the testing of a candidate program in more than one mix for that position.

As we shall see, the single-mix rule does not prevent any program P from participating in more than one mix -- for that matter, in any number of mixes. It simply means that after P has been placed in the schedule -- and before any further programs are placed -- P participates in more than one mix if and only if these mixes are consecutive in time and all mix changes involved are of the dropout type.

Later placements in the schedule may result in the occurrence of any number of mix changes of the pickup type during the time for which P is scheduled.

Rule 4: No Fragmentation

As programs are introduced into a space-shared facility to which the adjacency constraint applies, there is a tendency for the vacant space to be broken up into an increasing number of small fragments. Consider an example (Figure 4) in which a long program P occupies an inner position. Let S_1, S_2 be the space fractions available respectively above and below P. Then, for the entire duration of P, it is impossible to fit any program into the machine which requires on this facility a space fraction greater than the larger of S_1, S_2. On the other hand, with P in an extreme position (top or bottom) the available fraction of adjacent space is $S_1 + S_2$.

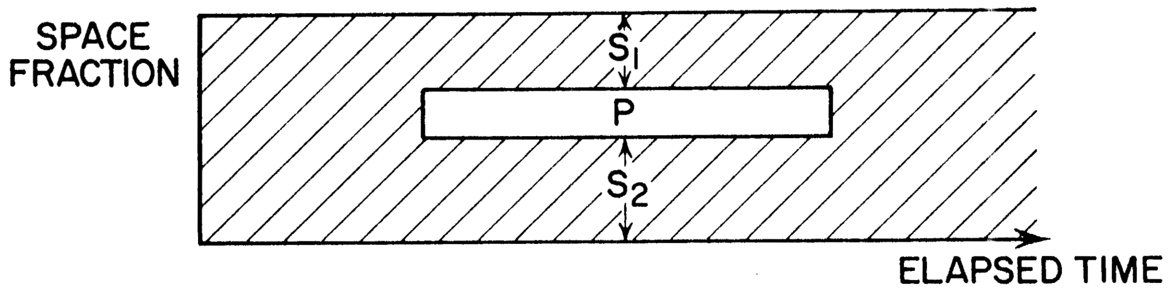

Fig. 4. Fragmentation.

The available space on a facility may be fragmented by a right or left hand projection of one program over another as in Figure 5 and 6.

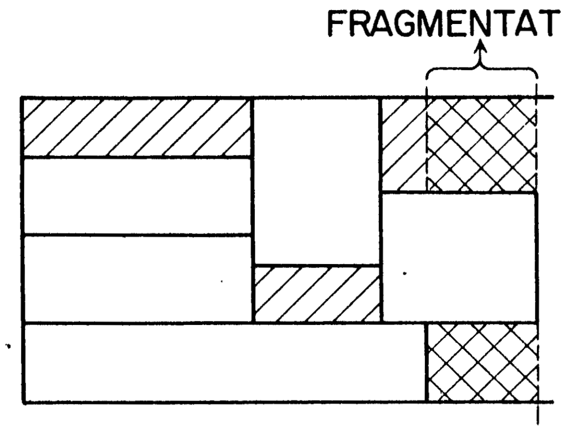

Fig. 5. R.H. Projection.

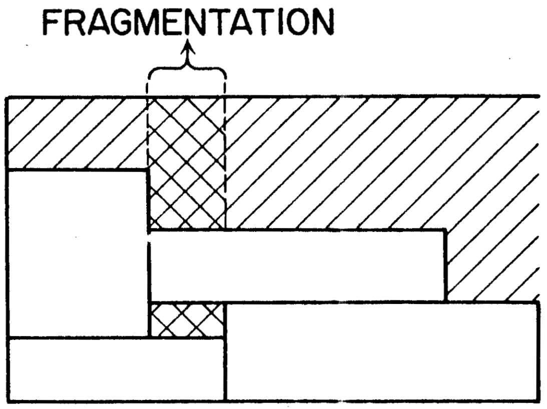

Fig. 6. L.H. Projection.

In order to avoid fragmenting available space on a given facility, it is necessary and sufficient that each program be so placed that throughout its whole elapsed time the space between its component rectangle and either the lower or upper boundary is solidly booked by the component rectangles of other programs.

