STUDY OF THE PERFORMANCE OF A SCHEDULING ALGORITHM FOR A TIME-SLICING SUPERVISOR

A Thesis

Presented To

the Faculty of Graduate Studies and Research

The University of Manitoba

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in the Institute for Computer Studies

---

by

Lloyd A. Chai

May 1968

TITLE: STUDY OF THE PERFORMANCE OF A SCHEDULING ALGORITHM FOR A TIME-SLICING SUPERVISOR

AUTHOR: Lloyd A. Chai

ABSTRACT

A job-scheduling algorithm based upon round-robin scheduling with a variable time-slice is studied under simulation. The simulation model is described, and a measure of performance of scheduling algorithms is developed. The measure of performance is based upon the cost curves suggested by Greenberger and, in contrast to Greenberger's analytical study, non-linear cost curves are used.

The job-scheduling algorithm which uses variable time-slice allocation is shown to be more efficient than an analogous algorithm which uses constant time-slice allocation. The improvement in the measure of performance for the variable time-slice algorithm relative to the constant time-slice algorithm is approximately 15% for the job load considered.

The optimum round-robin cycle time was determined to be 1.5 secs. for a particular job-load, and this value is in fair agreement with values found by Greenberger in an analytical study using linear cost curves with a constant time-slice algorithm.

ABSTRACT

A job-scheduling algorithm based upon round-robin scheduling with a variable time-slice is studied under simulation. The simulation model is described, and a measure of performance of scheduling algorithms is developed. The measure of performance is based upon the cost curves suggested by Greenberger [8] and, in contrast to Greenberger's analytical study, non-linear cost curves are used.

The job-scheduling algorithm which uses variable time-slice allocation is shown to be more efficient than an analogous algorithm which uses constant time-slice allocation. The improvement in the measure of performance for the variable time-slice algorithm relative to the constant time-slice algorithm is approximately 15% for the job load considered.

The optimum round-robin cycle time was determined to be 1.5 secs. for a particular job-load, and this value is in fair agreement with values found by Greenberger in an analytical study using linear cost curves with a constant time-slice algorithm.

ACKNOWLEDGMENTS

I would like to express my deep appreciation to Professor T.A. Rourke, my thesis supervisor, for the many hours he has spent providing guidance and direction in this research.

I wish to thank Professors B.A. Hodson and M.A.K. Hamid for the reading of this thesis.

Also, I would like to express my appreciation to Dr. J. Blatny for discussing some of the ideas contained herein; and to Dr. S.R. Clark for his constructive criticisms.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

The following notations have been adopted for use in this thesis:

CPU	:	Central processing unit
f_j	:	Penalty function associated with priority j
FTS	:	Fixed time-slice
i-j mix	:	Program mix consisting of a priority i program and a priority j program
i-j-k mix	:	Program mix consisting of a priority i, a priority j and a priority k program
m_i	:	Mean of normal distribution for requested CPU time of priority class i
$p_i(t)$	:	Current accumulative penalty of program i at time t
p_i^*	:	Final penalty of the finished program i
P(t)	:	Total operating penalty of the computer system at time t
P_N^*	:	Sum of the final penalties of N finished programs
q_i	:	Quantum or time-slice for program i
r_i	:	Mean of normal distribution for interarrival time of priority class i
s_{m_i}	:	Standard deviation of normal distribution for requested CPU time of priority class i
s_{r_i}	:	Standard deviation of normal distribution for interarrival time of priority class i

t_{rr} : Intended round-robin time and is referred to simply as the round-robin time

t^{o}_{rr} : Optimum round-robin time

T_i : The real time which has elapsed since the desired time of entry of program i into the computer system

T'_{rr} : Time interval between the start of the round-robin cycle in which the program completes processing and the time when the program leaves the system after it has completed processing

u_i : Lack of attention to program i; and is defined by the formula

$$u_i = T_i/\tau_i$$

VTS : Variable time-slice

Δp_i : Final penalty of program i under FTS algorithm minus final penalty of program i under VTS algorithm

Δt_i : Exit time of program i under FTS algorithm minus exit time of program i under VTS algorithm

μ : Population mean

π_i : Priority number of program i

σ : Standard deviation of the population

τ_i : The sum of the central processor time and device time devoted to program i since it entered the computer system

[n] : Refers to the n^{th} entry in the Reference Section

CHAPTER I

INTRODUCTION

A major advance in the design of digital computers was achieved when provision was made for autonomous input/output transfers, which allowed computation to proceed concurrently with transfer of data into or out of the main memory[1] (core) of a computer.

In many computations, however, the processing associated with an input/output transfer normally takes much less time than the transfer itself and therefore the total saving obtained in spite of the computer's capability of autonomous transfer of data is usually very small. To make further savings one could reduce the cost of having the central processing unit[2] (CPU) idle while a data transfer is taking place by using a slower and cheaper CPU. A problem

---

[1]Main memory is the fastest storage device of a computer and the one from which instructions are executed. The storage device is usually composed of ferromagnetic cores and hence is also called core memory or core.

[2]Central processing unit (CPU) is the unit of a computing system that contains the circuits that control and perform the execution of instructions. It contains an arithmetic unit and special register groups.

that arises from using the latter approach however, is that such a CPU could not handle those types of operation which require high-speed processing. Therefore, if the job mixture is at all varied regarding CPU demands, a fast CPU may be necessary.

A second problem concerning the efficient use of the capability of autonomous transfer arises because many input/output devices are capable of working independently of themselves and of the CPU. In addition, it is very unlikely that one program could use all the independent facilities all of the time. Thus, many of the facilities are often idle for significant periods of time. However, computer efficiency may be improved if these idle periods are reduced. The problem automatically suggests allowing more than one program to share the main memory in the hope that the combined demand of the executing programs can maintain the fullest possible useful activity in those parts of the computer which can function simultaneously. This facility is known as time-sharing.

It appears that time-sharing may largely solve the problem of how to use modern computer facilities most efficiently, but the logic to organize all the data transfers and the allocation of the central processor's time to the various programs being executed

must be present. This logic is usually in the form of a supervisory program, part of which is always resident in core.

The efficiency of a computer system is dependent on the capability and speed of the data transfers, on the speed of the CPU and on the design of the supervisory programs. If the supervisory program schedules the data transfers and allocates the central processor time inefficiently, the computer system will operate inefficiently. Thus, the selection of an efficient algorithm for the scheduling function of the supervisory program is necessary to achieve desired system performance standards. An algorithm which effects the scheduling function of a supervisory program is called a job-scheduling algorithm.

Competition for the Computer Facilities

Since it is the purpose of the supervisory program to keep the demand for the computer facilities at a high level, it is inevitable that two or more programs will require a particular data transfer facility or processing facility at the same time. The supervisory program should, therefore, contain sufficient logic to determine how the facilities should be shared among a number of competing programs.

A basic technique for scheduling the central processing unit is to give each program sharing the computer's main memory a short burst or quantum of computation. The quantum of computation is also called a time-slice. The sequence in which the programs in core receive time-slices may be a simple round-robin cycle in the most straightforward case. A supervisory program which schedules this way is called a time-slicing supervisor.

Study of the Performance of Supervisory Programs

The need to establish more and better guides or rules-of-thumb for evaluating the efficiency of modern computer systems, and hence the performance of supervisory programs, has been expressed by Neilsen [1][3] and Fife [2]. The acquiring of such information, however, usually requires a great amount of time and effort in the investigation of such complex systems.

Most of the performance studies which have been conducted on supervisory programs, have used either analytic or simulation techniques. Because the complexity of supervisory systems does not easily lend itself to

---

[3]The nomenclature [n] refers to the n^{th} entry in the Reference Section.

theoretical analysis, any analytical study usually involves making some simplifying assumptions about the system under consideration. Some of the analytical studies which have been made include studies by Scherr [3] who was able to design a simple model of the Project MAC (Multiple Access Computer) at Massachussetts Institute of Technology, by Kleinrock [4] who gave a theoretical treatment of several models of time-shared processor system and by Shemer [5] who did analytical studies of several scheduling algorithms. Such analytical models, however, usually have rather limited application and lack the flexibility necessary to test a wide variety of system changes, a facility which is desirable in this type of study.

Simulation, on the other hand, was found to be more adequate in dealing with the problems arising from the complexities of supervisory systems. One advantage of simulation models is their relative flexibility in adaptation to different computer systems. In studies conducted by Scherr [3] of the Project MAC system, by Fine and McIsaacs [6] and by Neilsen [1], their simulation models were not only used to discover areas in need of modification but also to determine the effectiveness of proposed improvements.

Of the two techniques, analytic and simulation, simulation seems to be more promising in studying

supervisory system performance. The simulation approach not only is potentially more flexible than analytic approach but has been found satisfactory in the analysis of some time-sharing systems, such as Scherr's study of the Project MAC system [3] and Fine and McIsaac's study [6].

The present research is an investigation, using simulation techniques, into the scheduling function of a supervisor, namely, a job-scheduling algorithm.

Time-slicing algorithms which are in use today can be broadly divided into two types: round-robin procedures and multiple-priority-level procedures. A round-robin scheme, which has been mentioned earlier, services programs in a queue, giving each program a slice of CPU time before passing control of the CPU to the next program in sequence. A multiple-priority-level scheme assigns each program, as it enters the system, to one of several priority-level queues according to its size and determines the amount of time-slice to be allocated to a program according to the level which the program occupies. In this scheme, servicing begins with the program at the head of the highest level queue which is occupied, and if a program is not completed after executing its time-slice, it is placed at the end of the next level queue immediately below the current level queue. A queue is only serviced when all higher level queues cannot make use of the service.

This latter scheme has been employed by Corbato's Compatible Time-Sharing System [7]. Compared to the round-robin procedure, the multiple-priority-level procedure has greater flexibility inherent in the choice of initial priority assignment of programs and the maximum CPU time allocated to programs of each priority level.

The algorithm which is simulated, services programs (up to a maximum of three) in core using the round-robin scheme with each program's quantum computed according to the logic of the algorithm. Programs which do not gain entry into core upon their arrival are queued and their selection from the queue is determined by the logic of the scheduling algorithm.

The emphasis of this study is on the behaviour of the job-scheduling algorithm with given work-load environment. The overall study includes the specification of a model and the specification of a performance measure which is used to evaluate the efficiency of the scheduling algorithm modeled.

The results from this type of study are necessary in order to gain a clearer insight into both the basic operation and the characteristics of the algorithm modeled and to enable some assessment both qualitative and quantitative to be made of the performance of the algorithm.

Further, the results obtained can be used to determine whether the algorithm should be commended for future studies.

1.1 Summary of Frequently-Used Terminology

* Central Processing Unit (CPU). This is the element of a computer which handles the actual computation and decision-making functions. It consists of two sections: the control section which directs and coordinates all operations called for by instructions and the arithmetic-logical section which contains the circuitry to perform arithmetic and logical operations.

* Job-Scheduling. This term refers to the task of determining the allocation of computer time among the different users of a time-shared computer system. It also involves the forming of queues, handling of priorities and swapping of programs.

* Main Memory. This term, usually called core, refers to the fastest storage device of a computer and the one from which instructions are executed. Generally, main memories utilize magnetic core construction.

* Supervisory Program. The supervisory program or supervisor is a special program responsible for controlling and coordinating all the activities of a computer system. It controls all input and output functions of the

system and is responsible for the scheduling of jobs for execution. It also establishes user priorities, allocates storage, keeps track of all work in progress, and activates the necessary routines to handle whatever problems that arise. In general, it acts as the controlling element which assures continuous, accurate operation of the computer system.

* Time-slicing. This term refers to the technique of processing a number of user programs by giving a predetermined amount of CPU time to each program in turn, providing the program can use it.

CHAPTER II

DESCRIPTION OF THE SIMULATION MODEL

The simulation model is designed to simulate a job-scheduling algorithm which can simultaneously schedule up to three jobs.[4] The model used is the first stage of a comprehensive computer-system simulation model which will include the operations of logging-in, loading into core, job-scheduling and outputting of the results. This first stage may be considered as a model of a hypothetical computer system which incurs no overheads to load programs into core, and which processes to completion programs which require no input or output during their execution.

The present study, therefore, concentrates on the logic and efficiency of the algorithm by which the programs to be processed receive their slices of central-processing-unit time.

The model includes a job generator which creates a series of entities, called jobs or programs, from previously chosen program-load statistics using Monte Carlo

---

[4] In this study, the maximum number of jobs which can be scheduled simultaneously is restricted to 3. However, for possible future studies, a facility is provided for expanding the model so that as many as 20 jobs can be scheduled simultaneously.

techniques. It maintains a list which records the description and status of all programs in core (execution list). Programs in core receive CPU attention until their total amount of CPU time required (a program-load statistic) has been accumulated. Since a maximum of three programs can be scheduled simultaneously, this execution list can have as many as three entries. In addition, the model maintains a queue of waiting programs (waiting list) which is fed by new programs that are not accepted into core upon their arrival and which is emptied by acceptance of programs into core for CPU attention according to the logic of the scheduling algorithm. The queue is limited to 20 programs. When this limit is exceeded, the simulation run is terminated automatically.

Appendix A gives the contents of an entry in the waiting list and also in the execution list.

The job-scheduling algorithm whose function is to coordinate the processing of the jobs may be divided into three sections:

(I) A piece of logic to select a program or job from the waiting list;

(II) A piece of logic to determine the amount of CPU time which a program in core is to receive as its time-slice; and

(III) A piece of logic to determine the action to be taken when a program has completed execution of its time-slice or when a program is interrupted during its time-slice by the arrival of a new program in core.

The basic design of the simulation model, and hence of the supervisory program which is being simulated, involves a perpetual cycling. On each such supervisor cycle the model performs one of three rather elementary operations which contribute towards the simulated execution of the programs that pass through the hypothetical computer system. These operations will be covered in detail later, but very briefly they are :

(i) the execution of a quantum-allocation routine which computes the time-slices of all the programs in core for the next round-robin cycle (this is done in accordance with the logic of the job-scheduling algorithm for determining a program's time-slice - section II above);

(ii) the execution of a routine to locate from the execution list a program in core with an outstanding time-slice (search routine), followed by the execution of a routine which

simulates the initiation of the time-slice (service routine) if the search routine finds a program for execution; and

(iii) the execution of an update routine which records the current accumulated CPU time for a program in core immediately after that program has received the whole or part of its time-slice, followed by the execution of a removal routine which removes the program from core if the required CPU time has been satisfied.

Figure 1 is a general flow-diagram of the model showing the positions of the five routines mentioned above. It also illustrates that one of three paths may be taken during each supervisor cycle with the event-control routine common to each path.

The job generator is contained in the event-control routine along with the logic to determine which program is accepted next from the waiting list (section I above).

It is convenient to call the quantum-allocation and the search routines primary routines so that they can be distinguished from the other routines later in the description.

