Ph.D. Dissertation 7063

THE BOARD OF GRADUATE STUDIES
APPROVED THIS DISSERTATION
FOR THE Ph. D. DEGREE ON 7 MAY 1970

THE ANALYSIS AND SIMULATION OF MULTI-ACCESS COMPUTER SYSTEMS

by

J.M.H. HUNTER

A dissertation submitted to the University of Cambridge for the Degree of Doctor of Philosophy

UNIVERSITY LIBRARY CAMBRIDGE

Trinity College

November 1969

PREFACE

The work described in this dissertation was carried out in the Cambridge University Mathematical Laboratory. I should like to thank the director and staff of the laboratory for their cooperation and for the use of the facilities, particularly the Titan computer.

I should particularly like to thank my supervisors, Professor D.W. Barron for the first two years and Dr. R.M. Needham for the last two years, for the help, encouragement and guidance they both gave me.

During the first three years I was grateful for the financial support of a Research Studentship from the Science Research Council.

No part of the work described in this dissertation has been submitted for a degree at any other university. Except where acknowledgement to other authors is given, the work described herein is original.

CONTENTS

Chapter		Page
	Summary	1
1	Introduction	2
2	The Cambridge System	25
3	Statistical Measurements	31
4	Analytical Models and Results	54
5	Simulation Languages	93
6	FOSSIL - Fortran Systems Simulation Language	104
7	Simulation Models and Results	119
8	Conclusions	151
	Bibliography	161
Appendix	Simulation Program	168

SUMMARY

The structure and purpose of multi-access systems are described and the various types are compared. The Cambridge multi-access system is described in detail with the methods used to gather information about its operation. Various statistics about it are given, some of which are used in the subsequent work.

The relative merits of deterministic and probabilistic analytical models of multi-access systems are discussed and models of both types are developed. The results of these are compared for various parameter values.

Next the types of simulation languages are discussed and details of the structure and use of the one designed and implemented for the work are given. The advantages and difficulties of simulation for investigating computer systems are set out and there is a detailed description of the actual simulation models used. The results of these are compared for various storage allocation schemes.

Finally conclusions are drawn about the use of the various techniques for modelling multi-access systems and the value of the work for systems design and development. References are made to previous analytical and simulation models and simulation languages and the reasons for choosing different models and languages according to the system being examined are discussed.

Chapter 1

INTRODUCTION

In the early days of computing most of the jobs were numerical calculations which sometimes lasted for hours, and the limiting factor was the speed of the central processor. As a result of this it was quite common for a programmer to sit at a computer, finding his mistakes by examining his results and rectifying them by punching another card or examining and changing the contents of the core store using hand keys or a typewriter. The time the central processor was idle waiting for this and for input-output was small compared with the total computing time.

As the speed of central processors increased, much of the potential increase in computing power was wasted by the processor waiting for input-output from the peripherals, the speed of which was limited by their moving parts. Various solutions were tried for this problem.

One of these was to use a small cheap computer to convert a batch of jobs on cards to magnetic tape, this was then transferred to the main computer where the jobs were run in succession without human intervention, the output going to another magnetic tape. Finally the contents of this tape were printed or punched as required on the small computer. This technique was known as "batch processing", the commonest example used an IBM 7090 as the main computer and a 1401 as the small computer. The 1401 was a character-oriented machine and thus particularly suitable for input-output.

To get the full advantages of this system the central processor must be able to work with the minimum of human intervention

and delays waiting for peripherals. Since the input and output was on magnetic tape a "buffer" is necessary in the core store, as a typical program consumes input and produces output in small irregular amounts, whereas a magnetic tape is best for large regular amounts. This reduces the amount of tape used for gaps between blocks of information and the time taken to start and stop the tape and traverse the gaps. With two buffers the contents of one can be transferred to or from the tape while the other is being filled or emptied and then the roles can be reversed.

In order that the processor can proceed at full speed when there is none of this work to do, without having to test regularly whether anything needs doing, a system of interrupts is used. When a tape transfer finishes or a buffer becomes full, the current user's program is interrupted, and control is transferred to a program often called the Supervisor, Executive or Monitor. This deals with the event as necessary and then resumes the interrupted user's program.

All input and output is done by calls to subroutines in the Supervisor, these can perform useful functions like translating from internal character coding to the various peripheral codings, assembling character by character input-output into lines or vice versa using buffers, and organising input and output into several streams so that it appears to be coming from or going to several peripherals simultaneously, whereas in fact each stream comes out in turn on the same peripheral.

In addition to this the Supervisor can perform some of the jobs of an operator on a small computer. At the end of each job control returns to the Supervisor which tidies up and then starts the

next job of the batch if any. Provision is made for the operator to terminate or restart the current job if anything has gone wrong. If there is a hardware clock it can be used to provide accounting information, and to terminate a job automatically if it has exceeded the programmer's time estimate.

While a job is running it often has several "phases", for example editing, compilation, assembly, loading and execution. The supervisor can ask the operator to load the relevant tapes and then read down the programs and data, thus steering the job through the different steps without any operator intervention except where necessary. This saves human and computer time and reduces the chances of error, since checks can be built in and faults are always reproducible. Such a system is known as an "operating system"[(50)].

The most serious disadvantage of this was the increased cost of having two computers instead of one and a large number of tape decks. In larger installations several small computers were necessary to cope with the volume of input and output. It was also difficult to sort out and load the many reels of magnetic tape and identify the output.

Direct access by the programmer while his program was being run was technically feasible but administratively impracticable. Not only would he be wasting valuable central processor time whilst he did his thinking, but also it was impossible for him to change a card and have another run because his cards were no longer there, and if he were allowed to alter the core store of the computer he might have upset the operating system or the other jobs that had already run or were waiting to run.

Another disadvantage imposed by the sequential nature of magnetic tape is that the jobs have to be run in the order that they are presented and hence the turnround time for short jobs is almost as long as for long jobs, since they all have to queue together on a magnetic tape at each stage. This can be improved slightly by devoting certain runs to short jobs only, or having different streams of input and output for short and long jobs, but a consistently low turnround is impossible, particularly because of the time spent handling the jobs. If a job has a high proportion of input or output relative to the amount of processing it requires, the central processor is idle for some of the time waiting for the magnetic tape.

Some of these problems were solved by the introduction of a technique known as "time sharing"[64]. The true use of this term is to describe the situation where two or more independent activities are proceeding simultaneously, such as the execution of a program and output to a printer. In time sharing systems only one computer need be used and the output of past jobs, execution of current job(s) and input for future jobs all takes place simultaneously under the control of the operating system.

When a peripheral has some input ready or is waiting for some output it sends an interrupt to the central processor. This causes the processor to suspend what it is doing, transfer the input or output to or from the buffer and set the peripheral going again. The processor then resumes what it was doing before unless there are any further interrupts waiting, in which case it deals with these in order of priority. Thus as far as possible both the peripherals and the central processor are kept going at full speed as long as there is work for

them to do. If the workload on the machine is sufficiently high and the buffers are large enough, one can approach the condition where all the equipment is working all the time, which gives the intuitive impression that the computer is being used with maximum efficiency.

From the programmer's point of view this is much the same as batch processing, as he still has no direct access to his program when it is running. The computer utilisation is highest when there is always a backlog of work to be done, which means a long turnround time. There are also hidden overheads associated with time sharing, the processor must fill andempty buffers, deal with peripherals and perform the functions of the operating system.

Since all the input or output cannot in general be kept in the core store, which would soon fill up, it must be transferred to backing storage, possibly magnetic tape but preferably magnetic cards, disc or drum, since fast random access is desirable. This opens up the possibility of programs being backing store channel limited instead of peripheral limited. It may appear paradoxical that if we remove one bottleneck we should immediately introduce another, but in fact this is inevitable in a fully loaded system, since unless all the units of the system have adequate speed or capacity the slower ones constitute the bottleneck. However it is possible to compensate partially for this by favouring jobs that use spare capacity in preference to those that cause overloads.

Systems have been proposed to overcome this problem using a hierarchy of different backing stores with different access times, capacities and costs, but this results in a large number of transfers. The solution usually adopted is "multiprogramming" which means having

several active programs in the core store at once, or "swapping" which means having several programs active at once, but only one in the core store at a time and the remainder on backing storage. Both of these can improve the turnround time for short jobs considerably by interrupting long jobs while they are running, but only multi-programming can keep the processor fully occupied while a program is waiting for backing store transfers, since with luck there is always one program in core free to go. It is possible to combine multi-programming and swapping, or have programs partly in the core store and partly on backing storage. This scheme, known as "paging", will be described later.

We have now described a system whereby peripherals of any speed can be catered for and several jobs can be run simultaneously, and this leads naturally to the idea of having a peripheral controlled by a human "on-line" to the computer. This would give the programmer back the facilities he used to have of being able to think about and modify his program, interact with it and get the results back quickly, and many new facilities which will be described later.

In a multi-access system several people have this facility simultaneously. In principle it is as easy to provide it for several people as for one, however in practice the main problem is allocating the limited resources of a computer system, such as processor time, core store, disc space, peripheral output, backing store channel time, etc., among a large number of users with conflicting demands to try and satisfy as many as possible as much as possible.

There are several types of multi-access system. First we have what are misleadingly known as "real time" systems. This description

refers to systems where it is important that there should be a response within a certain specified time, or alternatively that the probability of there not being a response should be less than a certain value. Examples of these are process control, telephone call switching and seat reservation systems, the latter being included because the customer does not want to wait a long time, rather than because of any intrinsic urgency.

Multi-access is not popular among such systems, except for the latter where the common data base makes it necessary. The main reason for this is that there is no further drop in response time as a result of multi-access since this has to be short anyway, and the extra complication and overhead needed to deal with several simultaneous real time processes reduce the reliability and the gains in efficiency.

However it will probably be used more in the future when computer networks are developed, since this will be a good means of ensuring the reliability that these systems require at reasonable cost, and computers can conveniently and economically deal with real time and ordinary work simultaneously using interrupts and multi-programming as described earlier. Such systems will not be considered explicitly though some of the conclusions drawn apply to them also.

Most real time systems are special purpose and limited to a certain set of programs, often only one. Some other multi-access systems have adopted this idea, which certainly leads to simplicity and fast response, since there need only be one copy of the program permanently resident in core with a separate set of data for each user if necessary, and thus channel time and core allocation present relatively little trouble. The languages used try to provide for

simple numerical calculations in a format that can easily be learnt by non-programmers.

However there is a limit to what can be done in these languages, and some programs may either be written in an existing language, or require a new language, or have to be written in machine code to make full use of the available facilities or run efficiently. It is because of the existence of such problems that more effort has gone into designing general purpose compatible systems, which enable any program that can be run off-line to be run on-line within reasonable limits. These are the type of systems that will be considered, since the problems involved in the design and running of such systems are much more complicated and interesting. The languages used in special purpose systems can easily be implemented in a general purpose system.

Most systems allow "conversational" input and output at the terminal while the program is active. However this is an expensive facility to provide since either the core store has to be reserved for as long as the user takes to decide what his next input will be, or it has to be "swapped" by writing it to backing storage and reading it down again when the input or output is finished and the program is free to continue running.