Another way of expressing this is obtained by looking at the load diagram for a complete schedule. If a vertical section is made of the schedule at any point on the elapsed time axis, the programs for which space is reserved at this instant fall into two classes: the lower-level programs which hold contiguous space reservations starting at the lower boundary and proceeding upwards; and the upper-level programs which hold contiguous space reservations starting at the upper boundary and proceeding downwards. (Of course, one or both of these classes may be empty.) All available space at this instant is then seen to lie in a single uninterrupted area which separates the two classes of programs.

By avoiding fragmentation of available capacity on all facilities (space-shared and time-shared), it becomes possible to represent a schedule in a very concise way. For each mix change the time is recorded together with two ordered lists: one of the lower-level programs, ordered by proximity to the lower boundary; the other of the upper-level programs, ordered by proximity to the upper boundary.

Rule 5: No In-Process Relocation

Space is reserved for each scheduled program on the assumption that the program will not be relocated at any time during its execution phase. Although the schedule is generated on this basis, this constraint may be lifted by the supervisory program during execution time if unexpected circumstances arise which make relocation desirable (assuming the machine permits such relocation).

Adoption of this rule is virtually mandatory for machines which are incapable of directly executing a program in relative form. However, for any machine this rule considerably simplifies the scheduling algorithm.

3.2 Pyramids and Steps

The method of applying the five placement rules is based on two concepts: the pyramid concept and the step concept.

Suppose that in the load diagram for a given facility the component rectangles of several mixable programs A, B, C, etc., are stacked vertically in that order, one immediately on top of the other. Further, suppose that in each of the remaining load diagrams the corresponding set of rectangles is similarly stacked: that is, the order A, B, C, etc., is preserved in each stacking; contiguity is preserved; and start and stop times for each of the rectangles belonging to any given program are held constant. If such a vertical stack satisfies the following conditions, it is a pyramid:

a. the start times of A, B, C, etc., considered in that order are nondecreasing;

b. the stop times of A, B, C, etc., considered in that order are nonincreasing.

Note that these two conditions imply a third: the elapsed times of A, B, C, etc., considered in that order are nonincreasing. Several pyramids are illustrated in the sample schedule shown in Figure 12. For example, programs P_1, P_6, P_8, P_{12} form a pyramid.

Whenever a program P forms part of a pyramid D, it is convenient to regard all of the component rectangles of P taken collectively as a layer of D.

Under certain circumstances, pyramids are built in an upward direction (that is, toward the upper boundary in each load diagram). Such pyramids form the lower level of the schedule. In other circumstances, pyramids are built in a downward direction and these form the upper level of the schedule.

The base of a pyramid rests upon:

a. the lower horizontal boundary of every load diagram (the elapsed time axis in every case);

or b. the upper horizontal boundary of every load diagram;

or c. an uncovered segment of a layer of an already established pyramid.

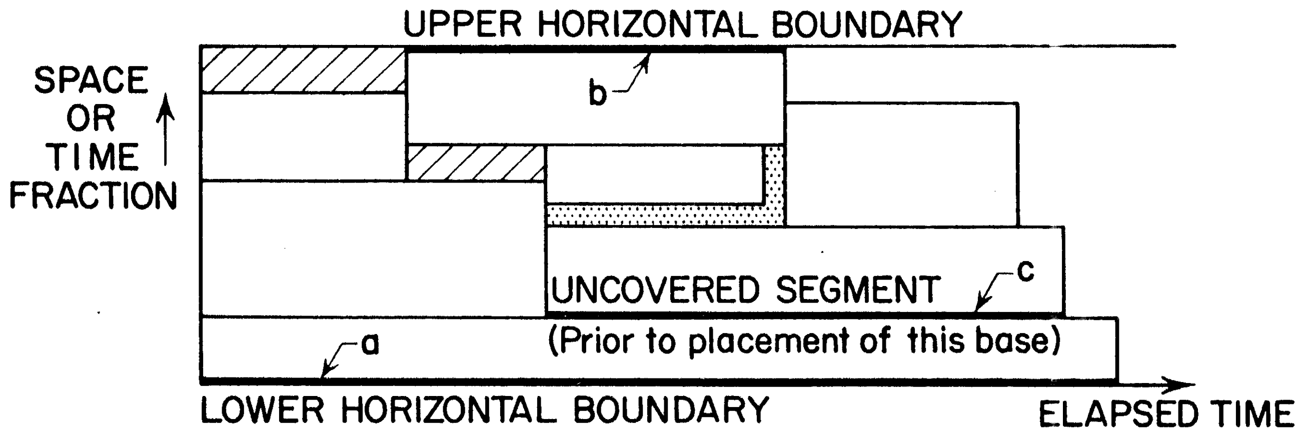
UPPER HORIZONTAL BOUNDARY
SPACE OR TIME FRACTION
b
UNCOVERED SEGMENT
c
a
(Prior to placement of this base)
LOWER HORIZONTAL BOUNDARY
ELAPSED TIME