As mentioned above, whenever the supervisor cycle is executed, the event-control routine is performed. The

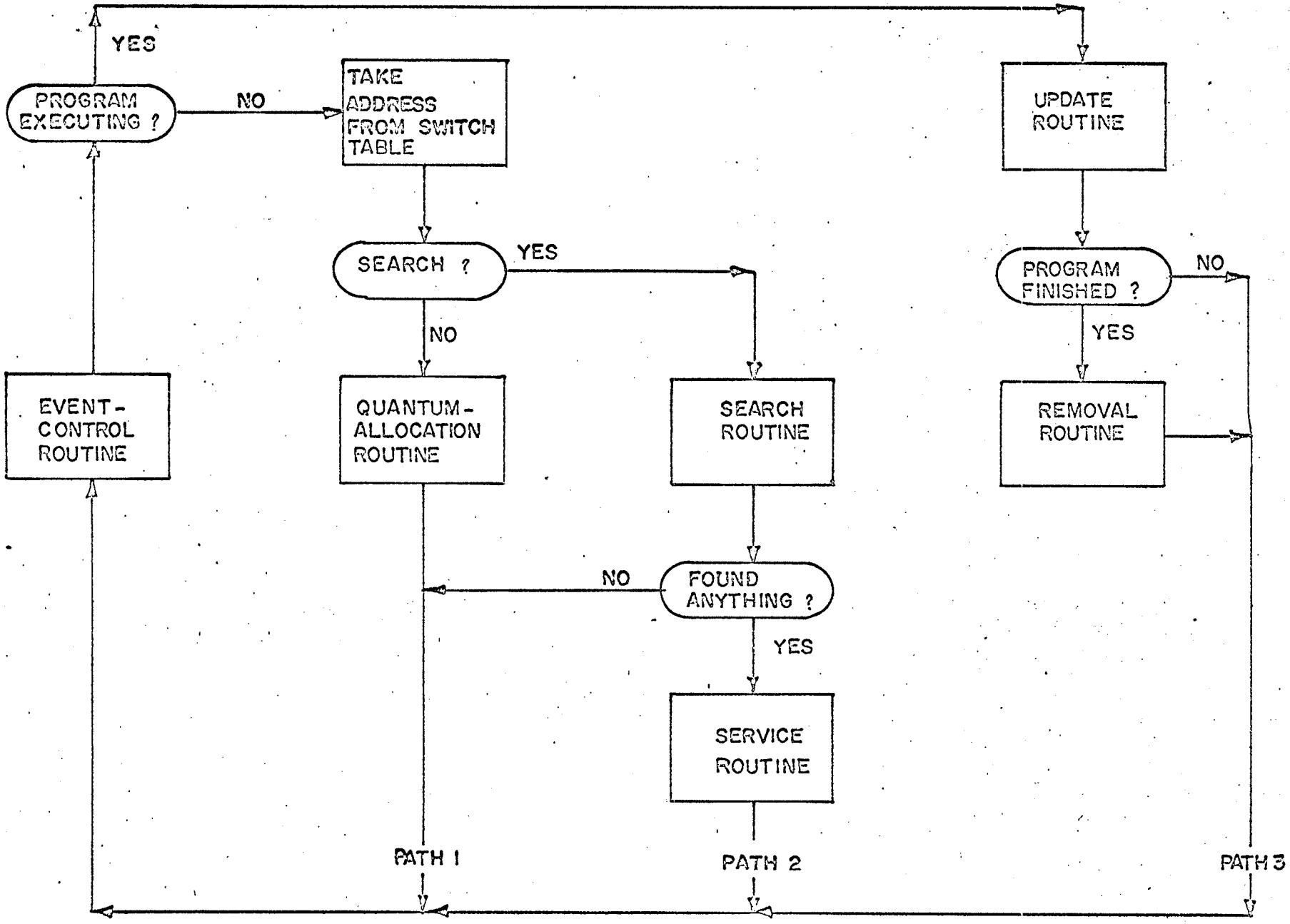


FIGURE 1

FLOW DIAGRAM OF SIMULATION MODEL WITH ITS ROUTINES AND SWITCH TABLE

function of this routine is to monitor the simulated time to determine if a new program is due and to determine if a program from the job queue can be accepted for execution according to the logic of the scheduling algorithm for job selection. One further important function of this routine is to advance the simulated time to the time of the next event. The next event could be the arrival of a new program into core or the termination of servicing of a program which has either completed its time-slice or completed processing during execution of its time-slice.

If no program is executing when the event-control routine has been completed, the supervisor cycling is allowed to continue along the section common to paths 1 and 2, otherwise control is transferred to path 3 where the update routine is entered. When no program is executing, an address is taken from a table (switch table) according to the most recent primary routine entered and the path taken on the most recent cycle. This address determines which of the two primary routines: quantum-allocation or search, should be executed next. The switch table, therefore, embodies the logic of the section of the scheduling algorithm for determining what to do next when a program is interrupted either by completion of its time-slice or by the arrival of a new program into core (section III of the job-scheduling algorithm). The switch table is referenced

after the details of an executing program have been updated.

A small amount is added to the simulated time by the event-control routine in order to allow for the time of execution of the forthcoming cycle. This small amount is the same regardless of the path involved (supervisor-cycle time).

2.1 Switch Table

The switch table which is entered whenever no program is executing, contains the logic of the scheduling algorithm for determining what to do next when one supervisor cycle is finished and the next is about to start. This table simply provides information about the next cycle when the most recent primary routine executed and path taken on the most recent cycle are specified. The switch table is presented as Table I.

The selection from the table of the next primary routine for execution depends on two factors :

(1) the most recent primary routine executed (since the cyclic process does not involve a primary routine on every cycle, the execution of the most recent primary routine might not necessarily occur on the most recent cycle); and

TABLE I

SWITCH TABLE CONTAINING THE DESCRIPTION OF THE NEXT CYCLES

Most Recent Primary Routine Executed	Path Number for the Most Recent Cycle		
	1	2	3
Search	Quantum-Allocation	-	Search
Quantum-Allocation	Search	-	-

(2) the path taken on the most recent cycle, that is, the most recent path-number used (the significance of the path numbers is explained in the following paragraphs).

The flow diagram of Figure 1 shows three paths which connect at the event-control routine. If the search routine is executed in the cycling, and path 1 is taken, it means that no program has been found with an outstanding time-slice. This can occur when all the programs in core have already received their time-slice in a round-robin cycle or when there are no programs in core. A cycling which involves the execution of the quantum-allocation routine and path 1, indicates that the time-slices of all the programs in core have been computed for the next round-robin cycle.

Not all of the entries in the switch table are filled. This arises because only certain combinations of most recent primary-routines and most recent path-number occur during the execution of the simulator. Of the non-occurring combinations of primary routine and path number (indicated by dashed entries in the switch table), the combination of the search routine and path 2 is noteworthy. When the cycling enters the event-control routine after completing path 2 the servicing of a program in core has

been initiated. The termination of this servicing is determined by the logic of the event-control routine which advances the simulated time to the time which occurs first from the following :

(1) the time when the time-slice is finished;

(2) the time when the next program is due; and

(3) the time when the program completes processing during execution of its time-slice.

When a program has been executing, control is always passed to the update routine on path 3 on the next cycle (and possibly the removal routine if the servicing of the program is completely finished). Therefore, there is no need for an entry in the switch table corresponding to a most recent primary-routine "search" and a most recent path-number of 2.

2.2 Penalty Function and Quantum-Allocation Routine

(a) Penalty Function

In a computer installation, where the service facility is demanded by a great many users, the inevitable question is posed of which user to service next, a problem for the selection logic of the scheduling algorithm for queued programs (section (I)). Such a question which is referred to as a priority problem, gives rise to some plan or rule (priority rule) whereby a user's program is selected

in some preferred manner according to the priority class to which it is assigned.

The method by which programs are selected from a waiting list, that is, the priority rule, depends to a large extent on the objectives or goals which are to be achieved. Three common priority rules are outlined below:

(1) First-come-first-served. According to this rule, programs are selected in the order of their arrival. This scheme tends to favour the longest-waiting user and guards against excessive delays. However, no recognition is given to more urgent jobs which may not be at the head of the queue.

(2) Shortest-job-next. This rule gives higher priorities to shorter jobs. If two jobs are equally short, then the job which arrives first is selected. The rule is aimed at reducing the number of programs in a queue. However, it tends to discriminate against long jobs.

(3) Highest-priority-number-next. In this scheme, a program's priority is designated by an integer (priority number) which is assigned to the program before it enters the computer

> system. Selection of programs is made according to priority numbers, the program with the highest-priority number being served next. In the event that two programs have the same priority numbers, the programs are selected on a first-come-first-served basis. The rule fails to recognize that after a period of neglect, the processing of a lower priority program may be as vital as that of a higher priority program which has not been neglected.

Another problem to be considered by the scheduling algorithm, is the determination of the amount of CPU time which a program in core should receive as its time-slice (section (II)). A frequently used technique involves a cyclic discipline within which, each program in core is given a constant slice of CPU time unless its processing is completed during the interval. Hence, control is transferred to the next program in sequence when a time-slice is completed or when a program has finished processing. If a program requires further processing after it has completed its time-slice, it enters a queue and waits for its next service cycle. This type of priority rule is called round-robin scheduling with a constant time-slice. Under this type of scheduling, short jobs are

favoured at the expense of long jobs which are urgent.

From the priority schemes or rules described so far, different scheduling algorithms can be constructed, each having a different priority rule for job selection combined with the round-robin scheduling with a constant time-slice. To decide on which algorithm gives best performance for a particular computer system, it is necessary to have some quantitative evidence of their relative merits. Therefore, a measure of performance of the algorithms is needed.

Greenberger [8] suggested an inverse measure of performance based upon penalty or cost of delay. He considers each program-type to have a separate cost rate curve which represents the variation of the rate at which the cost of delay is accumulating with the time of waiting. Examples of such cost curves are shown in Figure 2.

Curve (a) shows the simplest case of the cost of delay accumulating at a constant rate, c, throughout the waiting period of a program. If the waiting time is w, then the total accumulated cost is cw. However, this case does not give adequate attention to the growing wait suffered by longer programs if one believes the next time unit (say minute) of waiting is worse than the previous time unit.

For the latter problem, curves like (b), (c) and

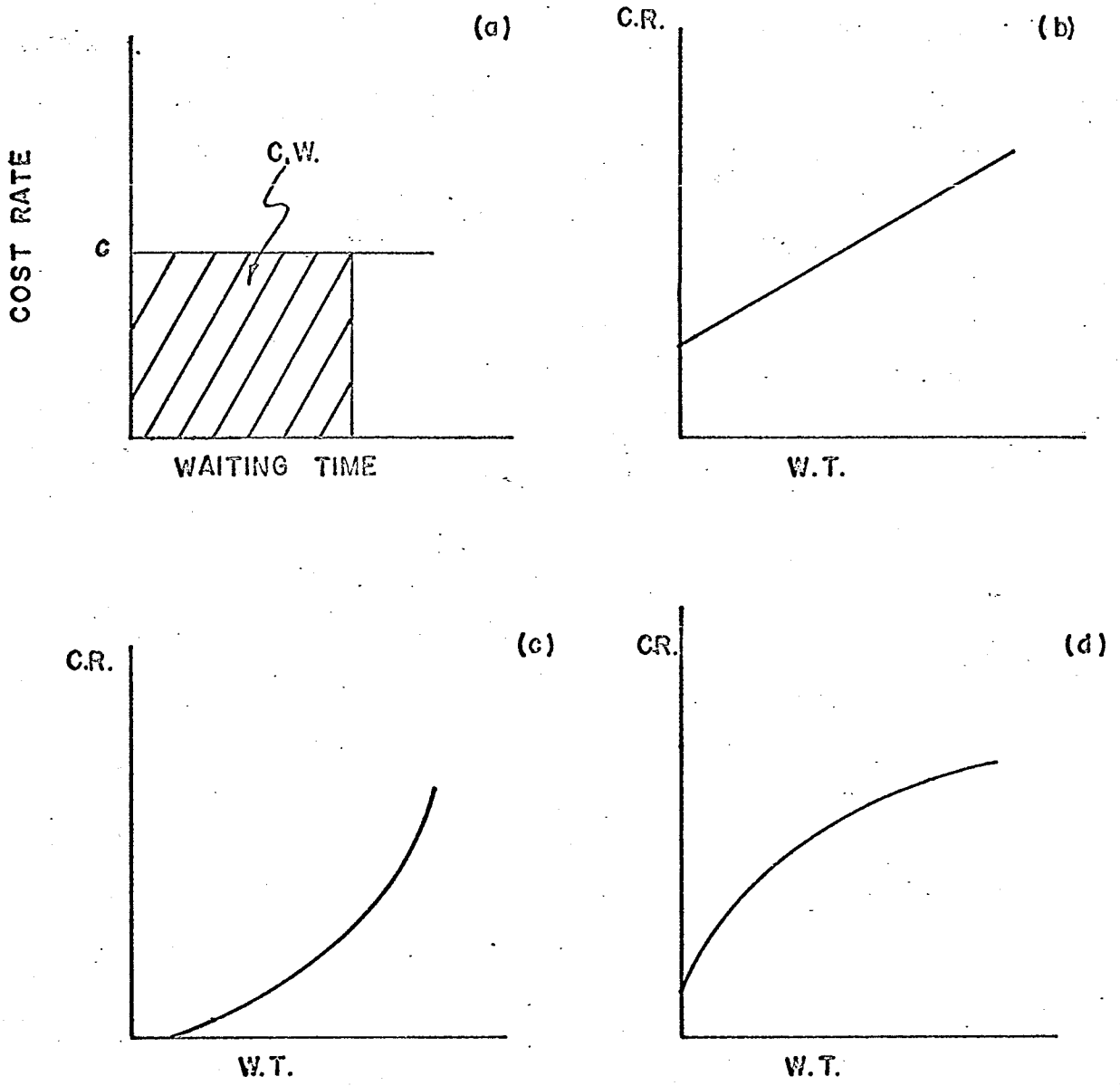


FIGURE 2

EXAMPLES OF COST CURVES

(d) in Figure 2 are necessary for the costing. In this case, the total accumulated cost is obtained by taking the integral of the curve over the period of waiting.

The form of these cost curves, therefore, can be used to express the types of attention required by programs insofar as the height of the curve reflects a program's importance and the slope reflects a user's intolerance to delay.

For the scheduling algorithm, the cost curve needs a few modifications. Since in most cases, the processing time required by a program is not known beforehand, the cost curve, as a function of real time, does not allow any mechanism whereby the scheduling algorithm or the scheduler may knowingly prevent a program from receiving a final penalty. Furthermore, if a program should have a penalty or cost accumulated at some stage prior to the completion of its processing, this penalty will continue to increase with time as long as the program has not completed processing. The scheduler, in this case, has no means of eliminating or reducing the intermediate penalty. The best the scheduler can do is to give larger time-slices to the program and hope that the final penalty is not much higher.

Rather than a function solely of real time, therefore, the penalty is made a function of the lack of

attention received by a program. The lack of attention to a program i is defined by

$$u_i = T_i/\tau_i \qquad (1)$$

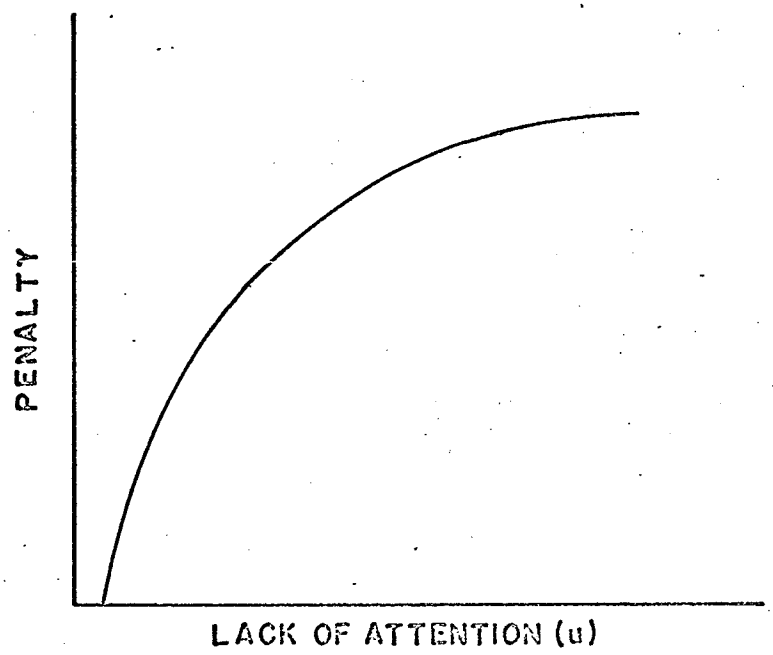
where T_i is the real time which has elapsed since the desired time of entry of the program into the computer system and τ_i is the sum of the central processor time and device time devoted to the program since it entered the system. Since no input or output operation is considered in this simulation, in order to take care of the indeterminate case of u_i when $\tau_i = 0$, a fixed value of 1.0 second is used as the device time of each program. This value is assigned at the time when a program enters the system.