One of the most difficult problems about multi-access systems is resource allocation. There will always be an unsatisfied demand for computers and computing power because of their flexibility. As the speed and reliability of computers increase and improvements in softeware and techniques make them easier to use, problems which before would have required too much computer time are now just soluble. The

progress of science and operational research and allied techniques bring to light problems whose existence had not been realised, which had always been treated as areas of human judgment not susceptible to analysis. This unfulfilled demand was the reason for the introduction of time-sharing, and subsequently multi-access systems, for if computer speed and power could not be raised overnight, the best use must be made of the equipment available. One of the most difficult problems however is deciding what is the best use, owing to the wide variety of requirements.

Some of the resources for which competition occurs can be allocated explicitly, such as magnetic tapes and magnetic card, drum or disc space for storing programs and data, and maximum amounts of core store, peripheral output, processor time, console time, and total running time. These can be decided by the management of the computing service and can be enforced by software, the operating staff, the honesty of the users, or finance. If a financial system is not used these resources are allocated according to political criteria, for example the relative claims or riches of two departments may have to be judged, or it may be important not to set a precedent by allocating unequally among users. Optimisation is usually done by a trial and error method, if nobody complains too much then there is pressure to retain the status quo, since somebody usually suffers from changes and these are only made when necessary. At the moment there is no objective basis for deciding the effectiveness of various policies.

A financial control system is perhaps the best attempt at this since it enables users to state the relative importance of individual factors to them and makes them realise what is expensive for the

system. The prices can be set according to the laws of supply and demand. In theory users will trade off some potential advantages against others so that priority can be given where necessary without everybody demanding it all the time with consequent loss of effect. In practice this may mean a considerable effort on the users' part to play the game and produce programs fitting the administration's stereotype, which may not be the most efficient for their particular problem in the long run.

When the users are within an organization the money is often imaginary, which people are much less reluctant to spend, and one department may be careful to use up their allowance to ensure that it is not reduced at the next allocation. If the money is allocated centrally it may be difficult to decide on the relative merits of competing claims, particularly as the amount needed may be very unpredictable. People may ask for more than they need, knowing that they will be given less than they asked for. Over-allocation will defeat the object by causing inflation, as the price of each resource must include scarcity value as well as capital cost in order to ensure an efficient use of the system.

One must balance the result of any such controls against the cost of human or computer time involved in implementing them. Different controls may be applicable in different environments, for example a seriously overloaded machine must be controlled much more rigidly than a relatively lightly loaded machine, where the important factor may be giving a good service. It may be preferable to give a good service to a few than a bad service to many, though the line is hard to draw. The effect of changes in the nature of the load should be taken into account, though it is the difficulty of estimating this

that partially accounts for the reluctance to change.

Many of these remarks also apply to the resources which are allocated implicitly by the structure and the algorithms of the system. The most important of these are the turnround time for off-line jobs and response time for on-line jobs. Both of these will depend on the nature of the jobs, such as what systems programs they use, how much core space and processor time and disc accesses they need, the load at the time, whether any backlogs have built up, and various other factors.

It is difficult to say what aspects of system performance should be optimized since this involves the weight that is given to various users' preferences and the importance attached to the efficient running of the system. Originally the main aim was to optimise the use of the central processor, since this was the main bottleneck. When buying a computer system there might be a difficult choice between having a slower processor and more core store, thus enabling the processor to be kept busy more of the time by multi-programming and providing a faster response time, or a faster processor which would be idle more because there was not enough core store to keep it busy. The decision between these two depends on the relative price of processors and core store, but it is often difficult to judge which would be better.

The decision to keep the processor busy through multiprogramming will weigh against jobs using a lot of core store, since these will either not be allowed or curtailed because they interfere with the multiprogramming, or will cause excessive overhead by requiring frequent swapping and thus keeping the disc or drum busy. It is also

possible that users will recompute sets of numbers or shift them about the core and backing store devices to avoid using too much store and being discriminated against, which will defeat the object of saving processor time. The more sophisticated schemes such as paging introduce considerable overhead which may not be justified.

Perhaps we should reconsider why processor time is important. If we are selling processor time by the second with no charge or conditions for the service and programmer's time is irrelevant then this is a reasonable criterion. Such a situation rarely occurs in practice. The objective is to maximize the satisfaction of existing and potential users, where relevant this is indicated by how much they are prepared to pay for the service, though this is often distorted by competition and lack of knowledge of the consequences of various alternatives. When it is free one should weigh up factors like the relative importance of off-line and on-line jobs, the cost of a programmer's time relative to machine time, how much machine time a certain degree of reliability is worth, how much tape jobs should be favoured or otherwise, how should big jobs be scheduled, and so on. Analysis and simulation can prove valuable to show the consequences of various alternatives in each respect to help make a choice between them.

As an example we can consider the response time for on-line jobs. Some people have taken a very simple factor to be optimised, the average waiting time for a response. While mathematically simple, and thus used at times in this work, this does not take account of the psychology of the average user. The important factors to him are the expected response time to a given request and its variation. It is

reasonable that jobs requiring more processor time should take longer but what the relationship between the two should be is open to debate.

If the user knows that he is going to have to wait less than a minute he will be patient, and if it will be more than five minutes he can get on with something else. In the latter case he should be able to use his console and be notified in a non-destructive manner when his first job has finished. Obviously he will prefer a rapid response, but what he wants more is to know how long it will take. If the time is variable or lies in the middle range he has a difficult decision to make.

One can insist on the user estimating the time required with a default estimate for short jobs. There is an incentive to estimate accurately since if the estimate is too low the job will be terminated by the system after it has exceeded it and if the estimate is too high the scheduling algorithm will result in a slow response.

This estimate can then be used by the scheduler to estimate a completion time and print it out, if it is more than say five minutes. Allowance has to be made for the possibility of other people initiating short or high priority jobs in the meantime and it may be better to give a range of completion times within which the actual time has say a 95% chance of falling.

Another difficult problem is how to arrange the active programs in the core store. The simplest allocation schemes have each active program in a fixed place in core store all the time that it is active. Some multiprogramming systems even insist that the store be divided in a fixed way, for example in a 64K machine there might be a limit of three active programs each limited to 16k and 16K for the supervisor.

Such a rigid arrangement is obviously very wasteful of store and restricts severely the maximum program size that can be run.

However if the programs go in variable positions this makes the core storage protection hardware, which prevents one program affecting another or the supervisor, much more complicated, and when one program finishes an awkward gap may be left since the waiting programs will probably be a different size. Some of the ways of choosing the next program and where to put it in the core store are discussed in Chapter 7.

This problem can be alleviated by moving sections of program around to close up the gaps, this will be called "shuffling". This is a satisfactory procedure in machines with efficient hardware instructions for doing this, but otherwise is liable to waste central processor time, both in deciding when and how to do it, which is not at all obvious, and to a greater extent in actually doing it.

The next possibility is swapping sections of program or whole programs onto backing store, usually magnetic drums. The pioneer multi-access system, Project MAC CTSS (Compatible Time-Sharing System) at MIT[12,53] had swapping without multiprogramming, and most of the analysis on multi-access systems has concentrated on systems of this type.[9,16,17,54,58,59]

Here we have to decide which programs to swap in and out, and have not only to consider the system overhead but also the use of the backing store channels, which can get seriously overloaded. If fully conversational working is permitted some form of swapping becomes essential.

The newest systems are paged, which means that although a user numbers his core store from location 0 upwards as a contiguous block it is stored in pages of fixed size, which may be scattered around the core store, the correspondence being made by hardware. The size of pages in different computers varies from about 100 to 2000words. Some of the pages may not be in core, and if the program attempts to refer to one of these they must be brought down from backing storage. Similarly if a page has not been used for some time, or the core space is required for some other program, the page can be written up to backing storage. The advocates of paging claim that it provides the user with an apparently large core store and permits conversational working at minimal expense to the system.

This system will only work satisfactorily if a program's references to core store follow a simple pattern and remain within one or two pages at a time. If a program genuinely uses the store for random access it will use up an inordinate amount of backing store channel time. When such things are concealed from the user there is a danger that he may use them inefficiently without realising it.

Paging in principle reduces the number of swaps and abolishes shuffling altogether, though it requires expensive associative registers to give quick access to where the pages of a program are in the store, and software to keep these tables up to date and deal with exceptional conditions and decide when and how to reject unwanted pages and get new ones. This depends on user address sequences as well as other factors.

A paged computer, Atlas I, has been in use for some time without multi-access but multi-access paged systems have only recently been developed. Extravagant claims are made for such systems, such as that they can serve 200 simultaneous users, but much of the central processor time is spent on administrative overhead or waiting for the backing store channels to provide pages. Such systems may prove satisfactory in the long run when they are better understood, but new developments in fast access mass stores may make this technique obsolete.

Attempts have been made to simulate such systems,[46,47] which certainly defy mathematical analysis in toto, but the cost and effort required is so great that trial and error is usually used in designing them. The difficulty lies in a lack of experimental data for such systems, both on the operation of the hardware and the software which is only just becoming available, and also on the behaviour of the users and the types of program that they write, which are intimately linked to the characteristics of the system. So many assumptions have to be made that it is difficult to place much confidence in the model. Because of this and the magnitude of the task such systems have not been considered here.

Having considered the various types of store organisation, we shall now consider some of the possible scheduling algorithms for allocating the processor time between active programs when multiprogramming or swapping is used.

The fundamental multiprogramming scheduling algorithm is the "Round Robin" (RR). In this system the jobs are in a circular queue and the processor serves each of them in turn for up to one

UNIVERSITY LIBRARY CAMBRIDGE

"quantum" of time, generally about a second. At the expiry of the quantum (generally detected by an internal clock interrupt) or when the job halts for input or output or finishes, the processor starts to serve the next job, and if the job still remains it is effectively at the end of the queue. This scheme is designed to reduce the waiting time of short jobs.

In the limit as the quantum tends to zero the processor serves all the users simultaneously, this is known as "Processor Shared" (PS) and is useful in theoretical analysis while giving a reasonable approximation to the case when the quantum is small. In the limit as the quantum tends to infinity each job is served to completion, this is known as "First Come First Served" (FCFS) and corresponds to a simple queue, however the short jobs then have to wait almost as long as the long jobs.

Various alternative schemes based on the round robin have been suggested. In the "Foreground Background" (FB) scheme a program joins the foreground queue and is given one quantum of processor time when its turn comes, if it has not finished by then it joins the background queue which is served to completion on a FCFS basis, but only when there are no jobs in the foreground queue. Usually this is "pre-emptive", if a new program arrives while the processor is serving the background queue it interrupts the background program and serves the new one for a quantum.

This ensures a good service for programs lasting less than one quantum while reducing overhead owing to program changes, which is particularly important when there is swapping. However if a program lasts more than one quantum its response is bad, and the

main advantage of this scheme over the schemes described below is the relative ease with which it can be analysed.

This scheme can be extended to several levels. Each time a program finishes a quantum at one level it joins the end of the queue at the next higher level, and a level is only served if there are no programs waiting on any lower level. This is generally done on a pre-emptive basis. A scheme of this nature was used in CTSS with quanta which doubled each time the level increased, larger programs which were expected to last longer entered at higher levels to reduce the amount of swapping and safeguards were included to ensure that no program reached the top level and never got run because there were always other programs waiting at lower levels.