Fig. 7. Positions for Pyramid Bases.

These positions are examples of steps (see Figure 7). A precise definition of step is rather complicated. For the time being, we may regard it as a position in the schedule at which a program may be placed without violating any of the placement rules.

3.3 The Step List

The step list serves the function of reducing the amount of searching needed to find a place in the schedule for a given program. It not only reduces the average number of positions which are examined, but it also provides for a much simpler test of horizontal fitness than the schedule table would allow.

Each entry consists of the start time of the step, its length (i.e., elapsed time), and an indication of its level (lower or upper). Steps are ordered by their start times, the earliest being first. When two steps have identical start times, the shorter is placed ahead of the longer; when the lengths are equal, the ordering is arbitrary.

Before the first program is placed in the schedule, the step list contains two entries: one for the lower horizontal boundary, the other for the upper. As each program is placed in the schedule, one or more steps may be introduced into or removed from the list, according to circumstances.

3.4 The Schedule Table

The schedule is built up in the form of a table, which not only serves finally as the output, but also as the means of determining vertical fitness of a candidate program for a selected position, whenever the step list has indicated its horizontal fitness for that position.

Before the first program is placed in the schedule, the schedule table is empty. As each program is placed, its terminating time will normally define a new change of mix. Only when the terminating time of a program just placed happens to coincide with that of a previously placed program will there be no new mix change introduced.

For every new mix change, an entry is created in the schedule table. Each entry consists of the time for the mix change together with the following information for the mix due to begin at this time:

a. A list of lower level programs ordered by proximity to the lower horizontal boundary;

b. A list of upper level programs ordered by proximity to the upper horizontal boundary;

c. The load vector for this mix.

The entries are ordered by mix change time, the earliest being first.

Placing a program in the schedule normally entails more than the creation of a new entry in the schedule table corresponding to the new mix change. For each of the mixes in which this program participates:

a. The load vector must be updated;

b. Either the lower or upper level list must be augmented by the program just introduced according to its placement (lower or upper) in the schedule.

3.5 Placing a Program

Suppose that a partial schedule has been developed, the schedule table and step list are up-to-date, and the system is ready to find a position in the schedule for the next program from the U list.

The step list is searched for the first step upon which the candidate program fits horizontally (that is, with respect to elapsed time). When such a step has been located (and there are always at least two such steps -- the infinite ones -- in the list), the schedule table is inspected beginning at the entry which has a mix change time equal to the start time of the step. The load vector for the corresponding mix is tentatively incremented by the load vector of the candidate program, and the result is compared with the upper-bound vector. If no component of the incremented load vector exceeds the corresponding component of the upper-bound vector, the program is known to fit on this step, because a step is constrained not to straddle mix changes other than the dropout type. This is, indeed, the technique by which the single-mix rule is observed.

If, however, the upper-bound vector is exceeded (in the component by component sense), then a new position is sought on the same step. To do this, the next entry in the schedule table is inspected, providing the remaining segment of the step under consideration is still sufficiently long to accommodate the program. These inspections continue until either an acceptable position is found or a segment remains which is too short (see Figure 8). In this latter case, the step-list search is resumed and the next entry of sufficient length is treated in a similar way. The process is repeated until the earliest step is found upon which the candidate program fits horizontally and vertically.