The lack of attention u_i can increase or decrease within real time. It is a relative function which is independent of the total processing time required by a program.

Another modification which is less serious, is to use an accumulative penalty variable instead of a cost rate variable. This removes the need for integration of the function.

Figure 3 shows two examples of the type of penalty curves which are used in the simulation. Curve (a) is used for a fast response program where the penalty increases very rapidly for low values of u_i and very slowly for high

(a)



(b)

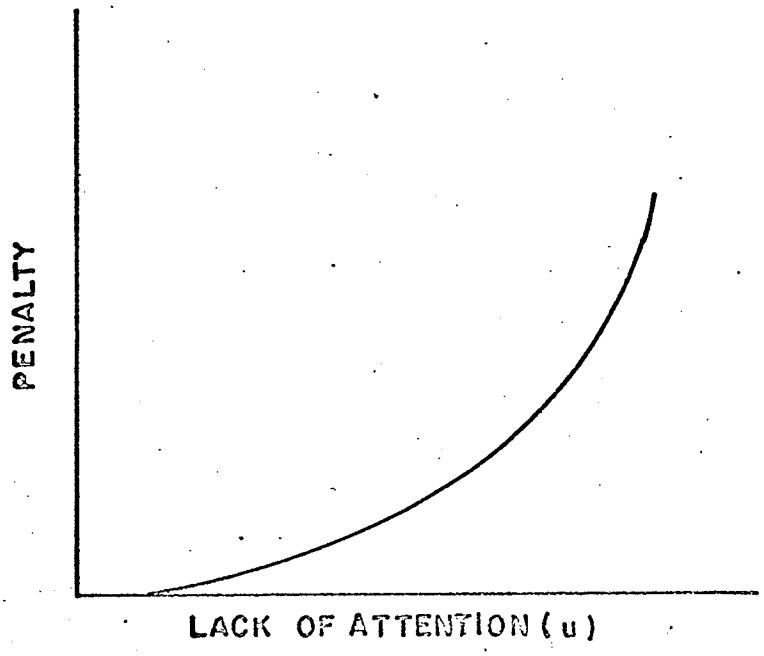


FIGURE 3

TWO EXAMPLES OF THE TYPES
OF PENALTY FUNCTION

values of u_i. Curve (b) is for a slow response program where the penalty value increases slowly at first but rises very rapidly as u_i gets larger.

As in the case of Greenberger's cost curve, each priority class of users has associated with it a penalty curve which rises to a different extent, according to the type of attention required, as the lack of attention increases. The degree of attention demanded by the user is reflected in the slope of the curve. The priority classes are designated by integers (priority numbers), where priority i is considered as more important than priority j if $i < j$.

The current accumulative penalty (or simply penalty) of program i at any time t may be expressed by

$$p_i(t) = f_{\pi_i}(u_i(t)) \tag{2}$$

where π_i is the priority number of program i and f_{π_i} is the penalty function associated with priority π_i.

The total operating penalty of the system can be calculated at any given time by performing the summation

$$P(t) = \sum_i p_i(t), \tag{3}$$

that is,

$$P(t) = \sum_i f_{\pi_i}(u_i(t)). \tag{4}$$

The penalty values are used for the following purposes :

(1) To serve as a basis for program selection from a queue with the program having the highest penalty being selected first. If two or more programs have the same highest penalty, the programs are selected on a first-come-first-served basis.

(2) To be used as a basis for determining the amount of CPU time a program should receive as its time-slice in a variable time-slice scheme. Under this scheme, the size of the time-slice is dependent on the program's current penalty and the total operating penalty of the system. (A more detailed description of this is given in the next section, "Quantum-Allocation Routine").

Further, a measure of the efficiency of a scheduling algorithm is taken as the sum of the final penalties of a number of completed programs. If p_i^* is the final penalty of the finished program i, then the sum of the final penalties of N finished programs may be denoted by

$$P_N^* = \Sigma_{i=1}^{N} p_i^* . \qquad (5)$$

A simple comparison between two scheduling algorithms

may be effected by evaluating P_N^* for the two algorithms using the same N programs and the same set of penalty functions.

(b) Quantum-Allocation Routine

This routine which is entered immediately before the start of every round-robin cycle, determines the amount of CPU time that each program is to be allocated as its time-slice (quantum-allocation scheme). It allocates CPU time according to two scheduling algorithms which are considered by the simulation study. Both algorithms have the same piece of logic for job-selection from a waiting list, namely, to select the program with the highest penalty, but use different types of quantum-allocation scheme with their round-robin scheduling.

One type is the constant time-slice allocation scheme in which a fixed amount of CPU time is allocated to each program in core. This amount of CPU time is determined from the relationship

$$q_i = t_{rr}/n \qquad (6)$$

where q_i is the constant time-slice, t_{rr} is the intended round-robin time[5] (and is simply referred to as the

[5]It must be noted that the actual round-robin time differs from the intended round-robin time, t_{rr}. This difference is due to the amount of supervisor cycle time incurred in the round-robin cycle and to the unused portion of the allocated time-slice in the case where a program completes processing before expiration of its time-slice.

round-robin time) and n is the maximum number of jobs which can be scheduled simultaneously by the algorithm (in this case, n = 3).

The round-robin scheduling with a constant time-slice, however, gives preferential treatment to shorter jobs at the expense of penalizing longer jobs which are urgent. There is an apparent need, therefore, for an algorithm which incorporates the interests of the individual users. In an attempt at solving the latter, it appears reasonable to modify the round-robin scheduling with a constant time-slice to a round-robin scheduling with variable time-slice. This, therefore, gives rise to the second type of allocation scheme which considers variable time-slice.

The calculation of the variable time-slice involves a two-stage process. First, the total penalty received by all the programs in core is determined by computing the current accumulative penalty of each program. Secondly, based on these penalty values, the time-slices of the programs are determined.

The time-slice allocated to program i by the quantum-allocation scheme at time t is calculated from the relationship

$$q_i = (p_i(t)/P(t))t_{rr} \qquad (7)$$

so that the round-robin time, t_{rr}, is divided among the

programs according to their current penalties. Two exceptions to the above should be noted.

(1) When the total operating penalty, P(t), is zero, t_{rr} is divided equally among the programs.

(2) If a program's quantum is calculated to be less than a fixed minimum amount of CPU time (minimum time-slice), then the minimum amount is allocated as the quantum of the program.

2.3 Event-Control Routine

The event-control routine which is entered on every cycle, performs the following basic functions.

(1) Assuming that a queue of waiting programs exists, the routine first calculates the penalty values of all the programs in the queue. It then selects the program with the highest penalty value and scans through the execution list to see if there is a space available for the program. If a space is found, the program is entered on the execution list, otherwise it is returned to the queue.

(2) It ascertains whether a new program is due or not by checking the program's scheduled arrival time against the simulated time. If it is established that a new program is due, the routine determines if it can be accepted into core by searching for a space on the execution list. In the case where the list is full, the new program is placed in the queue.

(3) It advances the simulated time to the time of the next event. The time of the next event is the smaller of the following :

(i) the time of arrival of a new program which can be accepted into core; and

(ii) the time of service termination which could be either the time when a time-slice is completed or the time when a program has completed processing prior to the expiration of its allocated time-slice.

(4) In the case where no program is executing when the event-control routine has been completed, the supervisor cycling is

allowed to continue along the section common to paths 1 and 2.

On each exit from the event-control routine, the supervisor cycle time is added to the simulated time to allow for the execution time of the forthcoming cycle.

Figure 4 gives a description of the flow logic of the event-control routine.

2.4 Search and Service Routines

(a) Search Routine

The purpose of this routine is to determine the next program to be serviced in the round-robin cycle. When the routine is entered, it scans through the execution list sequentially and selects the first program it finds with an outstanding time-slice. If a program is found, the service routine is entered to initiate servicing of the program. If no program is found with an outstanding time-slice, then path 1 is taken on exit from the search routine.

(b) Service Routine

When a program has been found with an outstanding time-slice by the search routine, control is passed to the service routine which initiates the servicing of the program. A record is also made of the time when the time-slice is completed or the time when the

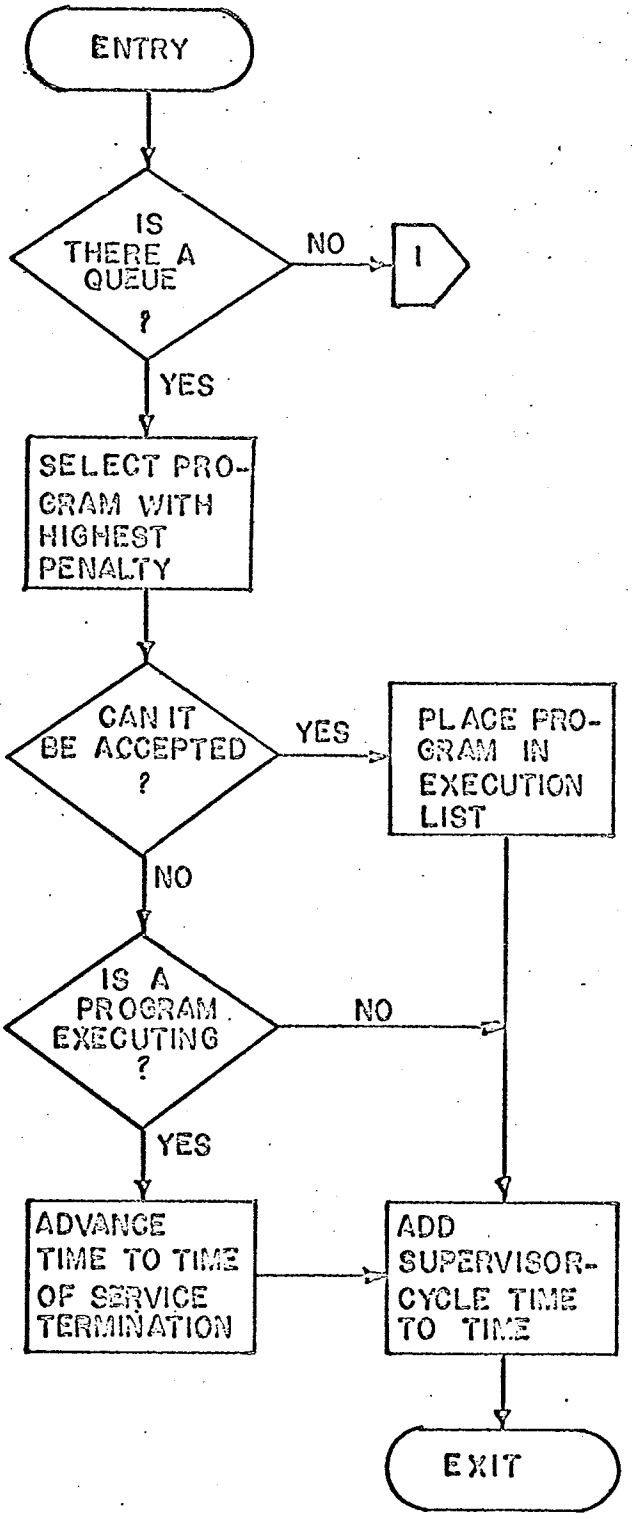


FIGURE 4

FLOW DIAGRAM OF THE EVENT-CONTROL ROUTINE

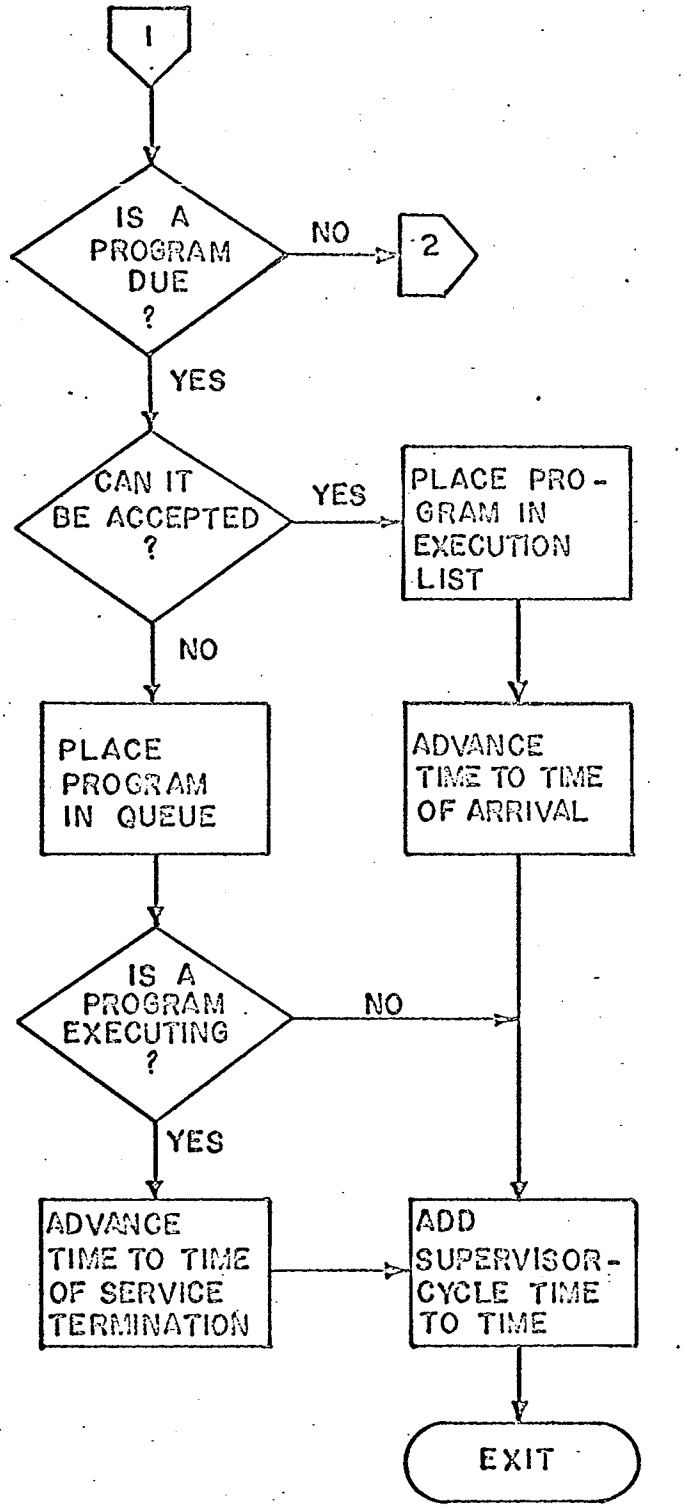


FIGURE 4

FLOW DIAGRAM OF THE EVENT-CONTROL ROUTINE
(Continued)

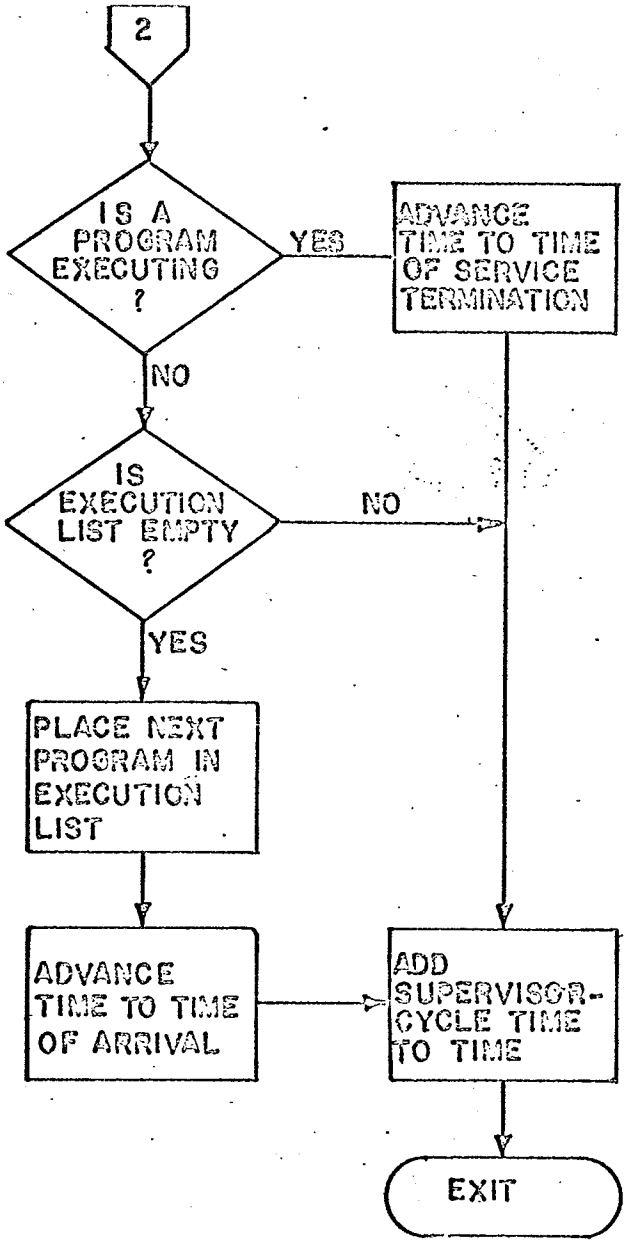


FIGURE 4

FLOW DIAGRAM OF THE EVENT-CONTROL ROUTINE
(Continued)

program completes processing, depending on which occurs first. On exit from this routine, path 2 is taken in the cycle.