This proved very successful since it gave a short response to short programs and yet reflected the fact that there is no point in giving a short quantum to a program that has already run for say 30 seconds when all other programs waiting have run as long or longer, since this only adds to overhead. In a system without swapping the size of the quanta makes little difference.

There are three techniques for comparing various possible hardware and software systems. The first is to do experiments with the actual systems. It is usually impossible to experiment with the hardware except for manufacturers, since the maximum configuration is fixed by the equipment available. Alsothe commitments to the users of a system make it impossible to experiment with the software of a system unless it is likely to show a significant improvement. Only a limited amount of information can be gathered from this, since the experiment cannot be controlled or repeated. However, useful

main advantage of this scheme over the schemes described below is the relative ease with which it can be analysed.

This scheme can be extended to several levels. Each time a program finishes a quantum at one level it joins the end of the queue at the next higher level, and a level is only served if there are no programs waiting on any lower level. This is generally done on a pre-emptive basis. A scheme of this nature was used in CTSS with quanta which doubled each time the level increased, larger programs which were expected to last longer entered at higher levels to reduce the amount of swapping and safeguards were included to ensure that no program reached the top level and never got run because there were always other programs waiting at lower levels.

This proved very successful since it gave a short response to short programs and yet reflected the fact that there is no point in giving a short quantum to a program that has already run for say 30 seconds when all other programs waiting have run as long or longer, since this only adds to overhead. In a system without swapping the size of the quanta makes little difference.

There are three techniques for comparing various possible hardware and software systems. The first is to do experiments with the actual systems. It is usually impossible to experiment with the hardware except for manufacturers, since the maximum configuration is fixed by the equipment available. Also the commitments to the users of a system make it impossible to experiment with the software of a system unless it is likely to show a significant improvement. Only a limited amount of information can be gathered from this, since the experiment cannot be controlled or repeated. However, useful

statistics can be gathered of user and system behaviour and the nature of the load by observing a real system, as is shown in Chapter 3.

One possible solution to this problem is suggested by Greenbaum[21] who used a computer to simulate several users typing away at their consoles. Scripts were used to enable any user session to be simulated and allowance was made for user thinking time. This makes controlled experiments possible, but plenty of computer time would be necessary to allow for a sufficiently representative user session, and it would be difficult to decide on criteria for judging how representative a session is. Also the experiment would have to be repeated for each proposed modification to the system, as with a simulation.

The next method is mathematical analysis. Some results can be obtained from deterministic models by considering rates of flow through the system and assuming everything to be constant. However it is more accurate to use the theory of queues and Markov processes, which involve representing the various times and queues involved by a probability distribution. The usual distribution adopted for the known times is the exponential distribution. This has the valuable property that the behaviour of the system is independent of its past history and depends only on the present conditions. Without this property the analysis becomes very much more complicated. In practice this represents a reasonable approximation to the observed distributions, even of thinking time, though more accurate fits with derived distributions are possible.[19,62]

Apart from this approximation a great deal of simplification of the system is necessary and thus one inevitably wonders whether the resulting model is a good one. The advantages of this method is that the amount of computer time to produce results is very small so that a wide range of cases may be considered, and it is sometimes possible to solve the equations analytically and see the explicit dependence of one parameter on another. One also knows that the result is an exact solution of the model, even if this does not give an exact representation of the system.

The other is simulation of a model of the system, where the situation is very different. Here we can make the model as complicated as necessary, subject to the serious limitations of one's ingenuity and computer time, and get a numerical solution. The data for simulations can be either real jobs or random numbers sampled from histograms or analytical distributions, preferably determined by statistical results from a real system. Statistics on queue lengths and waiting times can be collected during the simulation of a given period of real time and tabulated at the end. More detailed analysis can trace any apparent instabilities in the system.

The main advantage of simulation is that it can in principle be applied to any type of system and any desired statistics can be collected. Most multi-access systems involve several types of delay, waiting for input and output, waiting for backing store transfers, waiting for core store to become available, and waiting while other programs are being run. A mathematical model cannot deal with all of these simultaneously. It involves simplifying assumptions that

may change the picture completely, and only a limited number of results can be deduced. The results of simulation can give a good qualitative and quantitative picture of the system and indicate where the main bottlenecks occur.

Like most numerical methods, simulation suffers from the disadvantage that it is expensive in computer time. It is difficult to achieve accuracy without having several runs simulating long real time periods, and changing any one parameter will usually involve a complete set of reruns. Though techniques exist for estimating the error[60] much depends on the particular features of the runs used.

In cases as complicated as multi-access systems the accuracy of the model and its implicit assumptions is always in doubt and can only really be verified by comparison with the real system. Even this will only show that the assumptions are satisfactory for one set of parameter values, and when these are changed new factors might come into play which were legitimately ignored before. The accuracy of models of non-existent systems cannot be checked except against oversimplified mathematical models of the same system, and the same incorrect assumptions might be present in both, though this does provide a good technique for detecting programming errors.

Simulation cannot show clearly the dependence or lack of dependence of a result on a given parameter in the same way that an equation can. Unless fairly consistent results are obtained in different runs it is difficult to know whether to ascribe variations as directly due to a parameter or to other ignored factors, or due to the inherent randomness of the process. If the results are independent of one parameter for given values of the other parameters

this parameter might become critical with other values, and yet it would consume too much time to test all possible cases with sufficient accuracy.

In spite of these disadvantages simulation is a worthwhile technique that can produce useful results unattainable by other methods, and in comparable cases the results agree reasonably with those obtained from the real system and mathematical analysis.[54]

Most of the work described here concerns the Cambridge multi-access system, described in Chapter 2. This was partly because the author had first hand experience of using it and more convenient access to information and statistics about it, and partly because it represents the most successful example of a general purpose non-paged multi-access system in actual operation and has been operational sufficiently long for it and its users to have reached a reasonable degree of stability. Hypothetical modifications to the software and hardware were also considered as a basis for comparison.

The work was in four main sections which were in practice done concurrently. First statistics were gathered about the system giving disc access times, user response times, program sizes and running times, use of various systems programs, etc. Next analytical models of the system and similar systems using the theory of Markov processes and queues were designed, and various results tabulated. Next a simulation language, FOSSIL, was designed and implemented, which represented an important piece of work in its own right. This was used to simulate the system to investigate core store allocation.

At the end of this thesis the value of investigating multi-access systems is discussed and conclusions are drawn as to how this problem should be tackled in the light of this work and the uses of the various techniques described here.

Chapter 2

THE CAMBRIDGE SYSTEM

As we are confining our analysis to systems of a similar type of the Cambridge multi-access system and are using statistics derived from this system, a short description of it will now be given. Further details can be found in Ref.(25). This is designed to explain its most important hardware and software aspects and how it appears to the user.

The Cambridge system is based on the TITAN (ATLAS 2) computer, which differed from the ATLAS 1 by not being paged and not having a drum. There is a program called the Supervisor[26] which deals with the administration of jobs, controls all the peripherals, organises the buffering of input-output in the core store and on the disc, records logging and accounting information, deals with all interrupts, provides two way communication with the operators and does other routine jobs. This represents about 32K words of program and tables of which about half is permanently resident in the core store and the other half read in from disc when needed.

In addition there are certain systems programs, such as the filing system and the console logging in and out system, which are written as ordinary programs but have certain privileges. The filing system is used for the storage of information on the discs.[2] A file can contain either program or data copied from the core store or a "stream" of characters, as from a paper tape reader or to a printer.

Programs can take their input from and send their output to peripherals or files or tapes according to choice, except that on-line

Chapter 2

THE CAMBRIDGE SYSTEM

As we are confining our analysis to systems of a similar type of the Cambridge multi-access system and are using statistics derived from this system, a short description of it will now be given. Further details can be found in Ref.(25). This is designed to explain its most important hardware and software aspects and how it appears to the user.

The Cambridge system is based on the TITAN (ATLAS 2) computer, which differed from the ATLAS 1 by not being paged and not having a drum. There is a program called the Supervisor[26] which deals with the administration of jobs, controls all the peripherals, organises the buffering of input-output in the core store and on the disc, records logging and accounting information, deals with all interrupts, provides two way communication with the operators and does other routine jobs. This represents about 32K words of program and tables of which about half is permanently resident in the core store and the other half read in from disc when needed.

In addition there are certain systems programs, such as the filing system and the console logging in and out system, which are written as ordinary programs but have certain privileges. The filing system is used for the storage of information on the discs.[2] A file can contain either program or data copied from the core store or a "stream" of characters, as from a paper tape reader or to a printer.

Programs can take their input from and send their output to peripherals or files or tapes according to choice, except that on-line

jobs cannot use paper tape readers or magnetic tapes for operational convenience. The same system is used for buffering input-output streams and storing files on the discs. The user communicates with the supervisor, the peripherals and the filing system by means of special orders, known as extracodes. In this way he can perform operations requiring privileges in a controlled fashion.

The core store was originally 64K words of 48 bits, and the largest size program allowed was 40K, the remainder being used for the supervisor, buffering, and temporary working space. Later the core store was increased to 128K but no corresponding increase was made in the largest program size allowed, this was to allow more multiprogramming, particularly of on-line jobs, and reduce the rate at which "chapters" of non-resident supervisor had to be overwritten and retrieved from the disc owing to lack of working space in core.

The disc store consists of 32 discs in 2 units, each disc has a capacity of 512 blocks of 512 words making a total of 8 million words. In addition to these, which are read by heads fixed to arms moved with compressed air, there are 96 blocks which can be read by fixed heads and thus have a shorter access time. These are used for parts of the supervisor. There are two logic units which can each move the arms on either disc unit, and two channels for transferring information, and so we can approximate by treating the discs as being in two completely separate units as far as calculation of the access times and queues are concerned. Although the channels have to be shared with the six tape decks the degradation due to these is small.

A typical job passes through a number of phases such as

editing, compilation, assembly, loading and execution. Facilities are provided using extracodes or systems programs for joining these phases together. Information is usually passed between phases in the form of streams or files which are output by one phase and used as input for the next. At the end of each phase all the core store is freed and the job joins the queue again for more space for the next phase, except in the case of the IAL loader, the IIT assembler, and the system MONITOR which prints out information when a fault occurs in a program. For each of these part of the same space is used in two phases.

For on-line jobs various options are open. It is possible to create another job which will run in exactly the same was as an off-line job using an identical job description format (except for paper tape input not being allowed) and operate independently of the console. This would be the normal mode of operation for jobs lasting over about a minute, or using magnetic tape, or liable to long delays because they need a lot of store. However a more natural way of using the system, particularly during program development, is to run one phase at a time, examining the results of each phase at the console before going on to the next phase if it was successful.

All on-line jobs operate under the COMMAND program, which can also be used by off-line jobs but is primarily designed for interactive use. A request for a phase, which may use a system program or a private precompiled binary program, is typed in to the command program as the system or file name, followed by arguments which are usually stream numbers or file titles. The command program sets up the streams and files appropriately, and

then loads the appropriate system by "changing phase", which involves finding the core space or joining a queue for it and reading the system down from the disc.

When the phase has finished running it will generally return to the command program, again by changing phase, which will type out the output at the console. This may be interrupted and the output printed, punched, filed or edited. Thus the natural unit of interaction is a phase subject to certain exceptions.

Some of the more common commands, such as INPUT, FILE, DELETE, PRINT, etc. are built into the command program for efficiency reasons and do not involve changing phase.