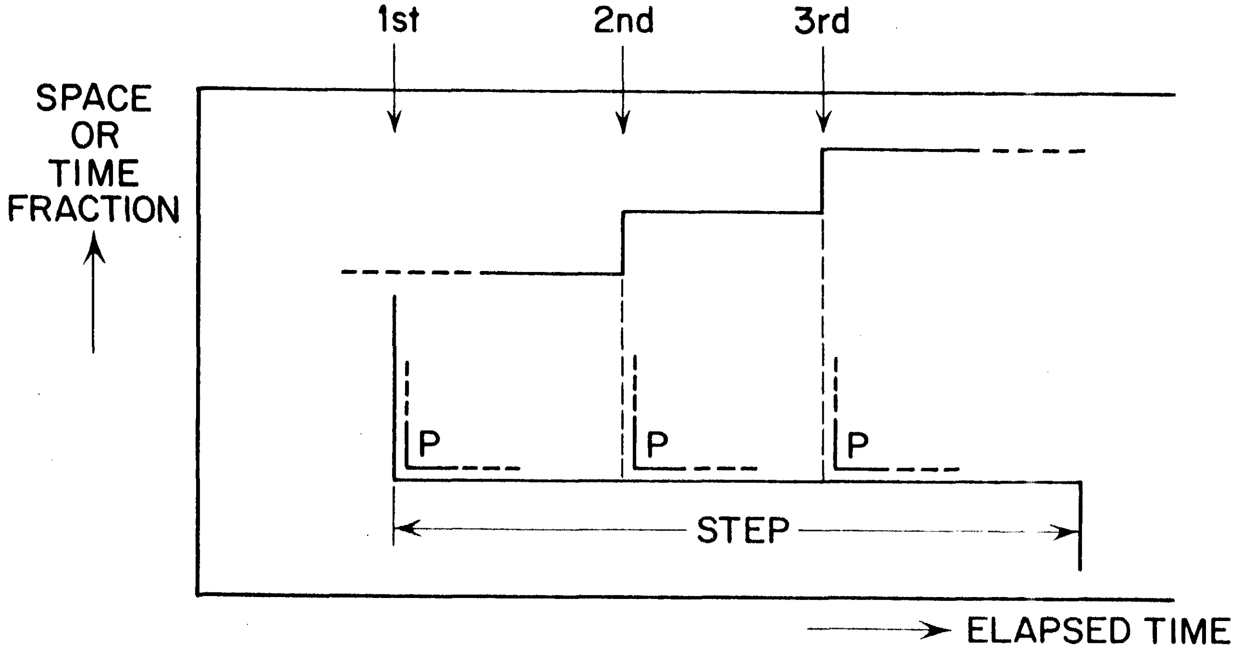

Fig. 8. Successive Trial Positions on a Step.

Having found a place in the schedule for the candidate program, the schedule table is updated as described in Section 3.4. The step list must also be updated; a description of this procedure will help to clarify the notion of step.

3.6 Updating the Step List

The alterations necessary to bring the step list up to date after a program P has been placed in the schedule depend upon two considerations:

a. whether or not P was placed so that its start time coincided with that of the step upon which it rests;

b. whether or not the lower limit $\underline{b}$ was reached in any of the mixes in which P participates.

In what follows we shall assume that P has been placed on a lower-level step. The arguments apply with equal force to placement on an upper-level step by simply interchanging lower and upper wherever they appear.

Let the elapsed time of P be d_P, the start time of the step upon which P is placed by x_o, its length d_o. The subscripts -1, 1 denote the immediately preceding and succeeding steps. A prime denotes the opposite level. Thus x'_{-1}, x'_1 denote the start times of the immediately preceding and succeeding steps on the opposite level. We now consider the case in which the start time of P coincides with that of the step upon which it rests (Figure 9).