2.5 Update and Removal Routines

(a) Update Routine

The update routine is executed whenever the execution of a program has been interrupted. The routine updates the accumulated central-processing time for the program in the execution list. If execution of the program is terminated before its time-slice is completed owing to the arrival of a new program into core, an updated time-slice, representing the remainder due to the program in the current round-robin cycle, is calculated.

(b) Removal Routine

The removal routine is executed whenever the central-processing requirements of one of the programs in core has been met. The routine outputs on a card for subsequent statistical analysis, the program number, the priority, the current simulated time and the amount of CPU time requested. After this information is output, the associated information stored in the execution list is erased so that another program can take its place on a later supervisor cycle.

CHAPTER III

VARIABLE PARAMETERS FOR THE SIMULATION MODEL

The variable parameters, which are input at the start of the simulation, are the penalty functions, job-load parameters, supervisor-cycle time, round-robin cycle time and minimum time-slice.

3.1 Penalty Functions

Five different penalty functions are used for the simulation. Each function is associated with one of the five priority classes which the scheduling algorithm presently admits.

The penalty functions are arbitrary analytic functions which are specified by using the mathematical relationship

$$f_{\pi}(u) = 999\ (((u-1)/14)^{b})\ +\ 1 \qquad (8)$$

where $b = 2^{(\pi-3)}$, π is the priority number which takes on integral values 1, 2, 3, 4 and 5, and u, the lack of attention, is only considered over the closed interval [1,15]. Thus if the calculated value of u lies outside this interval, u is set equal to 1 or 15 depending on whether u is less than 1 or greater than 15 respectively. These penalty functions remain unchanged throughout the entire simulation study.

Figure 5 illustrates the graphs of the five

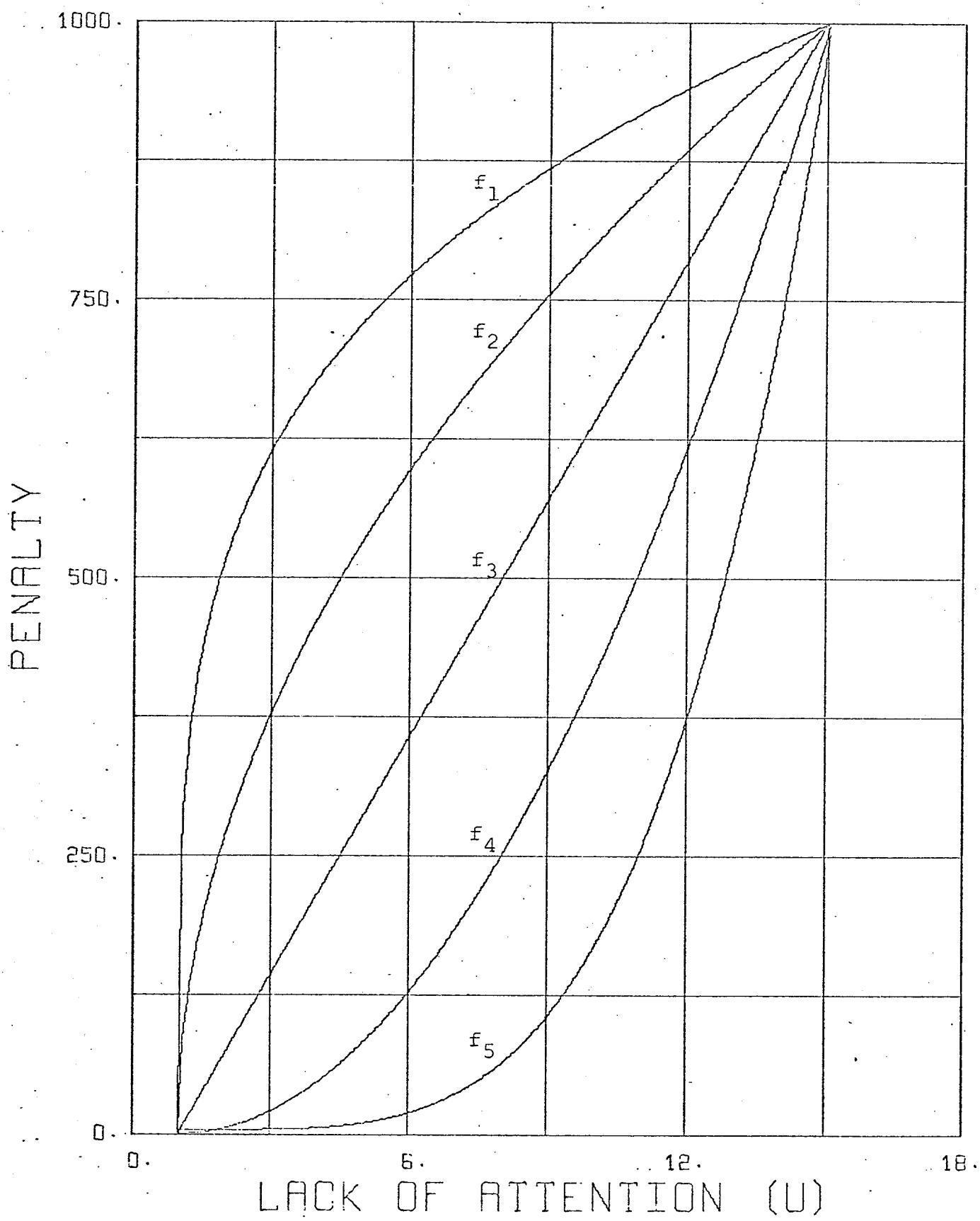


FIGURE 5

GRAPHS OF PENALTY FUNCTIONS USED IN THE SIMULATION

functions. The notation, f_i, denotes the penalty function associated with programs of priority class i. f_1 and f_2 which are functions for the shorter response programs with priorities 1 and 2, rise very steeply at low values of u. On the other hand, f_4 and f_5, which are functions for longer response programs with priorities 4 and 5, rise very rapidly at higher values of u.

3.2 Job-Load Parameters

The work-load environment for the simulation study is created by a job-generator routine that simulates a series of entities called jobs or programs using Monte-Carlo technique. Each job description consists of the following items :

(1) The job number

(2) The priority class number

(3) The amount of central processing time required or requested

(4) The arrival time

The CPU time and the interarrival time of jobs within a particular priority class are selected at random from a normal distribution of each. The normal distributions are specified by a mean and a standard deviation which are input parameters. Five priority classes are considered. Thus, this description of the job load gives

rise to twenty independent variables. In order to be able to study the effects of changes in the scheduling algorithm upon efficiency, as a function of job load, it is reasonable to reduce the number of independent variables to a manageable level by introducing restrictions.

The mean CPU times for the five priority classes are linked together according to the equation

$$m_{i+1} = km_i \tag{9}$$

where i = 1,2,3,4 and k is a proportionality constant. This, therefore, allows the complete set of mean CPU times to be calculated from the specification of k and the mean CPU time of priority 1 programs (m_1 seconds of CPU time per program).

The mean interarrival times are linked in a similar manner according to the equation

$$r_{i+1} = kr_i \tag{10}$$

where i = 1,2,3,4 and k is the same proportionality constant as in equation (9). This also allows the complete set of mean interarrival times to be calculated from the specification of k and the mean interarrival time of priority 1 programs (r_1 seconds per program).

All standard deviations are taken to be 25% of the corresponding mean values.

Values of the three independent parameters k, m_1 and r_1 are used to completely describe the job loads which are to be used in this simulation study.

In order to have some control over the conditions which give rise to system overloading, it is desirable to have some guide for estimating the proportion of the processor's capacity which is demanded by programs of each priority class over a given time interval. Using parameters m_i and r_i, the proportion of the processor's capacity demanded by priority class i for a given time interval can be approximated by means of the formula

$$d_i = m_i/r_i. \tag{11}$$

Thus for a specified d_i, the proper choice of r_i or m_i can be determined depending on whether m_i or r_i is given respectively. For example, if it is desired that each priority class demands only 1/5 of the total processor's capacity over a particular time interval, that is, $d_i = 1/5$, this can be effected, though loosely, by using the following equation

$$m_i/r_i = 1/5 \tag{12}$$

where $i = 1,2,3,4,5$. Hence, from equation (12), the proper choice of r_i can be obtained if m_i is given and

vice versa.

Furthermore, since from equations (9) and (10)

$$\frac{m_{i+1}}{r_{i+1}} = \frac{m_i}{r_i} , \qquad (13)$$

it follows that for a given m_1 and r_1, the proportion of the processor's capacity demanded by each priority class for a particular time interval is the same, namely, m_1/r_1. Thus, only the choice of m_1 and r_1 needs to be considered.

The method by which normally distributed random variates are generated using the Central Limit approach is described in Appendix C.

Ten separate sequences of random numbers are generated for the ten normal distributions (2 for each priority class: CPU time required and interarrival time). All ten starting values used by the random number generator for these sequences are the same. This value is 157832165. Its choice is arbitrary with the exception that it must be odd and less than 2^{31}. A discussion of the random number generator can be found in Appendix B.

The normal distributions used were truncated at the $\pm 4\sigma$ limits where σ is the standard deviation. Since all standard deviations are taken as 25% of their corresponding means, the lower and upper limits of each distribution are in fact 0 and 2μ respectively, where μ is the

mean. Thus within priority class i, the upper and lower limits of the normal distribution for the requested CPU time are 0 and $2m_i$ respectively, and for the interarrival time, are 0 and $2r_i$ respectively. Hence, if a normally distributed random variate generated lies outside these limits, it is rejected and another is generated.

The choice to truncate the normal distributions at the $\pm$ 4σ limits appears reasonable. For the lower limit, a normally distributed random variate less than $(\mu - 4\sigma)$ is undesirable since it is negative, while for the upper limit, a normal variate obtained in the region beyond $(\mu + 4\sigma)$ is likely to be unreliable (see [10]). Further, with limits of $(\mu \pm 4\sigma)$, the proportion of the normal distribution considered is about 99.98%.

A sample listing of 20 programs that have been created by the job generator routine is shown in Appendix D.

3.3 Supervisor-Cycle Time

The supervisor-cycle time is the time taken to execute a cyclic path in the simulation model (see Figure 1). Strictly speaking, this time which contributes to the supervisor overheads, varies according to which path is taken. However, for convenience, one fixed value is assumed for the supervisor-cycle time. It appears

reasonable to use a time of 0.01 second. On the IBM 360/65 computer system, this value is equivalent to approximately 3000 instructions.

In order to get an idea of the supervisor-cycle times which are involved during the simulation process, two simple examples are described illustrating the number of supervisor cycles necessary for the case under consideration.

Example 1. Consider the situation where a program enters core which is empty, completes its processing within the first time-slice it receives, and is then removed from core.

Since core is empty prior to the arrival of the program, the most recent primary routine is taken as "search" and the most recent path is taken as path 1. The combination of the search routine and path 1 is used to determine the next primary routine to be executed.

After the program has entered core, one supervisor cycle is taken to execute the quantum-allocation routine for calculating its time-slice, another cycle is needed to enter the search routine followed by the service routine which initiates servicing of the program and a final cycle is required to execute the update and removal routines when the program completes processing. Thus, three supervisor cycles are taken for the entire

operation. The total supervisor-cycle time involved in this case is 0.03 second.

If the program in the above example had required more than one time-slice, say two time-slices, to complete processing, then after the update routine is executed on the third cycle mentioned in the foregoing paragraph, a fourth cycle is used to enter the search routine to determine if any other program is to be serviced. In this case no other program is found. On the next (fifth) cycle, the quantum-allocation routine is executed followed by a further (sixth) cycle to execute the search and service routines with a final (seventh) cycle to execute the update and removal routines. Thus, for the second time-slice received by the program, an extra four supervisor cycles are required. The total supervisor cycle time involved in this case, therefore, is 0.07 second.

In general, if the program mentioned in Example 1 requires more than one time-slice to complete processing, then for each additional time-slice after the first time-slice, four supervisor cycles are required.

Example 2. Consider the same situation as in Example 1 with the addition that a second program enters core while the first one is executing.

In this case, the first two cycles are the same

as those in Example 1. However, when the first program is interrupted during its execution by the arrival of a new program in core, two cycles are necessary to resume processing of the first program; namely, one cycle for entering the update routine to calculate the updated time-slice of the first program and another cycle for executing the search and service routines to re-initiate servicing of the first program. When the program completes processing, a further cycle is needed for entering the update and removal routines. Five supervisor cycles are involved in the processing of the first program, thereby contributing 0.05 second to the supervisor overheads.

3.4 Round-Robin Cycle Time

For both the constant time-slice and the variable time-slice allocation schemes associated with the round-robin scheduling, the value of the intended round-robin time, t_{rr}, is read in as data. In the constant time-slice scheme, the quantum allocated to a program is simply $t_{rr}/3$. For the variable time-slice scheme, t_{rr} is divided up among the programs in core according to their current penalties.

It is very likely that the efficiency of the algorithm, that is, the value of P_N^* or $\Sigma_{i=1}^{N} p_i^*$ (where

N programs are considered), is dependent upon the size of t_{rr}. If t_{rr} decreases, the time-slices are also proportionately decreased and the frequency of switching from one program to another increases. This results in an increase in supervisor overheads which tend to degrade the processing efficiency of the system, that is, to increase the value of P_N^*. On the other hand, when t_{rr} is large, the supervisor overheads are low but the shorter programs, which tend to have penalty curves that increase very sharply at low values of u, are neglected more than when t_{rr} is small. This occurs when the average quantum, q, is much larger than the mean value of the CPU time of priority class 1 programs, that is, $q >> m_1$. These effects were noted by Greenberger [8] in an analytical study using linear penalty curves.

3.5 Minimum Time-Slice

In the round-robin scheduling with variable time-slice, a program's calculated time-slice, which is determined according to the program's current penalty, could be so small that it is not efficient for the system to process the program for such a short time interval due to the relatively high overheads involved. To illustrate this point, consider a simple case where only one program is in core. As seen from the discussion in the section

"Supervisor-Cycle Time" three supervisor cycles are necessary to carry out the operations of a round-robin cycle, which involve calculation of the time-slice, initiation of servicing and execution of the update routine after the time-slice is completed. If the time-slice in this case is approximately equal to the supervisor-cycle time, then only about 1/4 of the CPU time is useful, since out of the total CPU time of 0.04 second, only about 0.01 second is devoted to the processing of the program.