It is possible to string several phases together without having interaction, by the same means that are used for off-line jobs. Each phase can change phase into the next, and sometimes several phases are present behind the scenes (e.g. compile, assemble, run). Another steering program called MLS (Mixed Language System) can be used,[6] and there is a special COMMAND command which will make the command program read a list of commands from a stream and only return at the end or if an error occurs.

As well as the command program some other commands are allowed to be interactive themselves, such as EDIT (context editor), PATCH (debugging aid), FIGARO (for small numerical calculations), EXAMINE (for looking at file dictionaries) and FINISH (for logging in and out). This is usually done by having only one copy of the program in core to serve all the users, this is known as a "pure procedure" because it is not allowed to modify itself and has a

small amount (typically 64 words) of working space for each user. This reduces the amount of idle store during input-output while the users think and saves reading down the program from the disc each time, since often it is in the store already. Because of this, and the fact that a special area (16K) of the store which is rarely full is reserved for pure procedures only, the response time for pure procedures is shorter and less variable.

All other programs must have all the input typed in before they start and no output comes out till they have finished. The input and output is stored on the disc and thus the core store is not tied down waiting for the console. This may appear to nullify some of the advantages of on-line access but has in practice proved very satisfactory for most users. The facility for storing, printing and editing output can save a lot of time waiting for something coming at the end, for example a system MONITOR owing to an output loop, which would come at the end of the output causing the trouble.

The core allocation and scheduling routines are described implicitly in Chapter 7, but a brief summary from the system point of view will be given here. The core store available to object programs is divided into two sections, 0-40k and 64k-112k, in addition to the separate space available to pure procedures. Whenever space becomes available it is allocated to the first job in the queue that will fit, and so on down the line. If a job waits for more than a certain time, depending on the amount of store requested and whether it is on- or off-line, it joins a second queue which is served on a first come first served basis, and no allocation of the region 0-40k is made to jobs not in this queue until it is empty. This ensures

that all jobs get run eventually without all the inefficiencies of a simple first come first served system.

Swapping or shuffling is not employed because of the length of time required to access the disc, which is a slow one, and the absence of efficient hardware instructions for moving around the contents of the store. Also the base register, which gives the actual starting place of a program in the store is OR-ed with rather than added to the program's address which means that a program of size N must start in an absolute store address which is a multiple of 2^M where $2^M \geq N$. This means that there is only a restricted scope for closing up the gaps.

For scheduling a queue of jobs is maintained and at any stage the first one free to go is run. At 1.28 second intervals the queue is examined and the job that has used the most processor time in this period is put at the end of the queue. This ensures that each job gets run at least every 1.28 x N seconds where N is the number of jobs in the queue assuming it is free to go. Any job that uses up relatively little processor time and is usually halted, such as console and tape or disc limited jobs, has high priority when it is free. The algorithm combines the best features of the Round Robin and Foreground Background algorithms.

Chapter 3

STATISTICAL MEASUREMENTS

The first statistic that was measured was the disc access time. Two figures are of interest here, the length of time required to read a block and the time required to position the arm and verify. These were measured experimentally on an empty machine, to avoid queueing delays or multiprogramming causing an error in the real time measurement. From a fixed position on the disc with no arm movement 420 blocks were transferred in seven runs, respectively one block to seven blocks at a time, and the mean time between the beginning of the transfer and the end, which is determined by when the store is unlocked and the last word can be read, was measured. A similar experiment was also done with the arm in a random position on the disc instead of the correct position. The results of these are given in Table 3.1 and Figure 3.1.

With the arm in the correct position the mean time per block is 183 milliseconds, irrespective of the number of blocks transferred. The mean time required to position the arm is 153 milliseconds.

The remaining statistics were gathered from the log automatically collected by the supervisor and stored on magnetic tape for subsequent analysis. This gives details of the starting and finishing time for each job, whether it is off-line, on-line or logging in on-line, details of which system programs and how much computation time, execution time, and store it uses. When a console user first logs in his name is not known and so a job for a standard notional user is started which enters the logging in program. When

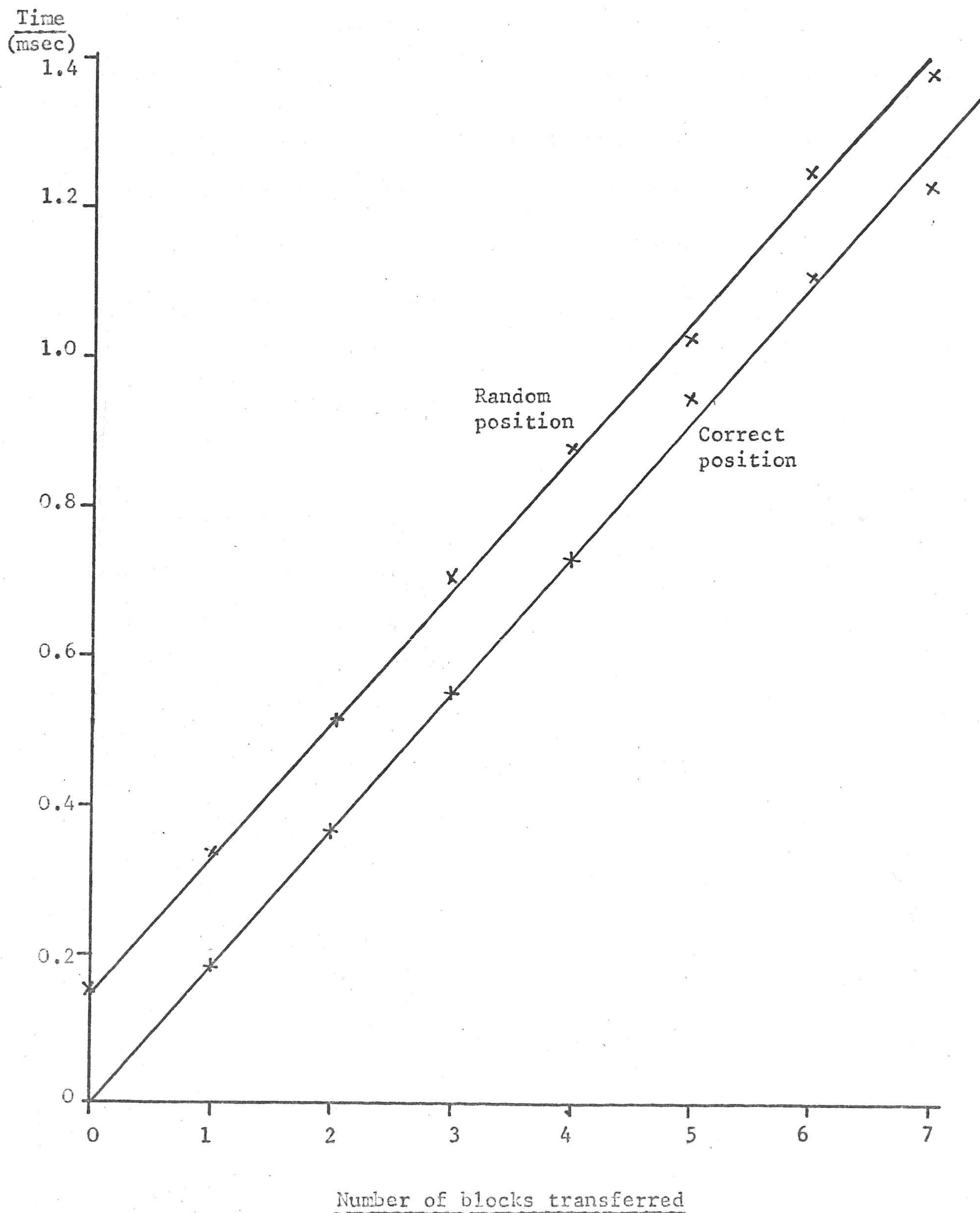
Table 3.1

Disc access times with arm in correct and random positions

Arm	Number of blocks	1	2	3	4	5	6	7
Correct Position	Total time (ms)	185	365	551	732	943	1106	1230
	Time/block (ms)	185	182	184	182	189	184	176
Random Position	Total time (ms)	338	501	708	878	1120	1256	1382
	Positioning time (ms)	153	136	157	146	177	150	152

Figure 3.1

Disc access times with arm in correct and random positions



Number of blocks transferred

the user has established his identity and stated what system resources he requires a new job is created with the correct title and the old job is terminated. This new job may be subject to scheduling delays if his requirements are large to even out the load.

If there is a serious software or hardware failure all console jobs that have finished logging in will be restarted automatically as will all off-line jobs which are still running, and thus it is desirable to create a second job whose name and resource requirements are known, to avoid the user having to type these again, since this information can be stored on the disc.

The execution time for a phase is the sum of the computation time and any time that the phase is in store and held up through its own fault. This can be for several reasons, the program may be waiting for input from or output to a console, it may be waiting for transfers from tape or disc, or it may have deliberately requested to wait, for example in the hope that some filing system interlock may clear. It does not include any time waiting for stream transfers, which varies according to the reliability of the disc, how full the core is, how much multiprogramming is going on, or any time the supervisor spends running other programs when it is free to go. For on-line jobs it roughly represents the amount of time required to receive the output from the previous phase, think about it, type in the input for the next phase and load the program from the disc.

This must be qualified by the remarks made earlier that some phases follow each other without any interaction and other phases are

interactive themselves, but this will be discussed later when the command usage is analysed.

An analysis program was written in Fortran and machine code to gather statistics from the details for each job recorded in the log. This proved to be a much more difficult task than expected, since many unforseen problems occurred. The most important of these was the wide variety in the results. This necessitated analysing data over a much longer period than was originally intended and precluded the possibility of sampling. This contrasts with Scherr[54,55] who found the users of CTSS relatively consistent in their habits.

Some of the variation is explained by terms, holidays, etc. which affect the nature of the load, if the on-line load or the number of short jobs is small then more long jobs are run which puts up the mean times. There are nearly always enough long jobs waiting to be run to keep the machine busy. Also during this period the size of the core store was increased from 64K to 128K, but since this was done intermittently for testing before being finalised it was impossible to separate the figures relating to each store size. The main effect of this was to improve efficiency by reducing disc transfers and increase the throughput. The mean times should not be affected significantly.

Because the data was automatically collected there was a number of errors or exceptional cases which it took some time to discover and protect against. These were caused by factors like restarts, errors in the logging program and possibilities which were not considered when the analysis program was first written. Some of

these produce obvious absurdities or faults which can easily be corrected but others proved more difficult to detect since the volume of logging information was too great to scrutinise by eye for errors.

A system had to be devised to store intermediate information to avoid complete reruns in the case of a fault occurring such as an unreadable block on the tape, or an error in the data,or the time allowed running out. The Fortran routines proved after much experiment to be unusable so machine code routines were written to store tables and blocks of information on streams and recover the relevant ones for the next run.

All the information was stored cumulatively, but details of the most important features of each stream on the logging tape, corresponding typically to one day's use, were printed. As an example of the variation the range of values of the daily means of comp. time and halt time (= exec. time - comp. time) are given in Table 3.2. The variations in the on-line figures are wider than the off-line ones because of changes in the nature of the load. The main statistics are given in Table 3.3 and the distributions of the number of jobs running are given in Table 3.4 and Figure 3.2.