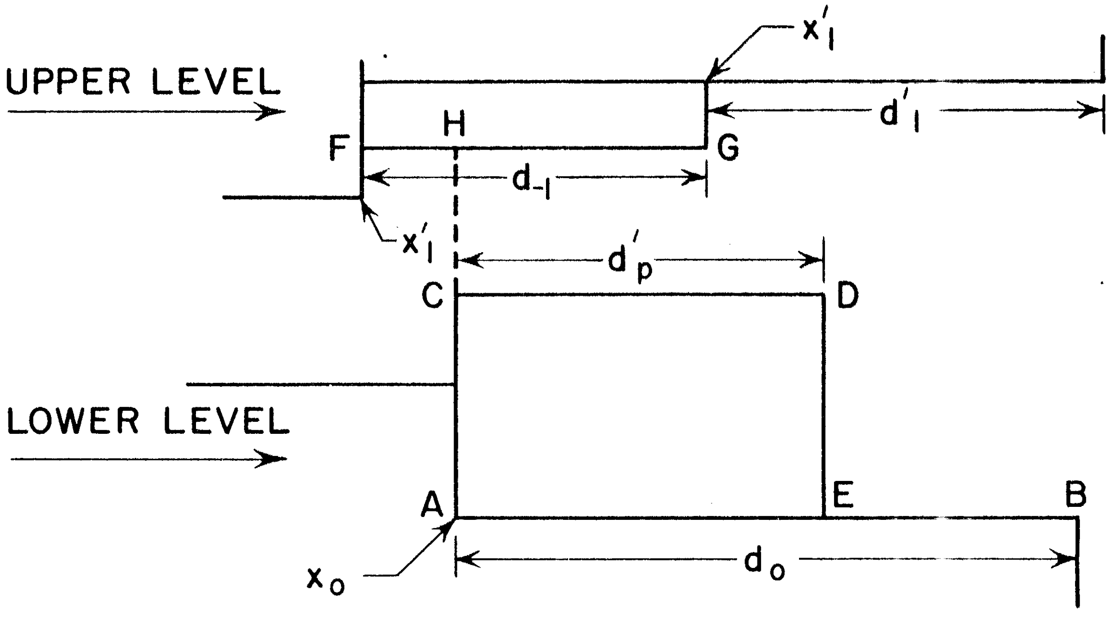

Fig. 9. Start Time of P Equals that of Step.

Program P (represented by the rectangle ACDE) rests on step AB. The entry (x_o , d_o) in the step list for step AB must now be removed and replaced by two entries:

(x_o , d_p) for step CD;

$(x_o + d_p , d_o - d_p)$ for step EB

Note that if AB is the lower boundary of the load diagram, $d_o = \infty$, $d_o - d_p = \infty$; and the two new steps are still specified correctly by the statements above.

On the upper level, there may exist a step FG which straddles the new mix change of pickup type at time x_o. If so, this step must also be split into two steps to comply with the single-mix rule. The existence of such a step is decidable by inspecting the upper-level step immediately preceding (x_o , d_o) in the step list. If its start time x'_{-1} plus length d' strictly exceeds x_o, the existence of a straddling step FG is confirmed. This entry is accordingly replaced by the two entries:

$x'_{-1} , x_o - x'_{-1}$ for step FH;

$x'_o , d'_{-1} - x_o + x'_{-1}$ for step HG

The alternate case may now be considered: the start time of P is later than the start time of the step upon which it rests (Figure 10). In this case, due to the adoption of the left justification rule, the start time of P

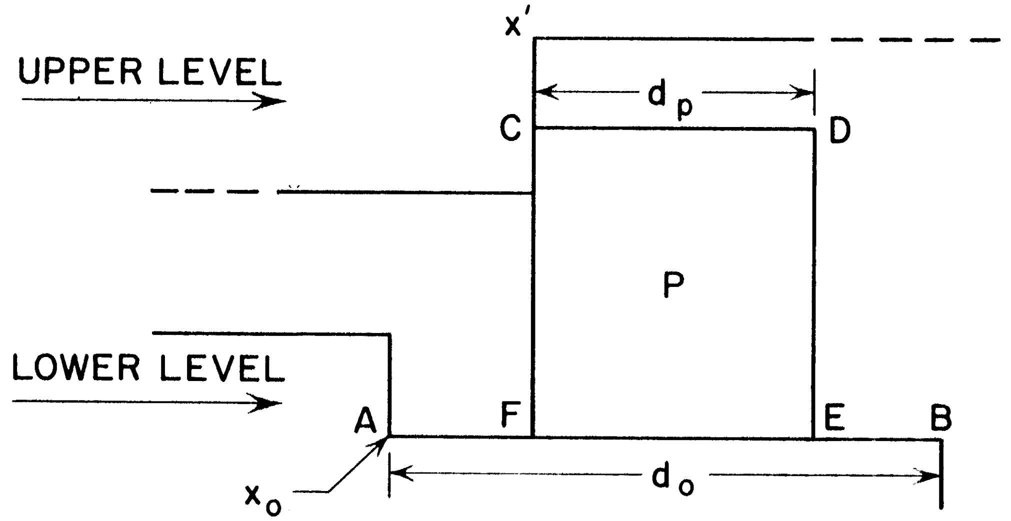

Fig. 10. Start Time of P Later than that of Step.