In an attempt to avoid excessive loss of CPU efficiency, the size of the time-slice is not allowed to fall below 1/100 of the round-robin cycle time. This amount is called the minimum time-slice. Thus, whenever a calculated time-slice falls below this value, the minimum time-slice is allocated instead.

It seems likely that the size of the minimum time-slice would affect the efficiency of the algorithm, that is, the value of P_N^*. As the size of the minimum time-slice increases to t_{rr} (the round-robin time) the round-robin scheduling with variable time-slice becomes a round-robin scheduling with a constant time-slice. If the minimum time-slice continues to increase until it becomes much larger than t_{rr}, the scheduling algorithm approaches a batch processing type of scheduling with a first-in-first-

out discipline,[6] where long jobs are favoured at the expense of short ones. On the other hand, if the minimum time-slice is very small, supervisor overheads are increased thereby degrading the efficiency of the algorithm, that is, increasing the value of P_N^*.

3.6 Output of the Simulation Program

Both intermediate and final details of each program which was processed are punched out on cards. The intermediate program details are output after each execution of the quantum-allocation routine, while the final program details are output when a program has completed its entire processing and has been removed from the execution list.

The intermediate program details consisted of :

(1) Program number

(2) CPU time required or requested

(3) Initial entry time into the system

(4) Accumulated CPU time received

(5) Accumulated combined processor and device time

---

[6] In a first-in-first-out discipline, programs are serviced on a first-come-first-served basis with each program being serviced to completion before control is passed to the next program.

(6) Current penalty value

(7) Amount of time-slice allocated

(8) Time when time-slice was allocated

The final program details have items (1) to (5) above and in addition, the following two items:

(9) Final penalty

(10) Exit time of program from the system

A listing of the final details of 20 programs whose initial job description are shown in Appendix D is shown in Appendix E.

CHAPTER IV

PERFORMANCE OF THE SIMULATION MODEL

4.1 Study of the Behaviour of Two Types of Scheduling Algorithm

The relative performance of two scheduling algorithms was estimated by simulating the execution of identical job streams with each algorithm in turn, and observing the effect on the measure of efficiency, P_N^*.

The two algorithms studied were :

(i) fixed time-slice (FTS) and

(ii) variable time-slice (VTS).

Both algorithms use round-robin scheduling.

Table II shows the input values that were used for the variable parameters.

(i) Fixed Time-Slice Algorithm

The fixed time-slice algorithm accepts programs from the waiting list according to which program has the highest penalty. If two or more programs have the same highest penalty, then the first of these programs to arrive is selected. For programs which are in core, the time-slices for these programs are calculated according to formula (6) (see section "Penalty Function

TABLE II

INPUT VALUES FOR THE VARIABLE PARAMETERS USED IN THE STUDY OF THE BEHAVIOUR OF TWO TYPES OF SCHEDULING ALGORITHM

VARIABLE PARAMETERS	VALUE
Job-load parameters:	
Proportionality Constant, k	4.0
Mean value of CPU time for priority class 1, m_1	10.0 secs.
Mean value of interarrival time for priority class 1, r_1	30.0 secs.
Round-robin time, t_{rr}	3.0 secs.
Minimum time-slice	0.03 sec.
Supervisor-cycle time	0.01 sec.

and Quantum-Allocation Routine", page 29), namely,

$$q_i = t_{rr}/n$$

which takes no account of the current penalties of the programs. The variable n refers to the maximum number of programs which can be scheduled simultaneously and in this case, its value is 3.

From the input values of the job-load parameters, m_1 (the mean of the normal distribution associated with the requested CPU time for priority 1 programs) and k (proportionality constant) in Table II, the means (m_i, where i = 2,3,4,5) of the other four priority classes of programs, were derived by using the following relationship

$$m_{i+1} = km_i \qquad (i = 1,2,3,4). \qquad (14)$$

The standard deviations (s_{m_i}) were calculated by taking 25% of the means. The lower and upper limits of the ranges for the requested CPU times considered, were determined by using $m_i - 4s_{m_i}$ and $m_i + 4s_{m_i}$ respectively where i denotes priority class i. Since s_{m_i} is 25% of m_i, the lower limit of a range is in fact zero which is a reasonable lower limit since negative values were not meaningful. The upper limit ($m_i + 4s_{m_i}$) appears reasonable also, since values lying in the region beyond this limit were not likely to be reliable

(see [10]).

Table III gives the values for the mean, the standard deviation and the range of the requested CPU time of each priority class.

Similarly, by using r_1 and k, the values for the means, the standard deviations and the ranges of the interarrival times of all the priority classes were determined. The result is shown in Table IV.

The simulation run was terminated after a simulated time period of 1202 seconds. During this time period, 48 programs completed processing, 2 programs were in core unfinished and 20 programs were in the queue. No priority 4 or 5 programs arrived in this run and therefore only the first three priority classes of programs were effectively considered.

Final and intermediate details of each program were collected and a graph of penalty versus time was plotted. Figure 6 shows a portion of this graph over the interval 450 seconds to 1000 seconds. The number or numbers on each curve represent the priority class of the associated program. The curves with their priority number marked at the beginning and end are for programs which completed processing, while curves with their priority number written only at the start are for programs which did not complete processing at the end of

TABLE III

VALUES OF THE MEAN, STANDARD DEVIATION AND RANGE OF THE REQUESTED CPU TIME FOR FIVE PRIORITY CLASSES

Priority Class	Mean CPU Time (secs)	Standard Dev. (secs)	Range of CPU Time (secs)
1	10.0	2.5	0.0 - 20.0
2	40.0	10.0	0.0 - 80.0
3	160.0	40.0	0.0 - 320.0
4	640.0	160.0	0.0 - 1280.0
5	2560.0	640.0	0.0 - 5120.0

TABLE IV

VALUES OF THE MEAN, STANDARD DEVIATION AND RANGE OF THE INTERARRIVAL TIME OF FIVE PRIORITY CLASSES

Priority Class	Mean Inter-arrival Time (secs)	Standard Dev. (secs)	Range of Inter-arrival Time (secs)
1	30.0	7.5	0.0 - 60.0
2	120.0	30.0	0.0 - 240.0
3	480.0	120.0	0.0 - 960.0
4	1920.0	480.0	0.0 - 3840.0
5	7680.0	1920.0	0.0 - 15360.0

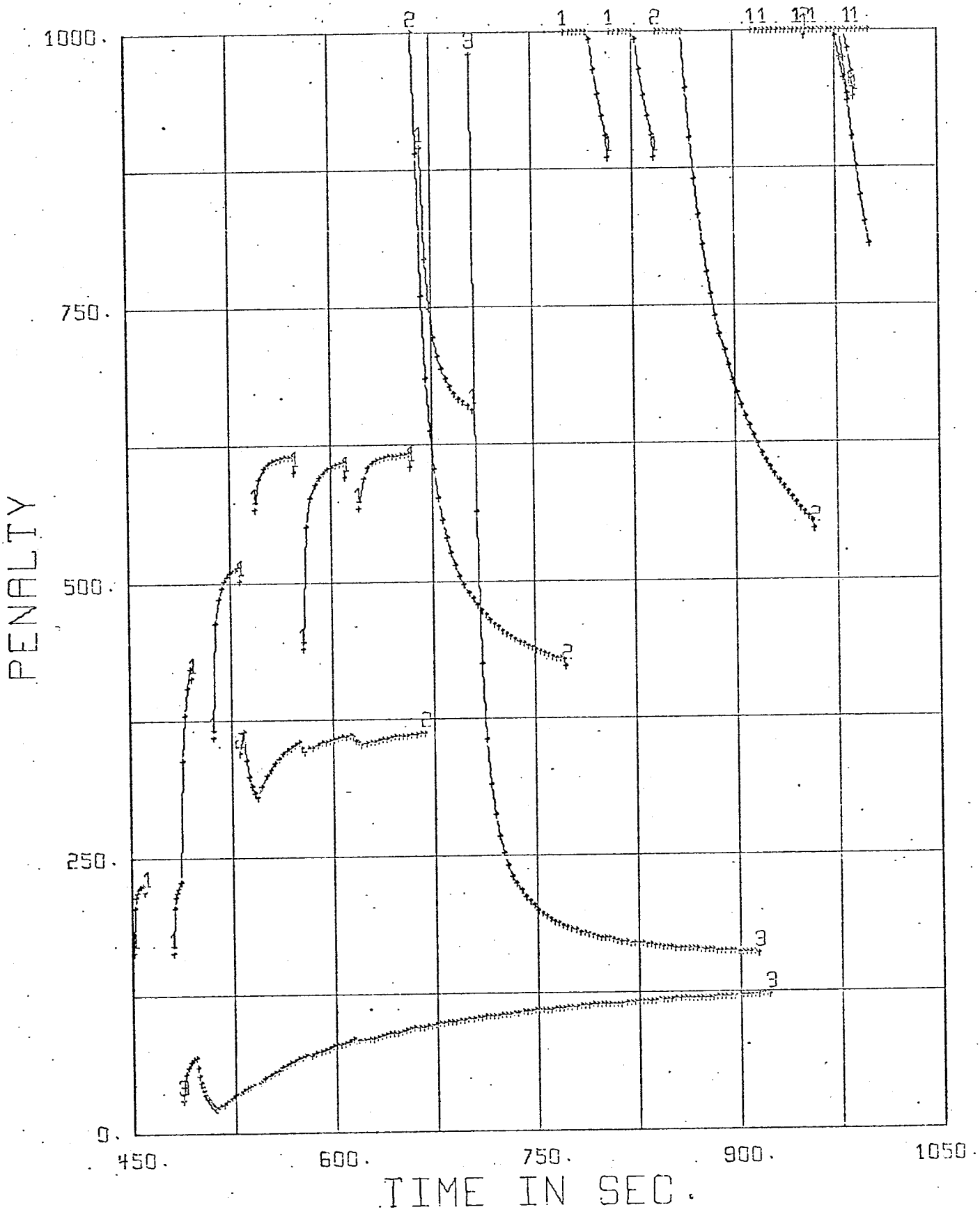


FIGURE 6

PENALTY VS. TIME OF PROGRAMS PROCESSED IN THE INTERVAL 450 - 1000 SECS. USING THE FTS ALGORITHM

1000 seconds.

One noticeable feature about the behaviour of the curves in Figure 6 is that when core is full curves belonging to the same priority class seem to approach a common penalty value which is specific to that priority class. Thus, there seem to be three different common values corresponding to the three priority classes studied. The common penalty values seem to decrease with descending order of priority. This characteristic may be attributed to the associated penalty functions themselves.

From Figure 6, the common penalty value for priority 1 programs is about 625 penalty units. For priority 2 programs, it is about 390 penalty units, while priority 3 programs approach an approximate value of 150 penalty units. Thus it can be seen that for programs of the same priority class, if their curves are above the common value for that class, they tend to decrease and approach this value while curves below the common value tend to rise to meet it.

In order to determine whether the observation of these common values could be supported by further analysis, a simple analytical study was performed.

With T_i and τ_i initially at zero and a program mix consisting of a priority 1, a priority 2 and a

priority 3 program, the current penalties of the three programs, each receiving a quantum of 1/3 of the round-robin time were calculated with supervisor overheads disregarded. The penalty values obtained, plotted against time, are shown in Figure 7. The numbers in parentheses at the end of each curve, from left to right, indicate the priority number, the round-robin time and the quantum respectively.

Each curve in Figure 7 is approaching a separate penalty level. For the priority 1 program, the level is about 600 penalty units. For the priority 2 program, it is about 375 penalty units, while for the priority 3 program, it is about 140 penalty units. These three values are relatively close to the common values observed in Figure 6, the difference being mainly due to supervisor overheads. Hence this supports the view stated earlier.

(ii) Variable Time-Slice Algorithm

The variable time-slice algorithm accepts programs from the waiting list in the same manner as the fixed time-slice algorithm. However, the amount of time-slice a program receives is dependent on its current penalty and on the operating penalty of the system. The time-slice to be allocated to a program at time t is determined by formula (7) (see section "Penalty Function and Quantum-

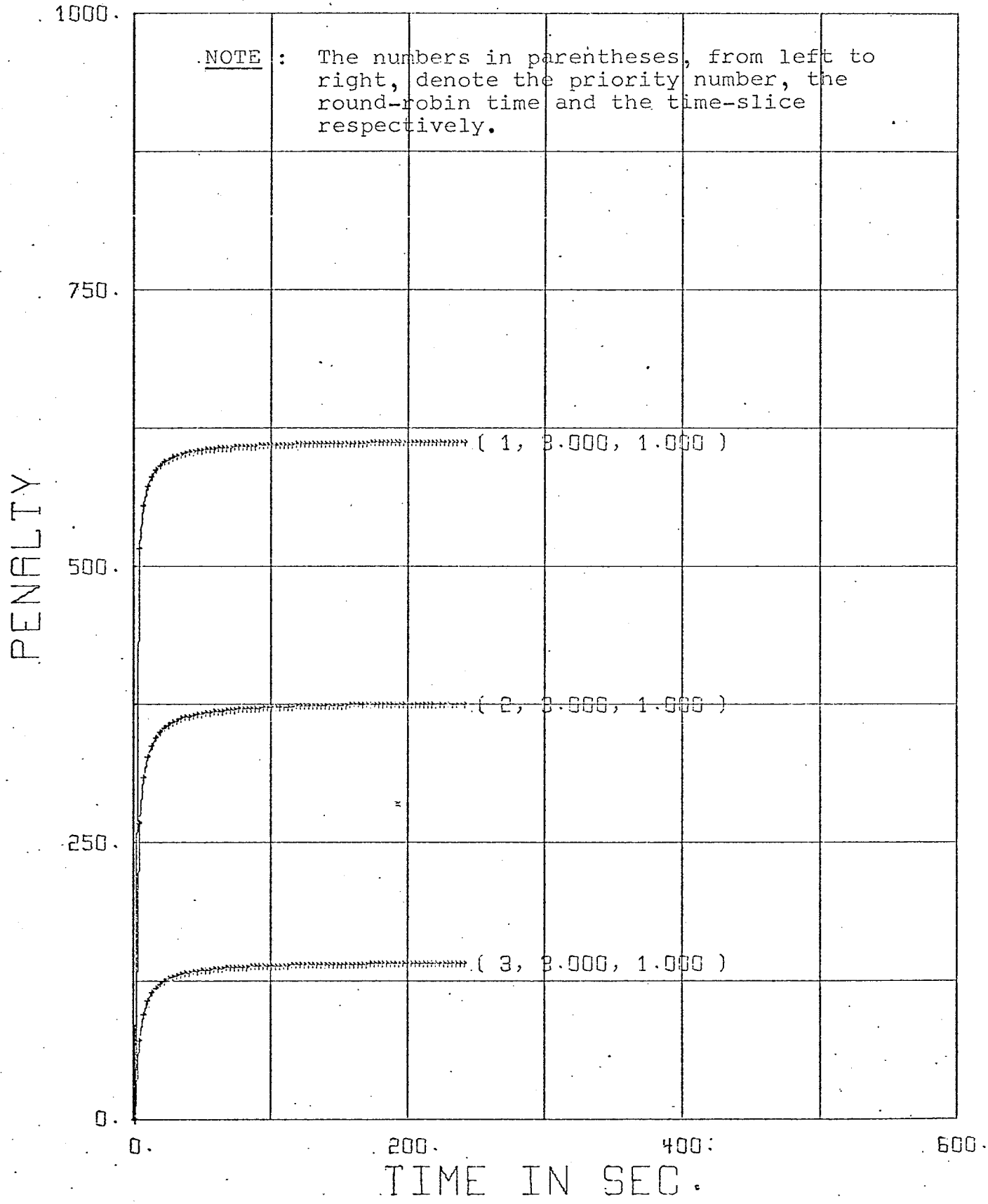


FIGURE 7

PENALTY VS. TIME OF THREE PROGRAMS OF DIFFERENT PRIORITY CLASSES USING THE FTS ALGORITHM

Allocation Routine", page 30), that is,

$$q_i = (p_i(t)/\Sigma_i p_i(t))t_{rr}$$

where $p_i(t)$ is the current penalty of program i, $\Sigma_i p_i(t)$ is the total operating penalty and t_{rr} is the round-robin time.