The remaining statistics were gathered over a much shorter period (19/8/68 - 29/8/68). This was partly because the original data had disappeared, since the logging tapes are reused on a cyclic basis and at the time it did not seem worthwhile preserving them. Also it takes about 25 minutes of Titan time to process each tape even if relatively little information is extracted, and since the subsequent figures refer mainly to distributions rather than means it did not seem worthwhile spending as much time as before to achieve

Table 3.2

Ranges of daily mean comp. and halt times

	Off-line		On-line	
	Min (sec)	Max (sec)	Min (sec)	Max (sec)
Comp.	1.97	120.05	0.79	1.92
Halt	1.41	31.25	36.18	77.87

Table 3.3

General Statistics for 7/4/68 - 30/5/68

No. of restarts	457
Total time (min)	35600
Disc transfers:	7462081
Filing system	1402098
Well	2663715
Object programs	1019060
Others	2377208

	Off-line	On-line	Total
Total comp. (min)	25300	4467	29767
% comp.	71	12.5	83.5
Jobs	20594	15278	35872
Logging in jobs		3575	
Phases	79349	175507	254856
Throughput (phases/min)	2.22	4.92	7.15
Mean comp. (sec)	19.13	1.51	7.00
Mean halt (sec)	9.36	44.59	34.04
Mean store (K)	5.80	2.27	3.37
Mean number running	5.37	4.61	9.98

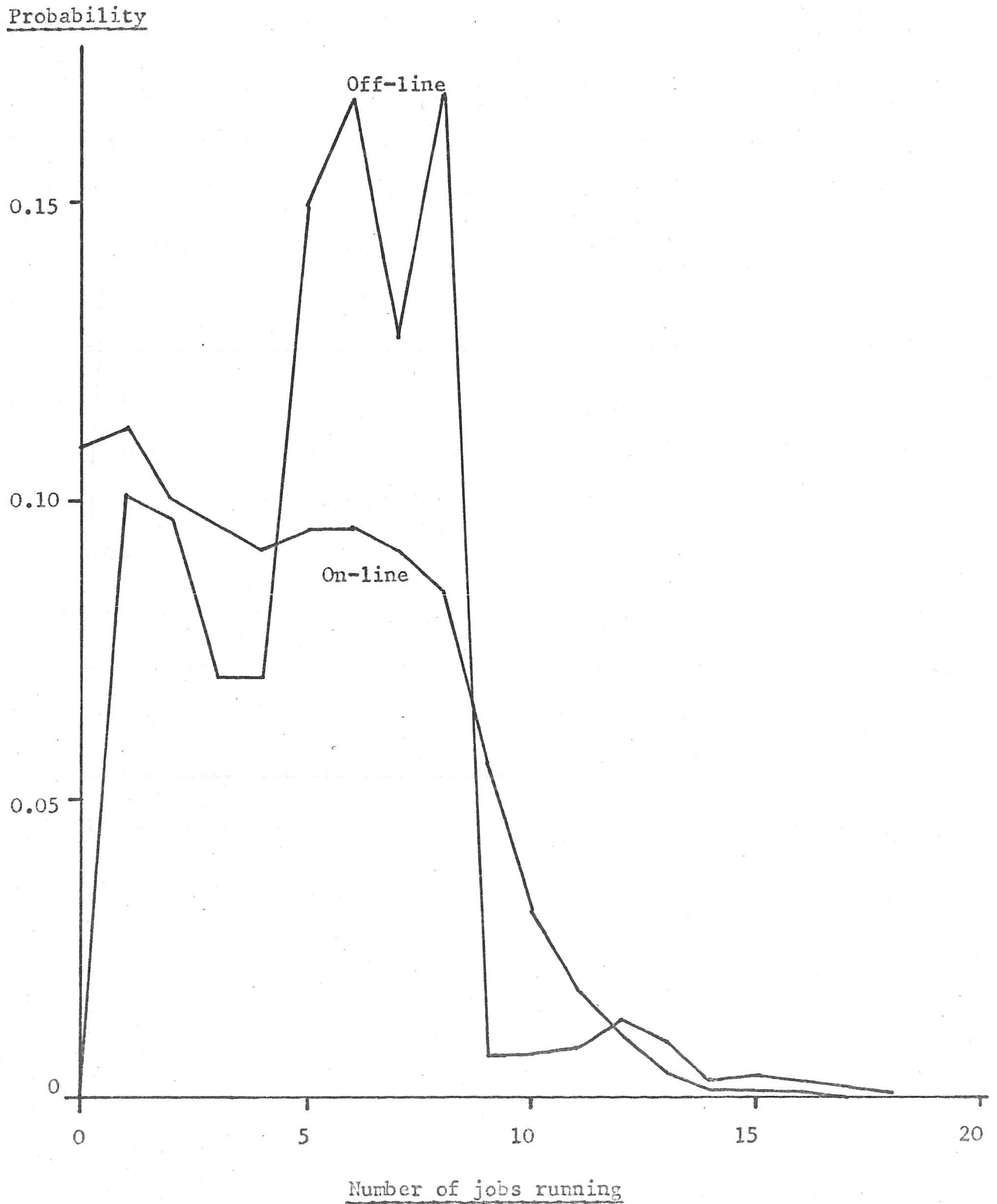
Table 3.4

Distributions of number of off-line and on-line jobs running

Number of jobs running	Off-line probability	On-line probability
0	.001	.109
1	.100	.112
2	.097	.100
3	.070	.096
4	.071	.092
5	.149	.095
6	.167	.095
7	.127	.092
8	.167	.085
9	.007	.056
10	.007	.031
11	.008	.018
12	.012	.013
13	.009	.004
14	.002	.002
15	.003	.001
16	.002	.001
17	.001	.001
18	.000	.000
19		.000

Figure 3.2

Distribution of number of off-line and on-line jobs running



accuracy, since it is not so crucial in this case. Also just processing one tape made it relatively simple to extract further statistics when these appeared necessary.

Some of the results were directly comparable, the means for the halt times were within 3% of the earlier values though the mean off-line and on-line comp. times were 33.9 and 1.1 seconds respectively. The difference between these values and the previous ones mainly reflects the fact that the second set of data is out of term in the summer when the on-line load is light and more long off-line jobs get run. The pattern of store sizes was also much the same, except that the mean size of off-line phases had increased from 5.8K to 7.4K for the same reason. Thus it is reasonable to treat the results as typical of a period with a slightly light on-line load.

The first object of these supplementary measurements was to find a satisfactory method of comparing the distribution of comp. and halt times with the exponential distribution to see if it is possible to use this. The original measurements were tabulated in a table with linear intervals and a cut-off point, but this was unsatisfactory since to get a reasonable number of cells the intervals must be too large to get adequate resolution at the low end of the table and the cut-off point too small to compare the tail end of the distributions. Also it is difficult to find an interval which will give adequate detail without producing too jagged a curve.

The solution adopted was to tabulate the log of the times, which gives sufficient detail at the low end without requiring a cut-off. The number of entries in each section of the table was

plotted on a logarithmic scale for the same reason. The tabulation and graph plotting routines developed for simulation were used for this.

To get the exponentially distributed graph corresponding a technique which is potentially useful for simulation was developed to generate the best possible set of a given number of exponentially distributed variables. The object of this was to produce as smooth a graph as possible and obviate unnecessary error owing to the use of random numbers.

Suppose we require N exponentially distributed numbers with mean unity (other means are possible by multiplying every number by the new mean). We divide the range into N intervals bounded by $x_N = 0, x_{N-1}, x_{N-2}, \ldots x_0 = \infty$. If an exponentially distributed variable has an equal chance of falling in each interval then we have:

$$\int_{x_i}^{x_{i-1}} e^{-x}\, dx = \frac{1}{N} \qquad e^{-x_i} = \frac{i}{N} \qquad x_i = -\log\frac{i}{N} \qquad (i = 0,1,\ldots N)$$

We now choose our exponentially distributed number y_i in the interval (x_i, x_{i-1}) such that its value is the expectation of a number drawn from the distribution which lies in that interval.

$$y_i = \frac{\int_{x_i}^{x_{i-1}} xe^{-x}\, dx}{\int_{x_i}^{x_{i-1}} e^{-x}\, dx} = N\Big[-xe^{-x}\Big]_{x_i}^{x_{i-1}} + N\int_{x_i}^{x_{i-1}} e^{-x}\, dx$$

$$y_i = N\Big[-(1+x)e^{-x}\Big]_{x_i}^{x_{i-1}} = i(1-\log\frac{i}{N}) - (i-1)(1-\log\frac{i-1}{N})$$

$$y_i = 1 + \log N + (i-1)\log(i-1) - i\log i$$

These may be conveniently calculated and tabulated in turn, only one log operation being required for each, the same as for random sampling from an exponential distribution.

Figures 3.3-3.6 show the quantities of interest, namely off-line and on-line comp. and halt times. The time is given on a logarithmic scale starting at 0.1 which is the lowest that can be measured, any items with value 0 are also entered here, and the kink in the theoretical distribution is due to all those items with value less than 0.1 being included. The increment in $\log_{10}(\text{time})$ for the table is 0.2, which seems to combine resolution and smoothness. The height of the curves is proportional to the log of the number of entries in a given cell of the table, adjacent points on the graph being joined by straight lines.

As might be expected there is a certain amount of divergence between the actual and the exponential distributions, the chief difference being that the standard deviation of the actual distributions is higher than that of the exponential ones, there being more very high and very low entries and fewer near the mean.

To get a better fit we must use a derived distribution, where one chooses according to a set of probabilities the mean of the exponential distribution to be sampled from. Distributions of this sort have been used before[19,52,58,59,62] to provide a better fit than a simple exponential, and are known as hyperexponential. If we examine Tables 3.5 and 3.6 and Figure 3.7 which gives the mean comp. times according to store size we see that there is a significant correlation between the two as would be expected.