must necessarily be the start time (x' say) of an upper-level step. Accordingly, we replace the entry (x_o , d_o) in the step list by the following two entries:

$$(x_o , x' - x_o) \quad \text{for step AF}$$

$$(x' , d_p) \quad \text{for step CD}$$

together with a third entry:

$$(x' + d_p , x_o + d_o - x' - d_p) \quad \text{for step EB}$$

providing $x_o + d_o - x' - d_p$ does not vanish.

To complete the updating of the step list, the earliest mix in which P participates is examined to ascertain whether all components of its load vector have equaled or exceeded the corresponding components of the lower-limit vector $\underline{b}$. If so, the entry in the step list for the step CD is adjusted as follows: its start time is incremented and its length decremented by the duration of the mix just examined. Similar action is taken on steps on the opposite level. The next mix is then examined in a similar way until either a mix is found in which the lower limit is not attained (in the all-component sense) or all the mixes in which P participates have been exhausted. Note that once a mix has been found in which the lower limit is not attained, succeeding mixes need not be examined since all mix changes for a given step are of the dropout type.

3.7 Current Step

It may be observed that the algorithm does not generate pyramids one by one; it tends instead to build a layer on one pyramid, then a layer on another, and so on. This is partly due to the fact that, for reasons of economy, only one sweep is made through the U list.

Actually, a further economy in effort may be made with some loss in compactness of resulting schedules by weakening the left justification rule in the following way: let the program most recently placed in the schedule be P; that segment of its uncovered horizontal side which survived the lower limit test is defined as the current step; the next program to be placed in the schedule is tested for horizontal and vertical fitness on the current step first, and only if this step proves unacceptable is a search made through the step list. This modification biases the algorithm to continue building the pyramid upon which it last placed a layer. This bias is of doubtful value unless the U list is strongly biased toward the preferred ordering (by elapsed time).

3.8 An Example

A set of scheduling data is provided in Figure 11, below, for an example in which 12 programs are to be scheduled on a single, space-shared facility. The schedule which results from treating them in the preferred order is shown in Figure 12 and the final state of the schedule table and step list tabulated in Figures 13 and 14. The reader who wishes to acquaint himself with the details of the algorithm is advised to generate the schedule step by step and check the result.

Program	Space Fraction (Units = 1/32)	Elapsed Time
P_1	8	100
P_2	4	70
P_3	8	50
P_4	6	40
P_5	10	35
P_6	8	27
P_7	2	25
P_8	4	22
P_9	5	17
P_{10}	4	14
P_{11}	14	12
P_{12}	7	10

Lower Limt = 28 (Units = 1/32)

Figure 11 . Table of Scheduling Data.