Using the same input values for the variable parameters shown in Table II, the simulation run was terminated after a simulated time period of 1584 seconds. In this time interval, 65 programs completed processing, 3 programs did not complete execution and 1 program was in the queue. As in the fixed time-slice run, only priority 1, priority 2 and priority 3 programs arrived during this time period.

Intermediate and final details of all the programs that received processing were collected and a graph of penalty values versus time was drawn. Figure 8 illustrates a portion of this graph over the interval from 450 secs. to 1000 secs. As in Figure 6 the priority numbers are marked beside the curves. Those curves with no priority number marked at the end are for programs which did not complete processing at the end of 1000 secs. Some of the curves are labeled with letters so that they can be referred to more conveniently in later discussions.

All the priority 1 and priority 2 programs,

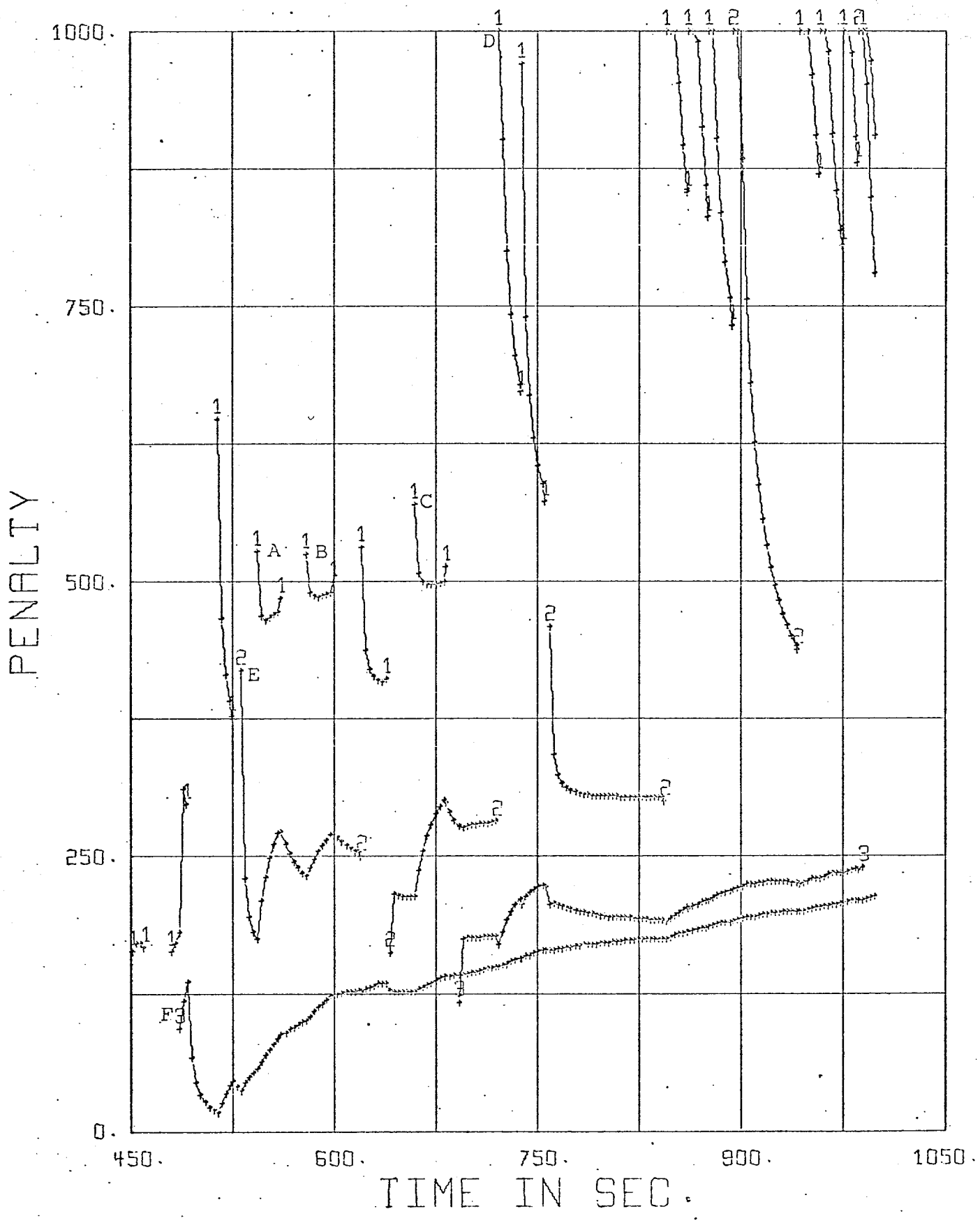


FIGURE 8

PENALTY VS. TIME OF PROGRAMS PROCESSED IN THE INTERVAL 450 - 1000 SECS. USING THE VTS ALGORITHM.

compared to those in Figure 6, have lower final penalties. This observation seems to indicate that priority 1 and priority 2 programs receive better treatment from the VTS algorithm than from the FTS algorithm.

Since the size of the time-slice in this scheme is variable, it is interesting to examine how the size of the time-slice affects the penalty value of a program. In particular, if $p(t_1)$ is the penalty of a program at time t_1 and $p(t_2)$ is the penalty of the same program at time t_2, where t_2 is either the time which is one round-robin cycle later or the time when the program leaves the system whichever occurs first, it is possible to determine how large a time-slice is necessary such that the condition

$$p(t_2) \leq p(t_1) \tag{15}$$

is satisfied.

By definition,

$$p(t_1) = f(u_1)$$

and

$$p(t_2) = f(u_2)$$

where f is the penalty function associated with the priority class of the program and u_1 and u_2 are the lack of attention values at times t_1 and t_2 respectively.

Thus,

$$p(t_2) \leq p(t_1)$$

is equivalent to

$$f(u_2) \leq f(u_1). \tag{16}$$

Since f is a monotonically increasing function over the closed interval [1,15] then (16) implies that

$$u_2 \leq u_1. \tag{17}$$

Let T be the time which has elapsed (since the arrival of the program) at time t_1 and ΔT be the increment in time from t_1 to t_2. Furthermore, let τ be the accumulated processing time received by the program at time t_1 and $\Delta\tau$ be the increment in processing time received during the time interval ΔT.

Then, according to the definition of the lack of attention from equation (1) (see "Penalty Function and Quantum-Allocation Routine", page 25), (17) becomes

$$\frac{T + \Delta T}{\tau + \Delta\tau} \leq \frac{T}{\tau}. \tag{18}$$

By factoring, (18) can be rewritten as

$$\frac{T\left(1 + \frac{\Delta T}{T}\right)}{\tau\left(1 + \frac{\Delta\tau}{\tau}\right)} \leq \frac{T}{\tau}. \tag{19}$$

However, since $T > 0$ and $\tau > 0$, (19) becomes

$$\frac{\left(1 + \frac{\Delta T}{T}\right)}{\left(1 + \frac{\Delta\tau}{\tau}\right)} \leq 1. \tag{20}$$

Also, since $(1 + \frac{\Delta\tau}{\tau}) > 0$, (20) can be written as

$$1 + \frac{\Delta T}{T} \leq 1 + \frac{\Delta\tau}{\tau} .$$

Therefore,

$$\frac{\Delta T}{T} \leq \frac{\Delta\tau}{\tau} .$$

Thus,

$$\Delta\tau \geq \Delta T/u_1 . \qquad (21)$$

From (21) therefore, the increment in central processing time (or the time-slice) $\Delta\tau$ must be greater than or equal to $\Delta T/u_1$ in order that $p(t_2) \leq p(t_1)$. For example, if $u_1 = 4.0$ at time t_1 and $\Delta T = 3.0$ seconds then in order that $p(t_1+\Delta T) \leq p(t_1)$, the quantum $\Delta\tau \geq 3.0/4.0$ second or 0.75 second. In the case where t_2 is the time when a finished program leaves the system during a round-robin cycle, ΔT is the time interval from the start of the cycle to the exit time of the program and may be denoted by T'_{rr}.

In Figure 8, the curves labelled A, B and C show pronounced hooks at their ends. These situations are due to the fact that (21) is not satisfied. For instance, when curve A was examined more closely it was found that in the last round-robin cycle, T'_{rr} was 2.052 seconds and the u value at the start of the cycle was 1.7. From (21), this therefore requires a quantum of at least 1.207 secs. to maintain the final penalty at the same level as the

intermediate penalty from the previous cycle. However, the amount of CPU time which the program required in this cycle before completing its processing was only 0.701 second. Therefore, the increase in penalty when the program completed processing in its final round-robin cycle was unavoidable.

It is observed also that in cases where a penalty increase incurred by a program is unavoidable, the size of this increase can vary depending on the position of the program on the execution list. Since programs are serviced sequentially with the one at the head of the list being serviced first, a program closer to the head of the list leaves the system sooner in its final round-robin cycle than if it were lower on the list. Thus, the unavoidable penalty increase will accordingly be reduced by an earlier departure from the system in the case where the program is higher on the list.

A situation where no penalty increase occurred in the final round-robin cycle of a program is seen in curve D. In this case, u at the start of the final round-robin cycle for the program was 4.3. With T'_{rr} as 0.458 second, the time-slice necessary to keep the penalty constant, $\Delta\tau$, was calculated to be 0.107 second. However, the actual time-slice given to the program was 0.438 second. This amount being greater than $\Delta\tau$, caused

a penalty decrease instead.

The curves in Figure 8 seem to vary according to the program load. For example, when the priority 1 program represented by curve A entered core, curve E which corresponds to a priority 2 program and which was decreasing began to rise. Also, when the priority 1 program represented by curve A left core, curve E started to decrease again until another priority 1 program (curve B) entered core later. In both instances however, the penalty values of the priority 3 program (curve F) which was in core during this interval, seem to rise quite steadily without much fluctuation as is seen in curve E. This behaviour is partly due to the constant slope of the penalty function associated with priority 3 programs. On the other hand the greater degree of fluctuation seen in curve E is partly reflected by the steeper slope of the penalty function associated with priority 2 programs in the region where the penalty value is between 200 and 300. In general, therefore, the curves are partly influenced by the program mix and partly by the height and slope of the associated penalty functions.

In the majority of cases, the priority 1 and priority 2 penalty curves decrease quite rapidly at first but seem to level out with time if the requested CPU times are long enough. The two priority 3 penalty curves seem

to increase quite steadily and their behaviour suggests that they would probably level out also as time increases, if the corresponding programs were given longer execution time.

Generally, the penalty curves in Figure 8 do not seem to converge at any common penalty level or levels such as those observed in Figure 6. Furthermore, since most of the priority 1 and priority 2 programs remain in core for relatively short times, it is difficult to predict the directions towards which their penalty curves are heading in the long run. However, it seems from the levelling out of some penalty curves, that the curves may become asymptotic at values which are specific to the program mixes, if the programs associated with the curves were to have greater requested CPU time.

In view of this, a supplementary experiment was performed to simulate the execution of programs from various program mixes using the time-slice allocation scheme of the VTS algorithm. No supervisor overheads were considered. All programs were given a requested CPU time of 80.0 secs. The intermediate and final penalties were collected and plotted against time.

A set of these graphs which have been obtained is shown in Figure 9 using a program mix which consists of a priority 1, a priority 2 and a priority 3 program

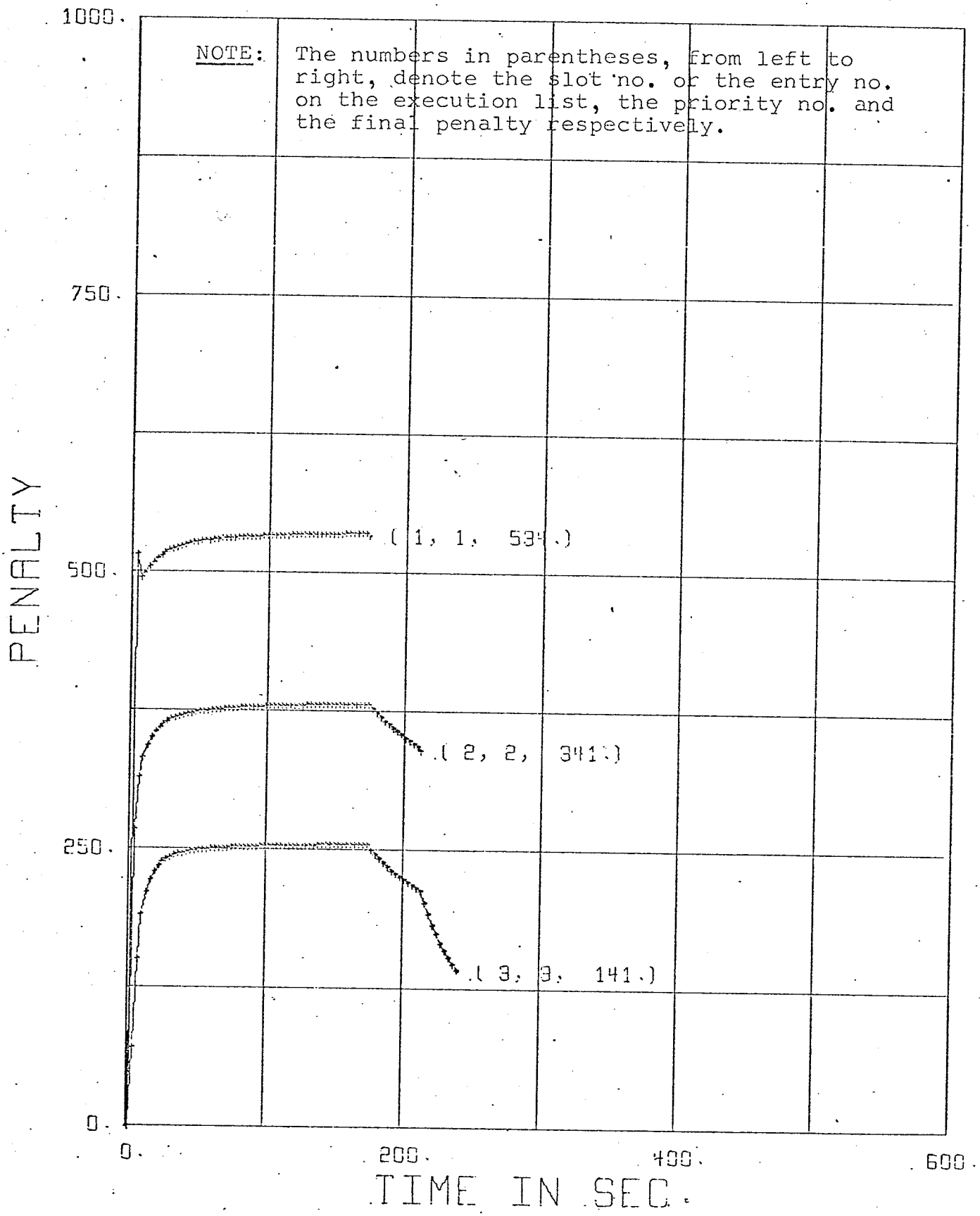


FIGURE 9

PENALTY VS. TIME OF A 1-2-3 MIX USING THE VTS ALGORITHM WITH SUPERVISOR OVERHEADS IGNORED

(that is, a 1-2-3 mix).[7] The numbers in parentheses at the end of each curve, from left to right, denote the slot number or the entry number on the execution list, the priority number and the final penalty.

In addition, when the final penalty of the first program to complete processing was collected, the intermediate penalties of the other programs in core were also recorded at this instant. These results for all possible i-j and i-j-k mixes are shown in Table V. The dashed entries within each row of the table indicate that these entries are not relevant.