Table 3.5

Frequency and length of off-line phases by store size

Store K	Phases		Comp.		Halt	
	Number	%	Total min	Mean sec	Total min	Mean sec
Pure*	2908	18	110	2.3	272	5.6
1-4	5668	35	1658	17.5	1009	10.7
5-8	3592	22	1254	20.9	492	8.5
9-12	1497	9	1070	42.9	144	5.7
13-16	656	4	693	63.4	93	8.5
17-20	441	3	417	56.7	17	2.3
21-24	405	3	521	77.2	100	14.8
25-28	259	2	161	37.3	33	7.6
29-32	427	3	531	74.6	150	21.0
33-36	178	1	583	196.4	46	15.4
37-40	238	2	2186	550.0	130	32.8

*Pure procedures with 64 words working space

Table 3.6

Frequency and length of on-line phases by store size

Store K	Phases		Comp.		Halt	
	Number	%	Total min	Mean sec	Total min	Mean sec
Pure*	41870	65	496	0.7	4425	63.0
1-4	18042	28	176	0.6	1974	6.6
5-8	2721	4	243	5.4	64	1.4
9-12	1814	3	138	4.6	67	2.2
13-16	292	1	85	17.4	1	0.3
17-20	88	0	16	10.9	0	0.3
21-24	24	0	12	28.9	1	2.3
25-28	0					
29-32	5	0	0	2.8	2	29.0
33-36	0					
37-40	0					