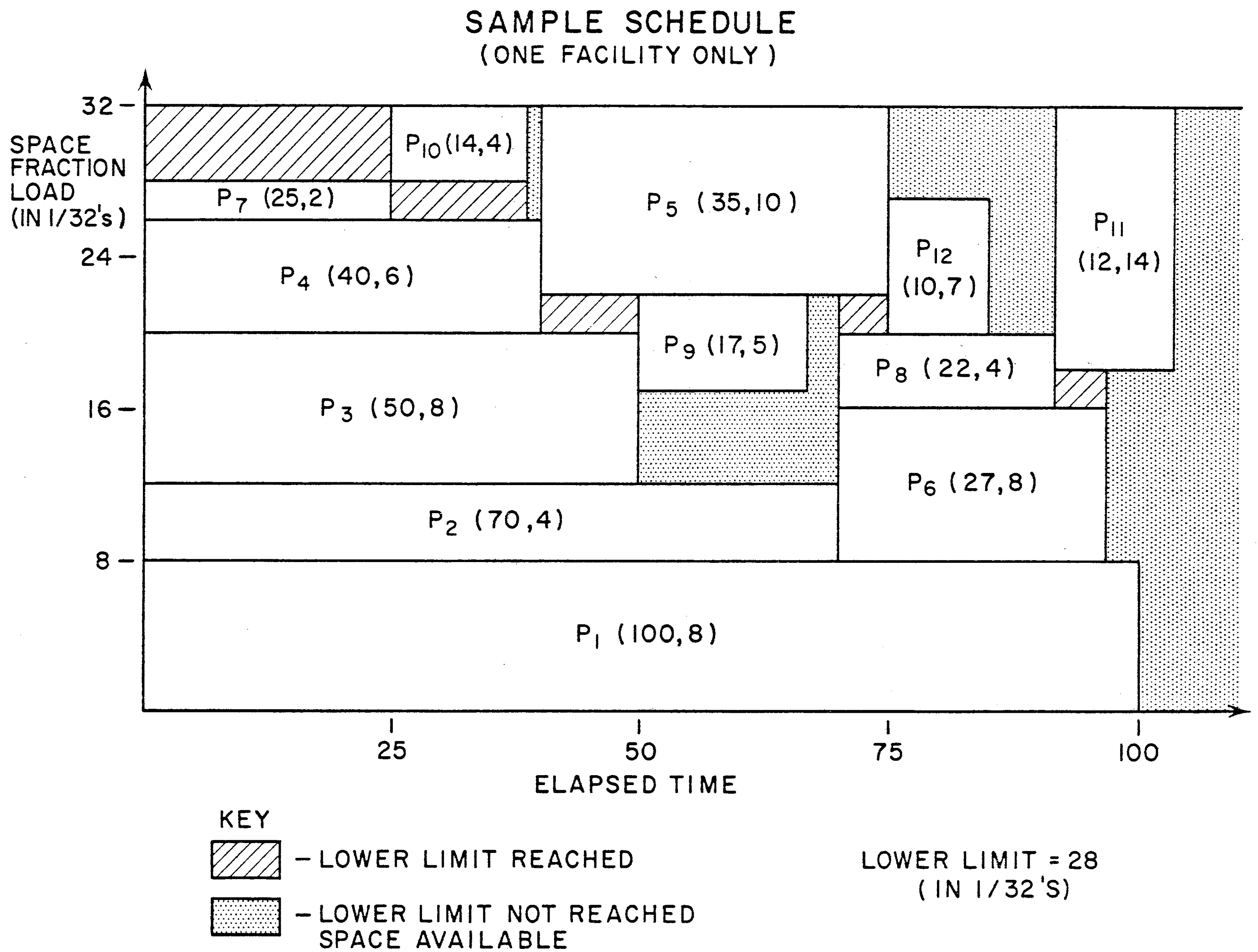
SAMPLE SCHEDULE
(ONE FACILITY ONLY)
32
24
16
8
SPACE FRACTION LOAD (IN 1/32's)
P10 (14,4)
P7 (25,2)
P5 (35,10)
P4 (40,6)
P12 (10,7)
P11 (12,14)
P9 (17,5)
P8 (22,4)
P3 (50,8)
P6 (27,8)
P2 (70,4)
P1 (100,8)
25
50
75
100
ELAPSED TIME
KEY
– LOWER LIMIT REACHED
– LOWER LIMIT NOT REACHED SPACE AVAILABLE
LOWER LIMIT = 28 (IN 1/32'S)

Fig. 12.

Time	Lower Level List	Upper Level List	Load Vector*
0	P_1, P_2, P_3, P_4, P_7	----	28
25	P_1, P_2, P_3, P_4	P_{10}	30
39	P_1, P_2, P_3, P_4	----	26
40	P_1, P_2, P_3	P_5	30
50	P_1, P_2	P_5, P_9	27
67	P_1, P_2	P_5	22
70	P_1, P_6, P_8	P_5	30
75	P_1, P_6, P_8, P_{12}	----	27
85	P_1, P_6, P_8	----	20
92	P_1, P_6	P_{11}	30
97	P_1	P_{11}	22
100	------	P_{11}	14
104	------	----	0

Figure 13: Final State of Schedule Table

Time	Length	Lower/Upper
39	1	L
39	1	U
50	17	U
50	20	L
67	3	U
75	10	L
75	17	U
85	7	L
97	3	L
97	7	U
100	∞	L
104	∞	U

Figure 14: Final State of Step List

* A scalar in this example due to treatment of one facility only. Units for the Scalar are 1/32.