From the graphs in Figure 9 and the graphs for the other mixes which have not been shown, it is observed that the curves in general exhibit some degree of fluctuation in the early stages of program execution, and that they all tend to level out into constant penalty levels as time increases until the first program to complete processing departs from core. Thus, the penalty values obtained in Table V can be regarded as the approximate asymptotes of the penalty curves for the programs in the various program mixes. In some mixes which contain two

---

[7]The nomenclature "i-j-k mix" denotes a program mix consisting of a priority i, a priority j and a priority k program. Similarly, the nomenclature "i-j mix" denotes a mix of a priority i and a priority j program.

TABLE V

APPROXIMATE ASYMPTOTIC VALUES OF PROGRAMS IN VARIOUS MIXES USING THE VARIABLE TIME-SLICE ALGORITHM (SUPERVISOR OVERHEADS IGNORED)

No. of Programs In Mix	PRIORITY CLASSES								
	1	1	1	2	2	2	3	3	3
2	515	515	-	-	-	-	-	-	-
2	466	-	-	321	-	-	-	-	-
2	410	-	-	-	-	-	171	-	-
2	-	-	-	265	266	-	-	-	-
2	-	-	-	203	-	-	120	-	-
2	-	-	-	-	-	-	71	71	-
3	613	613	613	-	-	-	-	-	-
3	589	591	-	437	-	-	-	-	-
3	568	569	-	-	-	-	284	-	-
3	-	-	-	376	376	376	-	-	-
3	564	-	-	410	410	-	-	-	-
3	-	-	-	339	340	-	220	-	-
3	-	-	-	-	-	-	141	141	141
3	499	-	-	-	-	-	227	227	-
3	-	-	-	294	-	-	184	184	-
3	534	-	-	382	-	-	255	-	-

programs belonging to the same priority class, the slight difference in penalty value in the table between the two programs is due to the order in which the programs were serviced, with the one serviced first having the smaller penalty. Within a particular program mix, therefore, programs whose penalty curves initially start above or below their asymptotes will eventually have their curves converge at the asymptotes, provided that they requested a sufficient amount of CPU time.

To facilitate the discussion of the curves of various program mixes in Figure 8, the time interval from 450 - 1000 secs. has been partitioned into time segments with most of these segments lettered as shown in Figure 10.

As seen in Figure 10, only two program mixes can be singled out as remaining in core for any length of time and these are the 2-3-3 mix in time segment I and the 2-3-3 mix in time segment L. The values to which these penalty curves are approaching compare quite favourably with the corresponding asymptotic values in Table V, which are 294 for the priority 2 penalty curve and 184 for the priority 3 penalty curve. In time segment P which also involves a 2-3-3 mix, it appears that the priority 2 curve would likely approach the value of 294 also. And in time segment G, the priority 1 and

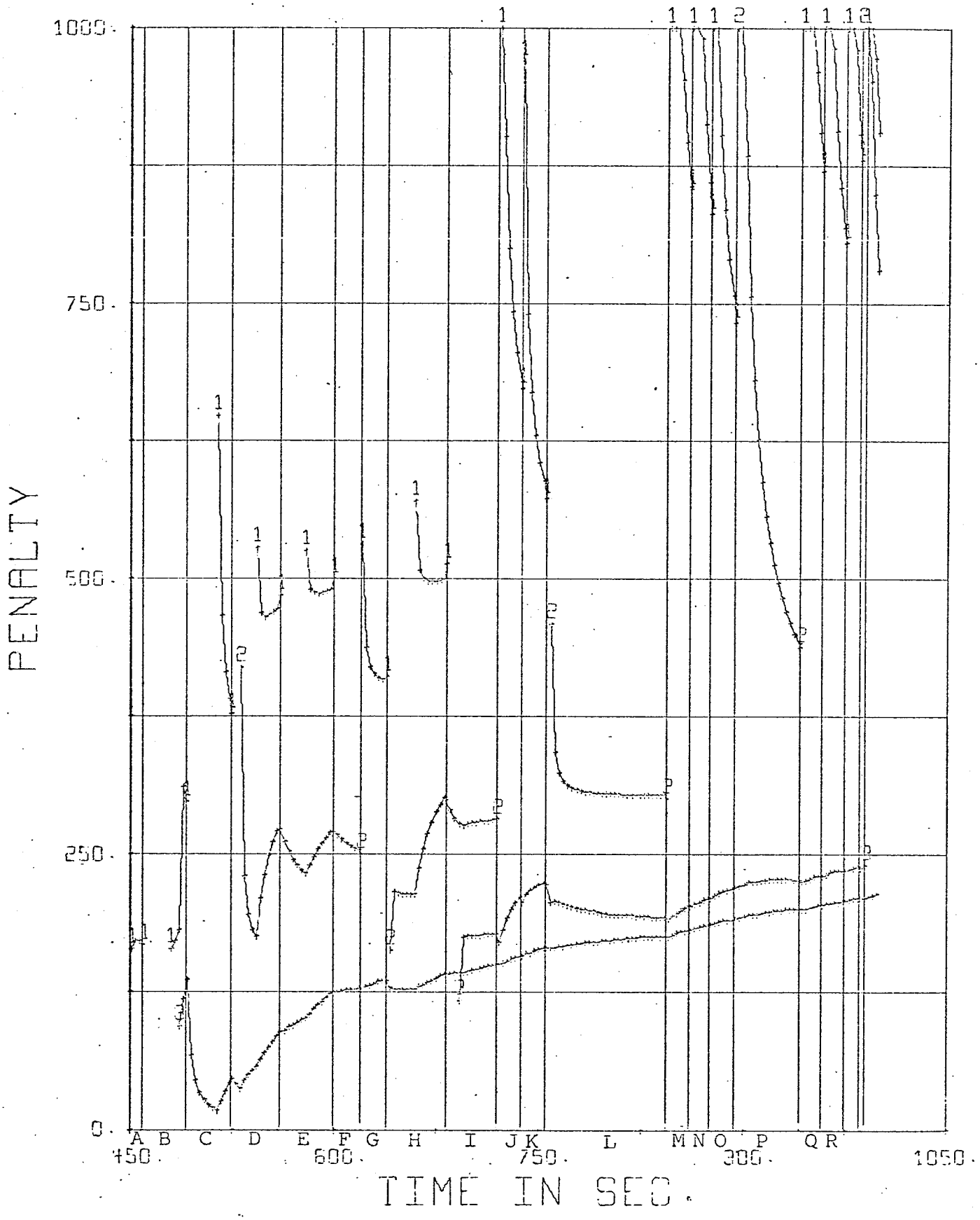


FIGURE 10

TIME INTERVAL 450 – 1000 SECS. IN FIGURE 8
PARTITIONED INTO SEGMENTS

priority 3 penalty curves in the 1-3 mix seem to be approaching the asymptotes of 410 and 171 respectively.

With regard to the other mixes in Figure 10, not much can be said about the penalty levels towards which their curves are approaching, since these mixes did not remain for a long enough period in core for the curves to achieve any form of stability. The increase in the priority 1 penalty curve in time segment B seems reasonable as the asymptotic value for the priority 1 curve in a 1-3 mix is 410.

The results in Table V can also be derived analytically, but such an approach involves fairly complex calculations in most cases. However, an example is considered using a 3-3 mix.

Since, from equation (2) (see section "Penalty Function and Quantum-Allocation Routine", page 27),

$$p_1 = f_3(u_1)$$

where p_1 is the penalty for program 1, f_3 is the penalty function associated with priority class 3 programs and u_1 is the lack of attention to program 1, then

$$p_1 = 999\ ((u_1 - 1)/14) + 1\ . \qquad (22)$$

Similarly,

$$p_2 = 999\ ((u_2 - 1)/14) + 1 \qquad (23)$$

Let $\Delta\tau_1$ and $\Delta\tau_2$ be the increments in processor

time received by program 1 and program 2 respectively. Then

$$\Delta\tau_1 + \Delta\tau_2 = t_{rr} . \tag{24}$$

For steady state,

$$u_i = T_i/\tau_i = t_{rr}/\Delta\tau_i \tag{25}$$

where u_i is the lack of attention, T_i is the elapsed time since program entry and τ_i is the accumulated processor time; and

$$\Delta\tau_1 = \Delta\tau_2 . \tag{26}$$

From (22) and (25) therefore,

$$p_1 = 999\ (t_{rr}/\Delta\tau_1 - 1)/14 + 1. \tag{27}$$

Since from (24) and (26)

$$t_{rr} = 2\Delta\tau_1$$

then (27) becomes

$$p_1 = 999\ (2-1)/14 + 1.$$

Hence,

$$p_1 = 72.4.$$

Similarly, it can be found that

$$p_2 = 72.4.$$

Thus the values obtained in Table V for a 3-3 mix appear quite reasonable compared to the analytic values.

(iii) Comparison of the Variable and Fixed Time-Slice Algorithms

The results obtained earlier on the fixed time-slice and variable time-slice algorithms were used to examine the relative performance of the two algorithms. Only those programs which had completed processing under both algorithms were used in the comparison. The number of completed programs were 48.

The difference in the final penalty values for the execution of program i under the two algorithms is defined as

$$\Delta p_i = p_i^* (FTS) - p_i^* (VTS) \qquad (28)$$

where $p_i^*(FTS)$ and $p_i^*(VTS)$ are the final penalty values for program i under the fixed time-slice and variable time-slice algorithms respectively.

Figure 11 shows the graph of Δp_i versus the exit time of program i under the variable time-slice algorithm, $t_{i,out}(VTS)$. The number beside each point or program indicates the priority number of the program.

As seen in Figure 11, most of the Δp_i's for priority 1 and priority 2 programs are positive. This resulted from the relatively lower penalties incurred by most of the priority 1 and priority 2 programs under the variable time-slice algorithm than under the fixed time-slice algorithm. Thus, higher priority programs seem

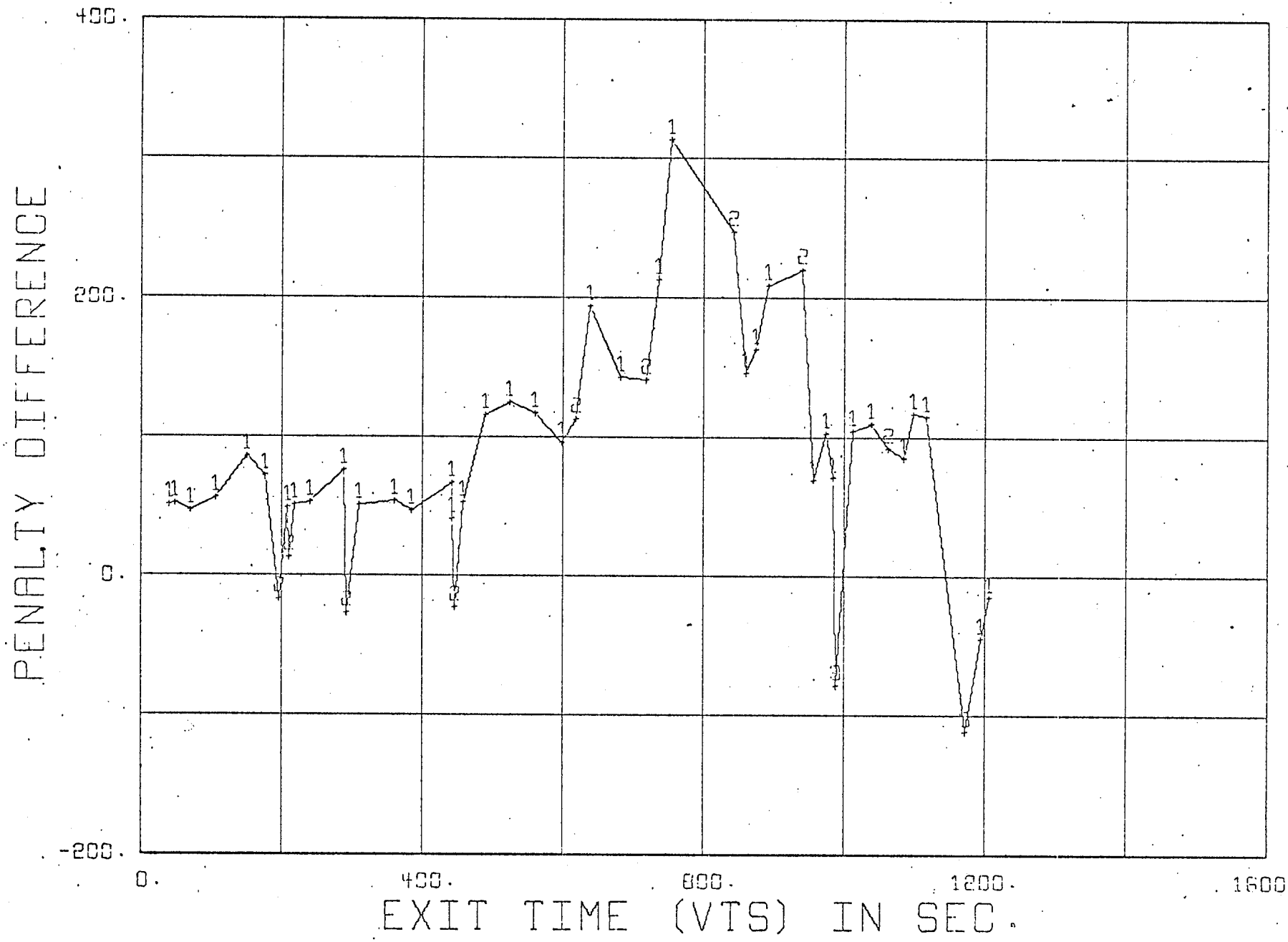


FIGURE 11

PENALTY DIFFERENCE VS. EXIT TIME (VTS)

to be favoured by the VTS algorithm. On the other hand, the negative Δp_i's for the two priority 3 programs in Figure 11 show that the priority 3 programs have relatively lower penalties under the FTS algorithm. Hence, the lower priority programs are given better service under FTS algorithm.

The difference in the exit times for the execution of program i under the two algorithms is defined as

$$\Delta t_i = t_{i,out}(FTS) - t_{i,out}(VTS) \qquad (29)$$

where $t_{i,out}(FTS)$ and $t_{i,out}(VTS)$ are the exit times for program i under fixed time-slice and variable time-slice algorithms respectively.

A graph of Δt_i versus $t_{i,out}(VTS)$ was plotted and is shown in Figure 12. The numbers on the graph represent priority numbers as in Figure 11.

In Figure 12, the Δt_i's for most of the priority 1 and priority 2 programs are positive. This is due to the earlier departure from the system of most of the priority 1 and priority 2 programs under the VTS algorithm than under the FTS algorithm. For the two priority 3 programs, however, the opposite is true, that is, they leave the system later under the VTS algorithm than under the FTS algorithm. These observations, therefore, are consistent with those made from Figure 11.