*Pure procedures with 64 words working space

Figure 3.3

Distributions of off-line comp. times

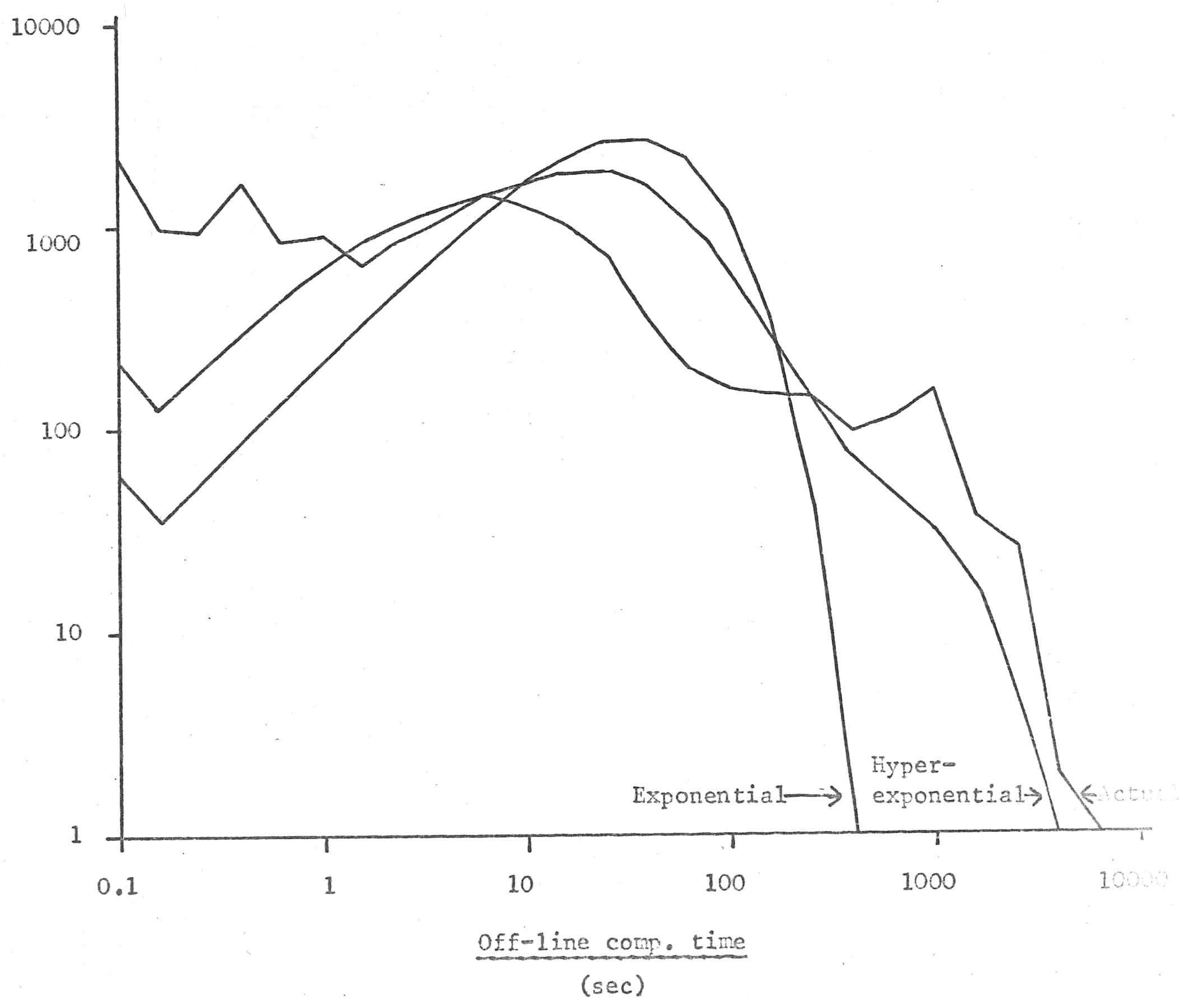


Figure 3.4

Distributions of on-line comp. times

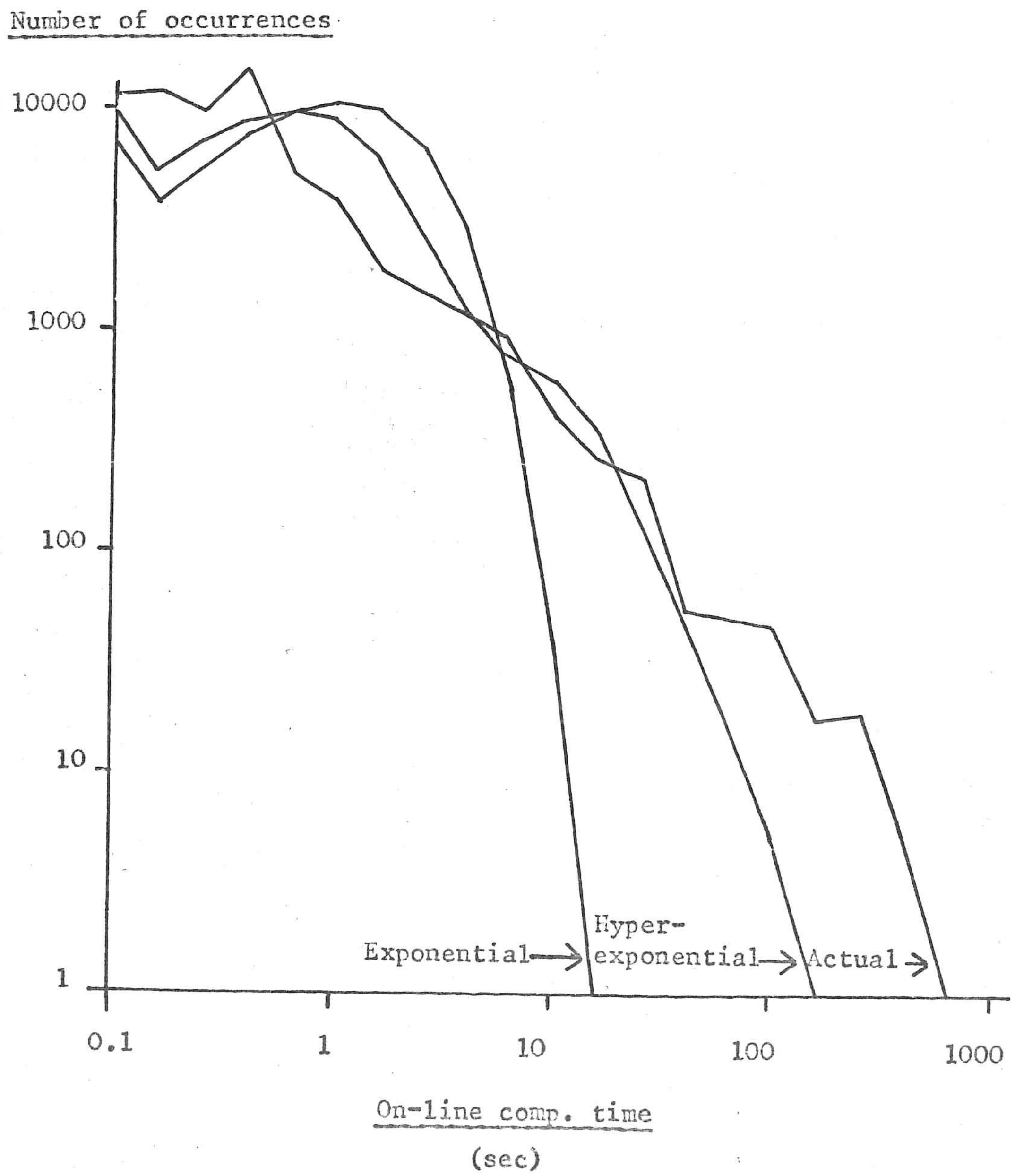


Figure 3.5

Distributions of off-line halt times

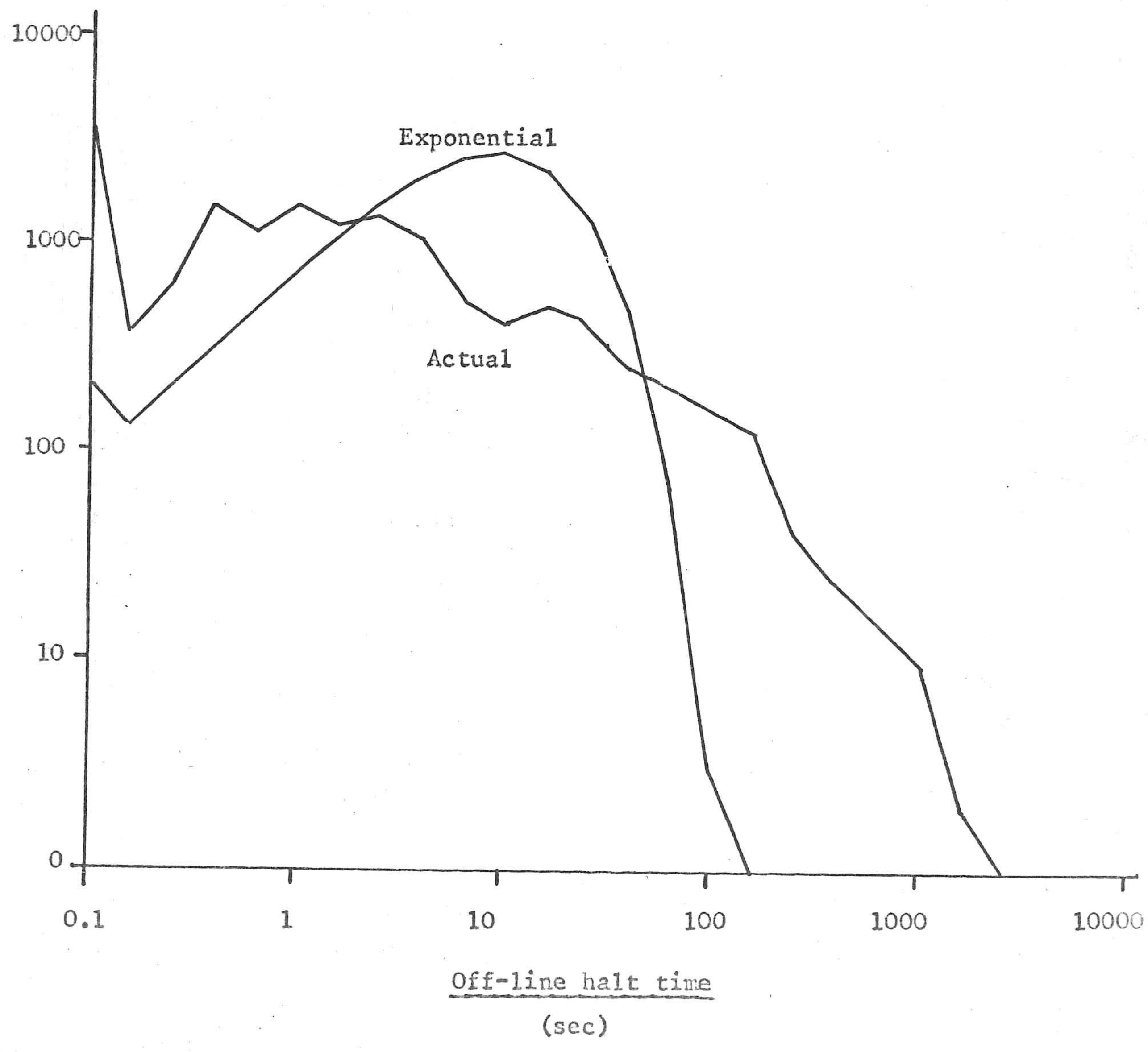


Figure 3.6

Distributions of on-line halt times

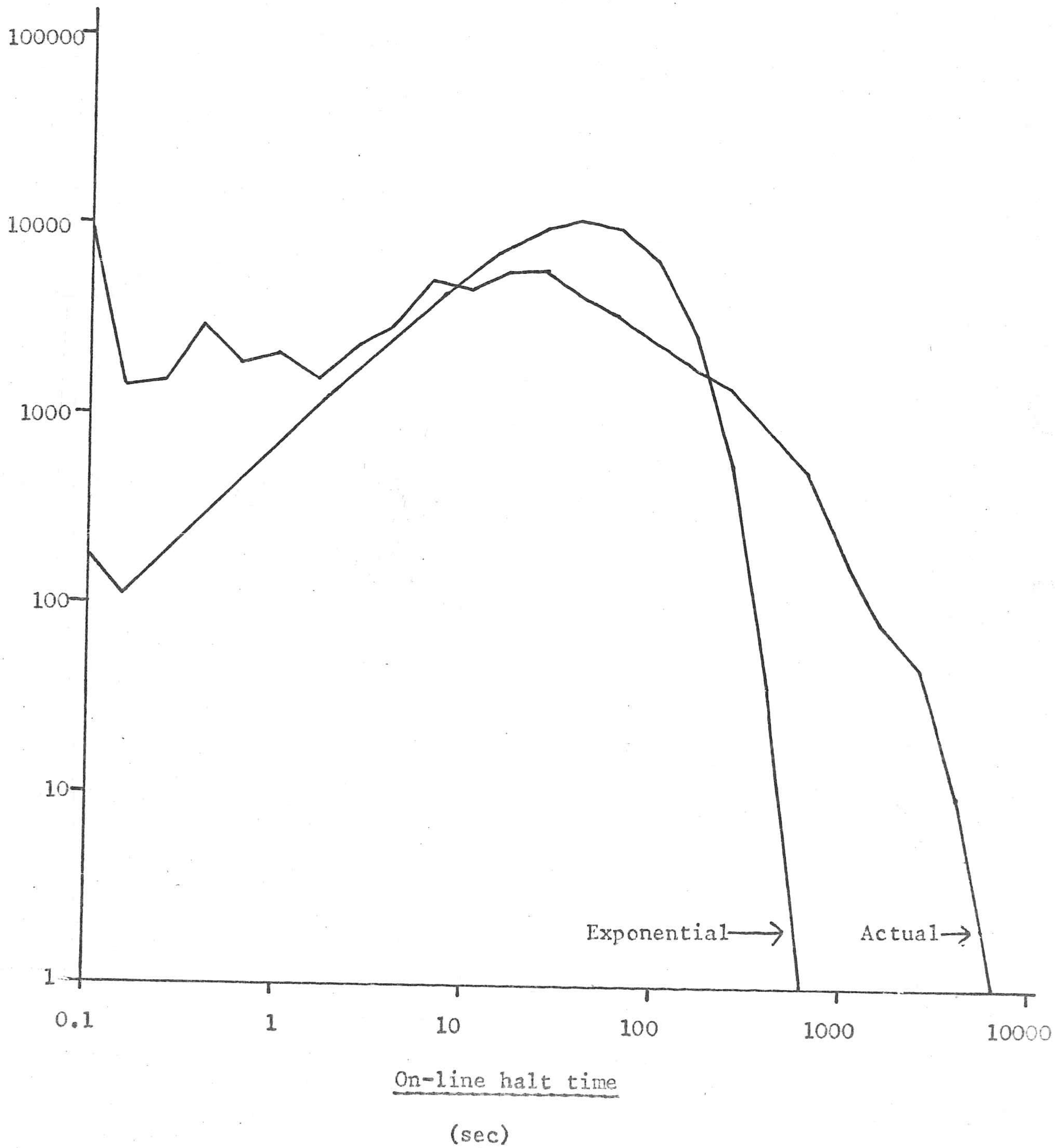
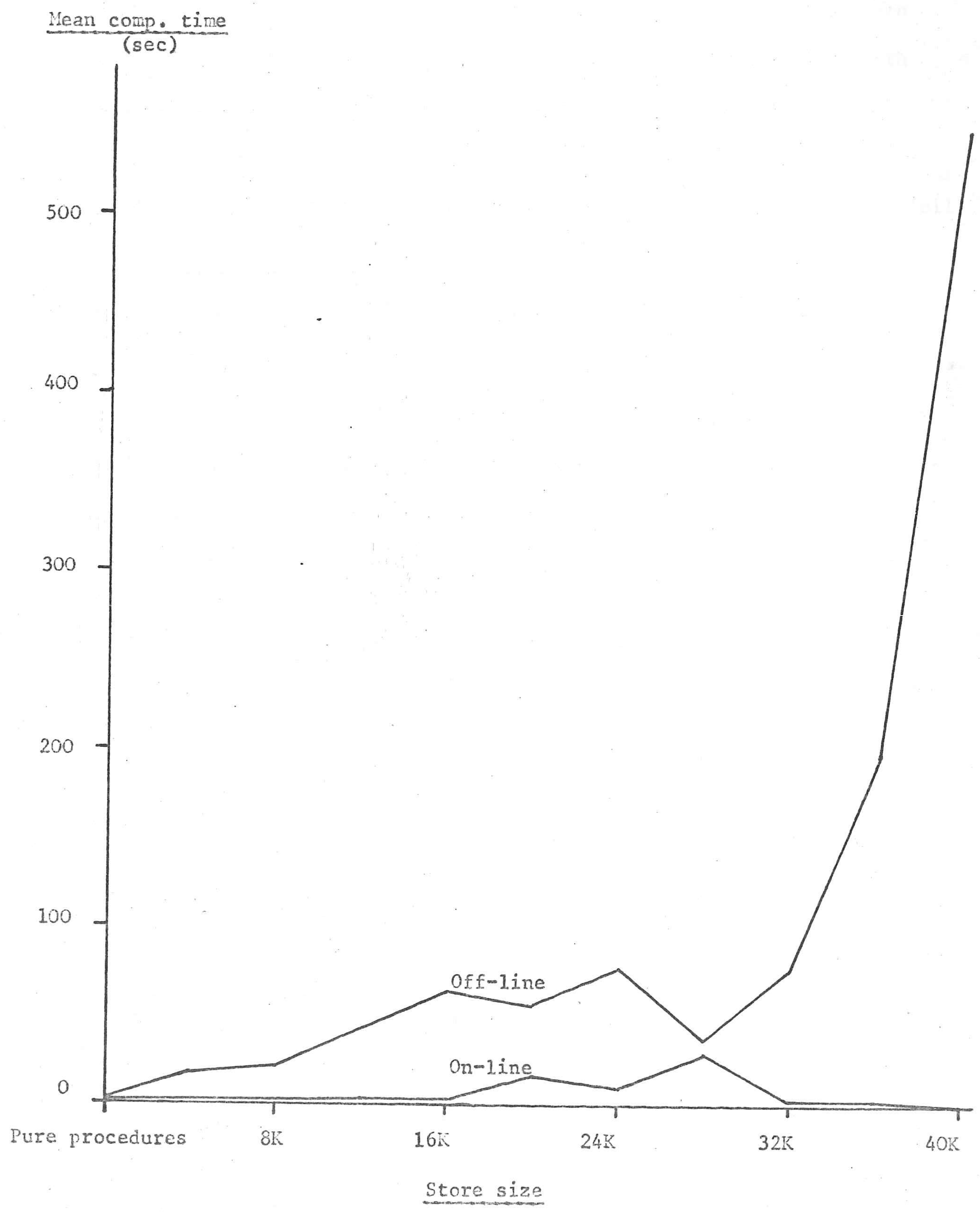


Figure 3.7

Mean comp. time of phases against store size



Some unevenness is caused by compilers of particular sizes which are used a lot. Tables 3.7 and 3.8 show the most commonly used systems for off-line and on-line jobs respectively and it can be seen how a few systems account for much of the time. Pure procedures with 64 words of working space are marked with an asterisk; these are classified separately since there is only one copy for all the users and this goes in an area reserved specially for them which is rarely full.

Using hyperexponential distribution for comp. times with a different mean for different store sizes we get a much better fit than the exponential as can be seen from Figures 3.3 to 3.6. This distribution was used for the simulations, but the simple exponential was used for the analysis, since the results from that appear to be relatively insensitive to distribution. For the halt times Tables 3.5 and 3.6 show that there is little correlation and so the simple exponential distribution provides a better fit.

Table 3.7

Off-line system usage

System	Phases		Comp.		Halt	
	Number	%	Total min	Mean sec	Total min	Mean sec
MLS	12316	14.8	95	0.5	578	2.8
IIT	10888	13.1	1933	10.7	515	0.8
IIT PROG	9852	11.9	17069	104.0	2881	17.5
*COMMAND	9807	11.8	21	0.1	54	0.3
AUTOCODE	8418	10.1	1015	7.2	508	3.6
*E3	7641	9.2	373	2.9	725	5.7
FMJOB	3684	4.4	5	0.1	0	0.0
*FINISH	2611	3.1	6	0.1	35	0.8
IAL	2540	3.1	281	6.6	192	4.5
PRIVATE	2480	3.0	649	15.7	803	19.4
FORTRAN	2443	2.9	248	6.1	179	4.4
IAL PROG	2149	2.6	1951	54.5	146	4.1
FILELOG	1861	2.2	386	12.4	814	26.2
RELOAD	1302	1.6	150	6.9	2428	111.9
*EDIT	1064	1.3	123	6.9	94	5.3
*ETAL	571	0.7	11	1.1	10	1.0
FILEM	494	0.6	137	16.6	1162	141.1
FMACTION	471	0.6	2	0.2	4	0.5
MACRO1	332	0.4	342	62.0	22	4.0
FILEDUMP	300	0.4	83	16.5	212	42.4
CRYST	278	0.3	1	0.1	14	3.1
MAKARCH	232	0.3	70	18.2	624	161.4
OWN	175	0.2	34	11.8	19	6.4
TSAS	168	0.2	201	71.9	627	223.9
SET	154	0.2	0	0.1	0	0.0
SIGNAL	136	0.2	1	0.2	2	0.8
EXAMINE	116	0.1	1	0.3	3	1.4
UNDUMP	112	0.1	2	1.0	50	26.8