PART IV - EXTERNAL CONSTRAINTS

4. 1 Urgency

The scheduling algorithm provides two ways for handling urgency; each complements the other. The first method consists of excluding from the U list all programs of urgency less than a specified degree. Programs excluded in this way are guaranteed inclusion in some subsequent scheduling operation by an automatic advance in their urgency ratings. The second method consists of ordering those programs included in the U list first by urgency rating and then by elapsed time. Both the membership and the ordering of the U list are assumed to be determined by an installation-oriented routine.

4. 2 Group Service

Where several groups of users are sharing a large central machine, it is likely that a policy will be established under which each group is guaranteed some measure of service, even though on occasions this may prolong the time taken to process the whole workload. As with urgency, such a policy may be implemented by controlling membership and ordering of the U list.

4. 3 Precedence

If program P is to precede program Q in the schedule, it is necessary to place P ahead of Q in the U list. However, this action alone is not sufficient, because the ordering of the U list has only an indirect effect upon the ordering of programs in the schedule. A direct constraint is required upon the placement procedure. This constraint employs the notions of next predecessor and next successor.

P is said to be a next predecessor of Q (and Q a next successor of P) if P precedes Q and there exists no Z such that P precedes Z and Z precedes Q. A program may possess more than one next predecessor. In the example of a precedence chain in Fig. 15, program F has 3 next predecessors C, D, E.

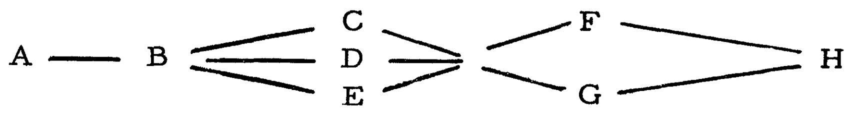

Fig. 15 Precedence Chain

Not until all three of these programs have been placed in the schedule can the earliest starting time of F be determined. It must then be given the value of the latest terminating time of C, D, E.

Accordingly, the U-list entry of each program is expanded to include a list of its next successors together with an earliest starting time. All earliest starting times are initially set to zero. Whenever a program is placed in the schedule, the list of its next successors is consulted. For each next successor the earliest starting time is replaced by the terminating time of the program just scheduled if this action results in a later earliest starting time. Placing a program now involves a search starting with the earliest position which has a starting time equal to or later than the earliest starting time of the given program.

4.4 Immediate Precedence

Often it is desired that execution of the next successor of a given program should start as soon as that program has been completed. This is very likely to be the case whenever a single program consisting of several phases which have significantly different space and time fractions is presented as though each phase were a program in its own right. In such a case, the phases would be linked together by precedence statements of the immediate type.

A flag is added to each entry in the U list to indicate whether or not the successors (if any) of the given program are immediate. The scheduling of such programs is considerably simplified if they are considered as a single, long program while the search for a position is in progress. However, as soon as a satisfactory position has been found, the schedule table is updated by the individual program load vectors.

4.5 The Dynamic Mode

In the dynamic mode, the unexecuted part of the schedule is augmented from time to time by incoming load, and the executed part is dismissed. Essentially, the same algorithm may be employed, except that, since some part of the schedule is always being implemented, there is little scope for evaluation and regeneration of alternative schedules.

4.6 Degraded Versions

The criteria of fitness of a program with respect to a position in the schedule are readily relaxed in case little or no information is supplied concerning the time requirements. As an example, suppose that the precision of time fractions is reduced to one bit for each time-shared facility indicating whether 100% of its time or less is required. Vertical fitness would be reduced to ascertaining whether the relevant mix was profitable or unprofitable in the sense defined in Section 2.5.

When elapsed times are not available, scheduling necessarily becomes very primitive.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the contributions of two colleagues: E. S. Lowry, who proposed several of the concepts in Part 2; and R. H. Ramey, who suggested simplifications in Part 3.

REFERENCE

1. E. F. Codd, E. S. Lowry, E. McDonough, C. A. Scalzi, "Multiprogramming STRETCH: Feasibility Considerations, " Communications of the ACM (November 1959).