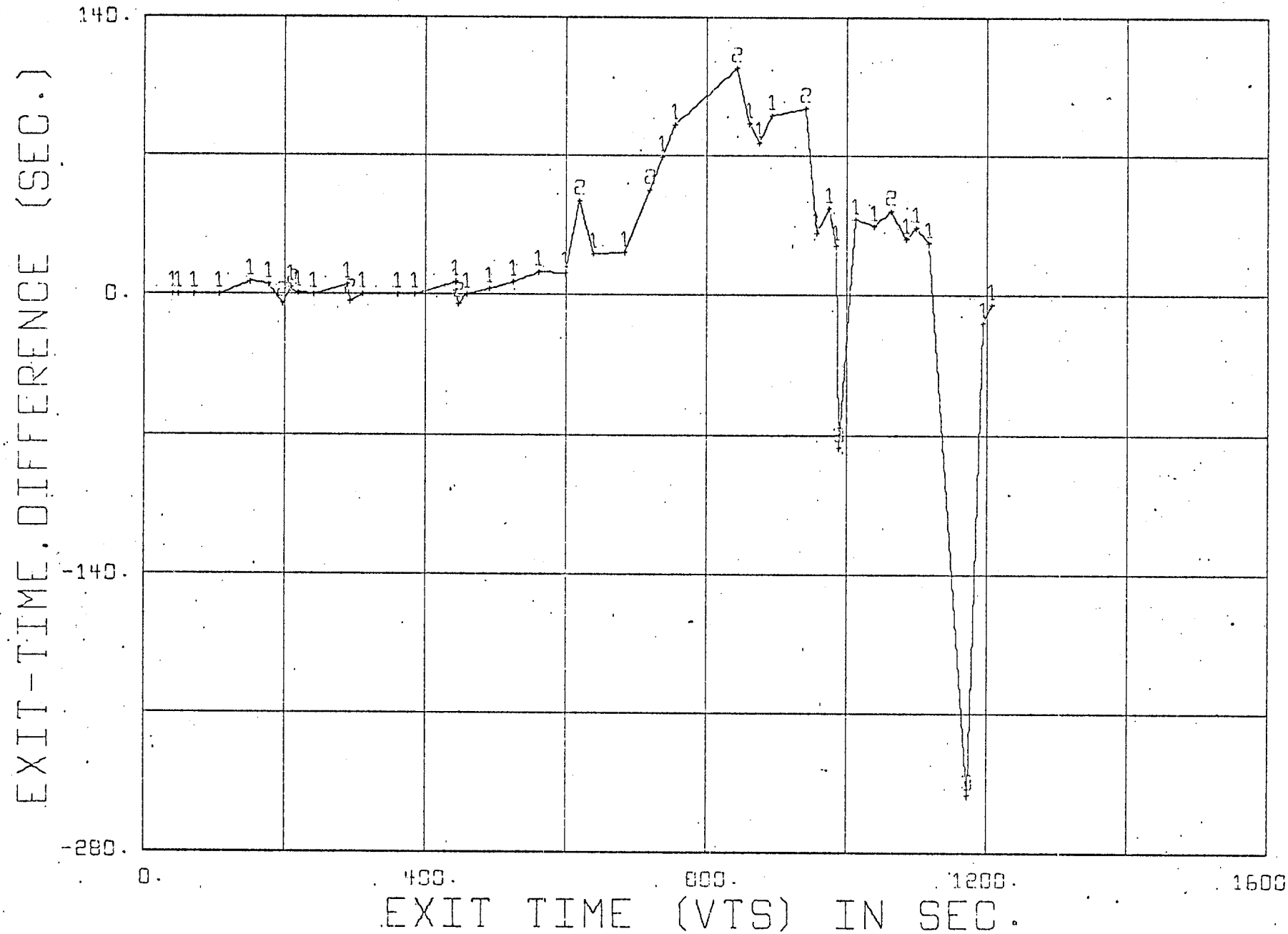


FIGURE 12

EXIT-TIME DIFFERENCE VS. EXIT TIME (VTS)

In summary, both Figure 11 and Figure 12 seem to indicate that higher priority programs are favoured by the VTS algorithm rather than by the FTS algorithm, while lower priority programs received better service under the latter scheme.

To determine the relative merits of both algorithms by a quantitative approach, the measure of efficiency, P_N^*, for each algorithm was calculated, where $N = 48$. The measure of efficiency for the VTS algorithm, $P_{48}^*(VTS)$, was 21869 while the measure of efficiency for the FTS algorithm, $P_{48}^*(FTS)$ was 25893. The relative percentage improvement of the VTS algorithm over the FTS algorithm was found to be 18.4% in this restricted study. This result further supports the earlier observations made and indicates that the VTS algorithm is more efficient than the FTS algorithm for a given job load and job mix.

4.2 Investigation of the Optimum Round-Robin Time

As was mentioned earlier (section "Round-Robin Cycle Time", page 47), the size of t_{rr} ought to influence the performance of the algorithm. Such effects were found by Greenberger for linear penalty functions [8]. It appears that for a given work-load environment, the shorter the round-robin time, the higher the supervisor overheads. However, since lengthening the round-robin

time to minimize the supervisor overheads gives rise to greater neglect of higher priority programs, the round-robin time cannot be chosen arbitrarily large for more efficient performance of the algorithm. Hence, there is apparently some intermediate value of t_{rr} where the combined influence of these two factors is a minimum.

The object of this study was to find the optimum round-robin cycle time for a given job load and job mix environment using the VTS algorithm with P_N^* as the performance measure.

The execution of a given job load and job mix environment was simulated in seven separate runs with t_{rr} assuming a different value in each run. The values of t_{rr} were 0.5 sec., 1.0 sec., 1.5 secs., 2.0 secs., 3.0 secs., 4.0 secs., and 6.0 secs.

Table VI shows the input values of the job-load parameters used for each run.

The mean CPU time and mean interarrival time of all five priority classes of programs with their standard deviations and ranges are shown in Tables VII and VIII respectively.

Slightly more than 500 programs were processed in each run and the P_N^* values were calculated from the 500 programs which were common to each run.

Table IX shows the performance measure of each run

TABLE VI

INPUT VALUES OF JOB-LOAD PARAMETERS FOR INVESTIGATION OF THE OPTIMUM ROUND-ROBIN TIME

JOB-LOAD PARAMETERS	VALUE
Proportionality constant, k	4
Mean CPU time for Priority class 1 programs, m_1	1.0 sec.
Mean Interarrival time for Priority Class 1 programs, r_1	6.0 secs.

TABLE VII

VALUES OF MEAN, STANDARD DEVIATION AND RANGE OF REQUESTED CPU TIME FOR FIVE PRIORITY CLASSES

Priority Class	Mean CPU Time (secs)	Standard Dev. (secs)	Range of CPU Time (secs)
1	1.0	0.25	0 - 2.0
2	4.0	1.00	0 - 8.0
3	16.0	4.00	0 - 32.0
4	64.0	16.00	0 - 128.0
5	256.0	64.00	0 - 512.0

TABLE VIII

VALUES OF MEAN, STANDARD DEVIATION AND RANGE OF INTERARRIVAL TIME FOR FIVE PRIORITY CLASSES

Priority Class	Mean Inter-arrival Time (secs)	Standard Dev. (secs)	Range of Inter-arrival Time (secs)
1	6.0	1.5	0 - 12.0
2	24.0	6.0	0 - 48.0
3	96.0	24.0	0 - 192.0
4	384.0	96.0	0 - 768.0
5	1536.0	384.0	0 - 3072.0

TABLE IX

RESULTS OF 7 SIMULATION RUNS SHOWING t_{rr} AND THE CORRESPONDING P_N^*, WHERE N = 500

Run	t_{rr}(secs)	P_{500}^*
1	0.5	142899
2	1.0	140418
3	1.5	126476
4	2.0	131365
5	3.0	135944
6	4.0	151089
7	6.0	157875

with its associated round-robin time. From the table, there is a minimum value among the values for P_N^* as t_{rr} goes from 0.5 sec. to 6.0 secs. The minimum P_N^* value is 126476, and this corresponds to a t_{rr} value of 1.5 secs. which is the optimum round-robin time (t_{rr}^o) for the given job load and job mix.

One would expect the optimum t_{rr} value to be dependent on the work-load environment. For example, if if the work load consists entirely of high priority programs with short response time, one would expect t_{rr}^o to be small; on the other hand, if the work load consists solely of low priority programs with long response time, one would expect t_{rr}^o to be large.

Greenberger observed that the optimum quantum size varied with parameter values in an analytical study using linear cost curves and a fixed time-slice scheduling algorithm. The parameter values used by Greenberger which are most comparable to the ones used here give an optimum quantum size of approximately 0.6 sec. This value compares well with the optimum round-robin time of 1.5 sec., since on the average the number of programs being processed at any one time is intermediate between 2 and 3.

4.3 Summary

This chapter investigated the performance of the simulation model. A study of the behaviour of the variable time-slice (VTS) scheduling algorithm and of the relative performance of this algorithm compared to the fixed time-slice (FTS) algorithm has been made. Furthermore, an optimum round-robin time for the VTS algorithm with a given job load was determined.

The analysis, both qualitative and quantitative (using the measure of performance P_N^*), showed that the VTS scheduling algorithm when compared to the FTS algorithm gave preferred treatment to higher priority programs at the expense of penalizing but not neglecting lower priority programs. The quantitative improvement in the P_N^* value of the VTS algorithm over the FTS algorithm in a very restricted study was 14.8%.

The investigation of the optimum round-robin time using a given job load has indicated that while more supervisor overheads might be incurred in using a small round-robin time, it does not imply that the round-robin time can be taken arbitrarily large (since in such a case high priority programs with short response time will likely be neglected). The optimum value found (1.5 secs.) is one that minimizes the inefficiency of the

scheduling algorithm for the job-load considered.

4.4 Suggestions for Future Research

The results of the present research have provided some understanding about the basic operation and characteristics of the job-scheduling algorithm using round-robin scheduling with a variable time-slice allocation scheme. The overall study revealed that the algorithm reacted quite favourably in the test environment to which it was subjected. The algorithm, therefore, should be commended for future studies.

Some investigations which may be considered for future research are the following:

(1) To increase the simulation run to include the processing of more priority 3, priority 4 and priority 5 programs so that their effect on the performance of the algorithm can be investigated.

(2) To determine the optimum value for the maximum number of programs which can be scheduled simultaneously by the job-scheduling algorithm for given work-load environment.

(3) To expand the simulation study to include input and output operations.

(4) To study the effects of roll-in and roll-out of programs in the case where a higher priority program in a queue can pre-empt a lower priority program in core for CPU attention.

APPENDIX A

DESCRIPTION OF WAITING-LIST AND EXECUTION-LIST ENTRIES

The items of description which are recorded for an entry in the waiting list are the following :

(1) The job number

(2) The priority class number

(3) The requested CPU time

(4) The arrival time

(5) The current penalty

The items of description which are recorded for an entry in the execution list are the following :

(1) The job number

(2) The priority class number

(3) The requested CPU time

(4) The arrival time

(5) The accumulated CPU time received

(6) The intermediate or final penalty

(7) The allocated time-slice

APPENDIX B

DESCRIPTION OF RANDOM NUMBER GENERATOR

The random number generator used by the job generator routine is the power residue or multiplicative congruential method. The method starts with a constant k, a starting value n_0 and a modulus m. A sequence $\{n_i\}$ of non-negative integers, randomly distributed, with each less than m is generated by means of the recursive formula

$$n_{i+1} = kn_i \pmod{m} \tag{30}$$

where k = 65539 and $m = 2^{31}$.

Any positive odd integer less than 2^{31} may be chosen as the starting value n_0.

To obtain a real number, y, randomly distributed in the interval (0,1) a further calculation is necessary using the formula

$$y = n_i/(2^{31} - 1) \tag{31}$$

where n_i is an integer randomly distributed and is first obtained by means of formula (30).

Further discussion of the multiplicative congruential method may be found in [9].

APPENDIX C

DESCRIPTION OF METHOD FOR GENERATING NORMAL RANDOM VARIATES

The method for generating normally distributed random variates in this study utilizes the Central Limit approach (see [10]).

With a given mean m_x and standard deviation s_x, a normally distributed random variate, x, may be generated using the following formula:

$$x = s_x(12/K)^{1/2} \left(\sum_{i=1}^{K} n_i - K/2\right) + m_x \qquad (32)$$

where n_i is a uniformly distributed random number between 0 and 1 (n_i can be obtained by using the random number generator described in Appendix B) and K is the number of values n_i to be used.

According to the Central Limit Theorem, as K approaches infinity, the set of values of x approaches a true normal distribution asymptotically. However, to reduce execution time, K was chosen as 12. Thus formula (32) becomes :

$$x = s_x \left(\sum_{i=1}^{12} n_i - 6.0\right) + m_x. \qquad (33)$$

For further treatment of this topic, see [10].

APPENDIX D

SAMPLE LISTING OF INPUT JOB STREAM

With the input values shown in Table II for the variable parameters and 157832165 as the starting number for the random number generator, a sample listing of the first 20 programs with their initial job description (namely, program number, priority number, CPU time requested and arrival time) is shown below.

Program No.	Priority No.	CPU Time Requested (secs.)	Arrival Time (secs.)
1	1	10	29.350
2	1	5	42.154
3	1	7	62.406
4	1	11	94.822
5	2	40	120.398
6	1	12	131.210
7	1	9	159.228
8	2	17	171.606
9	1	10	188.162
10	1	7	208.860
11	1	8	232.234
12	2	27	252.612
13	1	12	270.438
14	1	10	299.088
15	1	14	344.906
16	1	10	373.549
17	2	43	382.276
18	1	14	417.436
19	1	5	428.958
20	1	8	450.382

APPENDIX E

SAMPLE OUTPUT LISTING OF FINISHED PROGRAMS

The programs in the sample input job stream shown in Appendix D were processed using the variable time-slice algorithm and the input parameter values shown in Table II.

The final details of these programs are listed on the following page in order of program departure time from the system.

Program No.	Priority No.	CPU Time Requested (secs)	Arrival Time (secs)	Accum. CPU Time Rec'd (secs)	Accum. CPU & Device Time Rec'd (secs)	Final Penalty	Exit Time of Program (secs)
1	1	10.0	29.350	10.0	11.0	.172	40.500
2	1	5.0	42.154	5.0	6.0	.164	48.224
3	1	7.0	62.406	7.0	8.0	.172	70.516
4	1	11.0	94.822	11.0	12.0	.168	106.972
6	1	12.0	131.210	12.0	13.0	.435	150.689
7	1	9.0	159.228	9.0	10.0	.470	176.018
5	2	40.0	120.398	40.0	41.0	.247	196.303
9	1	10.0	188.162	10.0	11.0	.491	208.095
8	2	17.0	171.606	17.0	18.0	.294	211.248
10	1	7.0	208.860	7.0	8.0	.341	218.368
11	1	8.0	232.234	8.0	9.0	.168	241.344
13	1	12.0	270.438	12.0	13.0	.415	288.825
12	2	27.0	252.612	27.0	28.0	.181	293.300
14	1	10.0	299.088	10.0	11.0	.172	310.238
15	1	14.0	344.906	14.0	15.0	.172	360.096
16	1	10.0	373.549	10.0	11.0	.179	384.719
18	1	14.0	417.436	14.0	15.0	.465	442.218
19	1	5.0	428.958	5.0	6.0	.562	443.318
17	2	43.0	382.276	43.0	44.0	.188	447.859
20	1	8.0	450.382	8.0	9.0	.168	459.493

REFERENCES

[1] Neilsen, N.R., "The Simulation of Time-Sharing Systems," Comm. ACM, Vol. 10, No. 7 (July 1967), pp. 397 - 412.

[2] Fife, D.W., "An Optimization Model for Time-Sharing," Proceedings, AFIPS, 1966 Spring Joint Comput. Conf., Vol. 28, pp. 97 - 104.

[3] Scherr, A.L., An Analysis of Time-Shared Computer Systems (MAC-TR-18, MIT Project MAC, Cambridge, Mass., 1965).

[4] Kleinrock, L., "Time-Shared Systems: A Theoretical Treatment," JACM, Vol. 14, No. 2 (Apr. 1967), pp. 241 - 261.

[5] Shemer, J.E., "Some Mathematical Considerations of Time-Sharing Scheduling Algorithms," JACM, Vol. 14, No. 2 (Apr. 1967), pp. 262 - 272.

[6] Fine, G.H., and McIsaac, P.V., "Simulation of a Time-Sharing System," Management Science, Vol. 12, No. 6 (Feb. 1966), pp. B180 - 194.

[7] Corbato, E.J., Merwin-Daggett, M., and Daley, R.C., "An Experimental Time-Sharing System," Proceedings of the Spring Joint Computer Conference (1962), pp. 335 - 344.

[8] Greenberger, M., "The Priority Problem and Computer Time Sharing," Management Science, Vol. 12, No. 11 (July 1966), pp. 888 - 906.

[9] Naylor, T.H., Balintfy, J.L., Burdick, D.S. and Chu, K., Computer Simulation Techniques (New York: John Wiley & Sons, Inc., 1966), pp. 49 - 54.

[10] Ibid., pp. 90 - 95.