*Pure procedure with 64 words working space

Table 3.8

On-line system usage

System	Phases		Comp.		Halt	
	Number	%	Total min	Mean sec	Total min	Mean sec
*COMMAND	76956	43.8	944	0.7	71639	55.9
*EDIT	17486	10.0	552	1.9	36594	125.6
PRIVATE	17208	9.8	249	0.9	821	2.9
*LOGIN	11703	6.7	116	0.6	6388	32.7
MLS	7280	4.1	56	0.5	287	2.4
*FINISH	7023	4.0	533	4.6	16749	143.1
IIT	5336	3.0	393	4.4	15	0.2
EXAMINE	5049	2.9	32	0.4	3348	39.8
IIT PROG	4947	2.8	596	7.2	540	6.6
*RUNJOB	2987	1.7	7	0.1	3	0.1
IAL	2286	1.3	181	4.7	50	1.3
SET	1966	1.1	2	0.1	0	0.0
IAL PROG	1960	1.1	197	6.0	57	1.8
AUTOCODE	1493	0.9	96	3.9	30	1.2
FORTRAN	1790	0.8	80	3.2	124	5.0
FSPACE	1449	0.8	2	0.1	1	0.0
*ETAL	1191	0.7	38	1.9	10	0.5
COPY	961	0.5	4	0.3	55	3.5
STATUS	715	0.4		0.1	0	0.0
SIGNAL	528	0.3	2	0.2	8	0.9
MESSAGE	525	0.3	1	0.1	0	0.0
HELP	519	0.3	20	2.3	0	0.0
TIME	508	0.3		0.1	0	0.0
*E3	466	0.3	16	2.1	5	0.7
STREAMS	448	0.3		0.1	0	0.1
TSAS	436	0.2	83	11.4	190	26.2
PDP	376	0.2	8	1.3	15	2.4
UNLOCK	301	0.2		0.1	34	6.8
TESTSWIT	295	0.2	1	0.1	0	0.0
FIGARO	238	0.1	6	1.6	956	241.1
MACRO1	232	0.1	80	20.8	7	1.7
CROSSREF	204	0.1	38	11.2	2	0.6
RECOVER	114	0.1		0.1	0	0.0
FILEDUMP	109	0.1	32	17.6	54	29.5
RELATION	100	0.1		0.1	5	2.9

*Pure procedure with 64 words working space

Chapter 4

ANALYTICAL MODELS

Computer systems have been modelled analytically in various ways, which reflect the differences between systems and the particular interests of the analyst. Several types of model will be developed, the first of which is a deterministic one. Here we consider the flow of jobs through the system at a constant rate spending a constant time in each section of the system. In spite of these assumptions these models have several advantages.

First the resulting equations are extremely simple, and enable one to see clearly and explicitly the relationship between certain quantities. This relationship is often similar to or the same as more complicated models where the results can only be evaluated numerically. Sometimes systems can be modelled which would otherwise prove too difficult. Finally they provide a useful basis for comparison, for checking results from more complicated analytical models and simulations and seeing how much is gained or lost in practice by the variability of programs. They can give a good idea of the means of various quantities, such as waiting times, queue sizes and idle time, but they cannot tell us anything about the distributions.

Kleinrock(36) used a deterministic model to define the concept of saturation in a simple multi-access system, and compared the results with a Markovian model. Penny(49) used two such models to get upper and lower bounds for the performance of a system with various parameter values for interpolating between simulation results as described in Chapter 7.

Figure 4.1 shows the deterministic model in diagrammatic form. We regard the system as a set of stages of service denoted by square boxes with the service time for each stage given inside each box. In this model all the service times are constant. The flow of jobs through the system is indicated by the lines, dotted lines showing where there is a choice of route.

There is assumed to be an infinite supply of off-line phases which have two stages of service, first by the discs and then by the processor. Although in fact most jobs will have several phases and go through these stages several times we are only considering the total number of phases run and not the job they are part of.

On-line jobs also have these stages of service, with a different processor time, but they also have a third stage, corresponding to input-output and thinking time at a console. This is necessary because we want to consider the performance of the system with a fixed number of on-line jobs running rather than an infinite supply as for off-line jobs. A delay in one phase of a job will delay the start of the next phase, and so on, and so the system is stable as no backlog is possible.

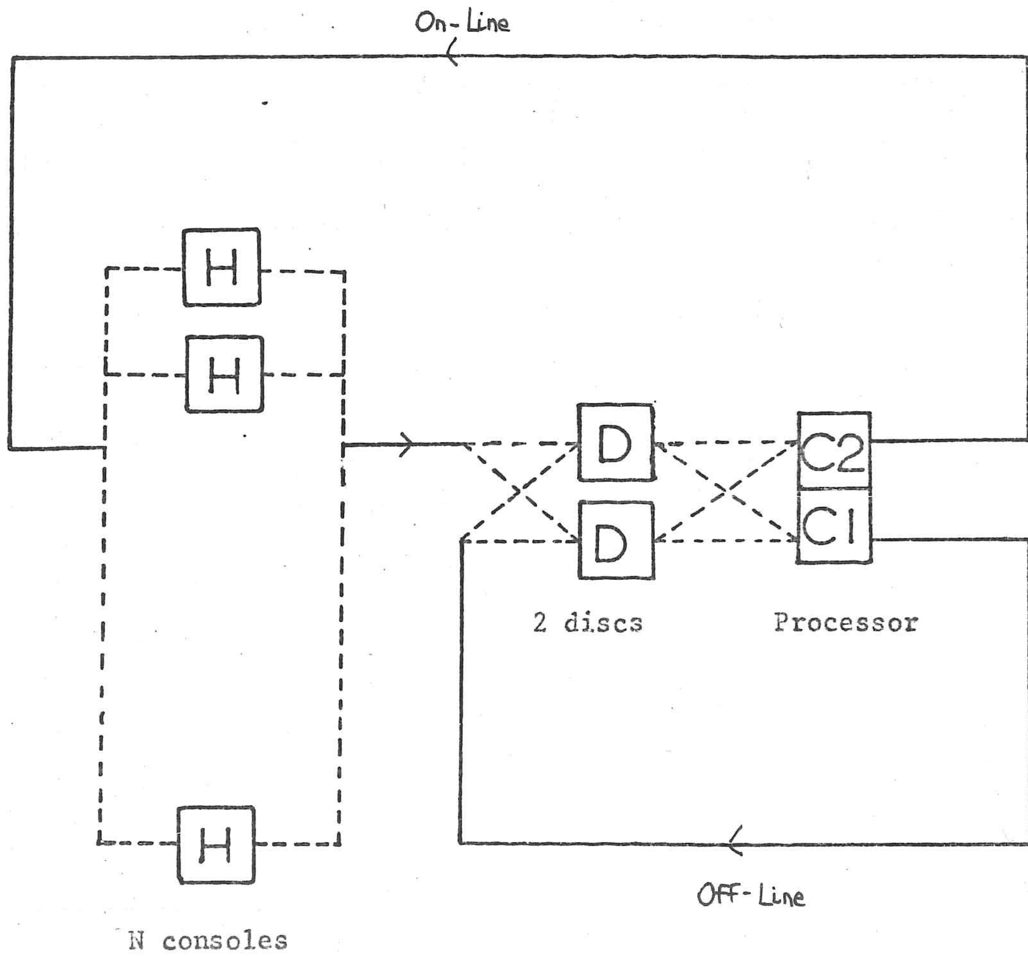
We assume that all the disc activity is concentrated at the beginning of a phase to give a clearer model, though it makes no difference even for the statistical model as will be proved later. The figures used for the constant values are the means derived from observation of the Cambridge system given in Table 3.3, namely:

C1	= 19.1 sec	Mean comp. time for off-line phases
C2	= 1.51 sec	Mean comp. time for on-line phases
H	= 42.0 sec	Mean thinking time for on-line phases
D	= 5.6 sec	Mean disc time for all phases

H is derived from the mean halt time (= exec. time - comp. time) which was 44.59 sec, by subtracting the amount of disc time that is

Figure 4.1

Deterministic model of Cambridge system



chargeable to exec. time, namely that which is not used for streams. D is derived by noting that there are roughly 28 disc transfers for each phase, 10 for the filing system and object programs, 10 for the well, and 8 for other reasons, including fixed head transfers for the supervisor which are quicker. If each of these except the latter take 0.183 seconds and there are typically 8 arm repositionings taking 0.153 seconds, the total comes to about 5.6 seconds. This figure is unfortunately not very accurately determined, since we do not know how many arm repositionings there actually are, or even whether the amount of disc activity is proportional to the number of phases as has been assumed, but it is impossible to get further details within the present system.

This model is solved by equating the arrival and departure rates at each stage to the throughputs (T1 for off-line jobs, T2 for on-line jobs). If this were not true in equilibrium we would have an indefinite build-up of jobs at one stage. We have not included the number of off-line jobs in core explicitly since this only affects the waiting time for off-line jobs which is not of great importance. We can compare different scheduling algorithms according to the values of T1 and T2 they produce, whereas later models only enable us to consider a few alternatives.

Both the discs and the processor can be in one of two states. Either they can be saturated, that is working all the time, or unsaturated, that is idle some of the time. The condition for saturation of a stage is that the throughput multiplied by the effective service time by the stage should be equal to one, if this is less than one it gives the utilisation of the stage. This gives us

the following two conditions:

$$(T2 + T1)xD/2 = DU \leq 1 \quad \text{(2 discs)}$$

$$(T1 \times C1 + T2 \times C2)/V = CU \leq 1$$

where CU is the processor utilisation, DU is the disc utilisation, and V is the system overhead factor, that is the fraction of the processor time spent in obeying user's programs. This is taken to be 0.833 though later measurements indicate that it has risen to nearer 0.9. This may be the result of increasing the store size or just random variation.

In practice we can assume an infinite supply of off-line jobs, so if we are given T2 we can then work out T1 to be the maximum that just saturates either the discs or the processor, that is:

$$T1 = \text{Min}\left(\frac{2}{D} - T2, \frac{V - T2 \times C2}{C1}\right)$$

If a stage is not saturated the waiting time is equal to the service time. Thus we have a maximum on T2 imposed by the fact that it will take a certain time for a job to pass through all the 3 stages even if there are no delays at all, that is:

$$T2 \leq \frac{N2}{H + D + C2/V}$$

where N2 is the number of on-line jobs running.

The effect of saturation of a stage will be to cause waiting at that stage to make the total time required to pass round the circuit multiplied by the throughput equal to the number of consoles, that is:

$$T2 = \frac{N2}{H + DW + CW}$$

where DW is the disc waiting time for on-line jobs and CW is the processor waiting time for on-line jobs.

Hence we can calculate the sum of the waiting times, i.e. the response time, and the two times individually except in the case when both stages are saturated, where the individual waiting times depend on the initial conditions.

Finally the mean queue length for a stage is equal to the throughput multiplied by the waiting time for it. This fact is obvious for queues with constant arrival and service times, since the waiting time is equal to the mean queue size multiplied by the mean service time. In fact it is also true for all queueing systems with exponentially distributed arrival and service times.[43] The mean queue size for an unsaturated stage is the same as the fraction of time it is in use, since there will never be a queue of more than one. If there were by the next time round the system the arrivals would have evened themselves out to correspond to times when the stage was free. The queue length includes the current user of a stage.

The results of this model are given in Table 4.1 and Figure 4.2. The throughputs (T1 and T2) are given in phases/min for convenience of comparison, whereas in the equations they are in phases/sec. As can be seen from the equations the waiting time is a function only of the throughput per console (T2/N), whereas the mean number of consoles waiting for the disc or the processor (Q) and the fraction of processor time used by on-line jobs (CU2) including overhead are proportional to the throughput (T2), or to the number of consoles for a given value of T2/N. We can take the arithmetic mean of the number of on-line jobs running measured from the system and compare the results, using linear interpolation. The values were:

Table 4.1

Deterministic model of Cambridge system

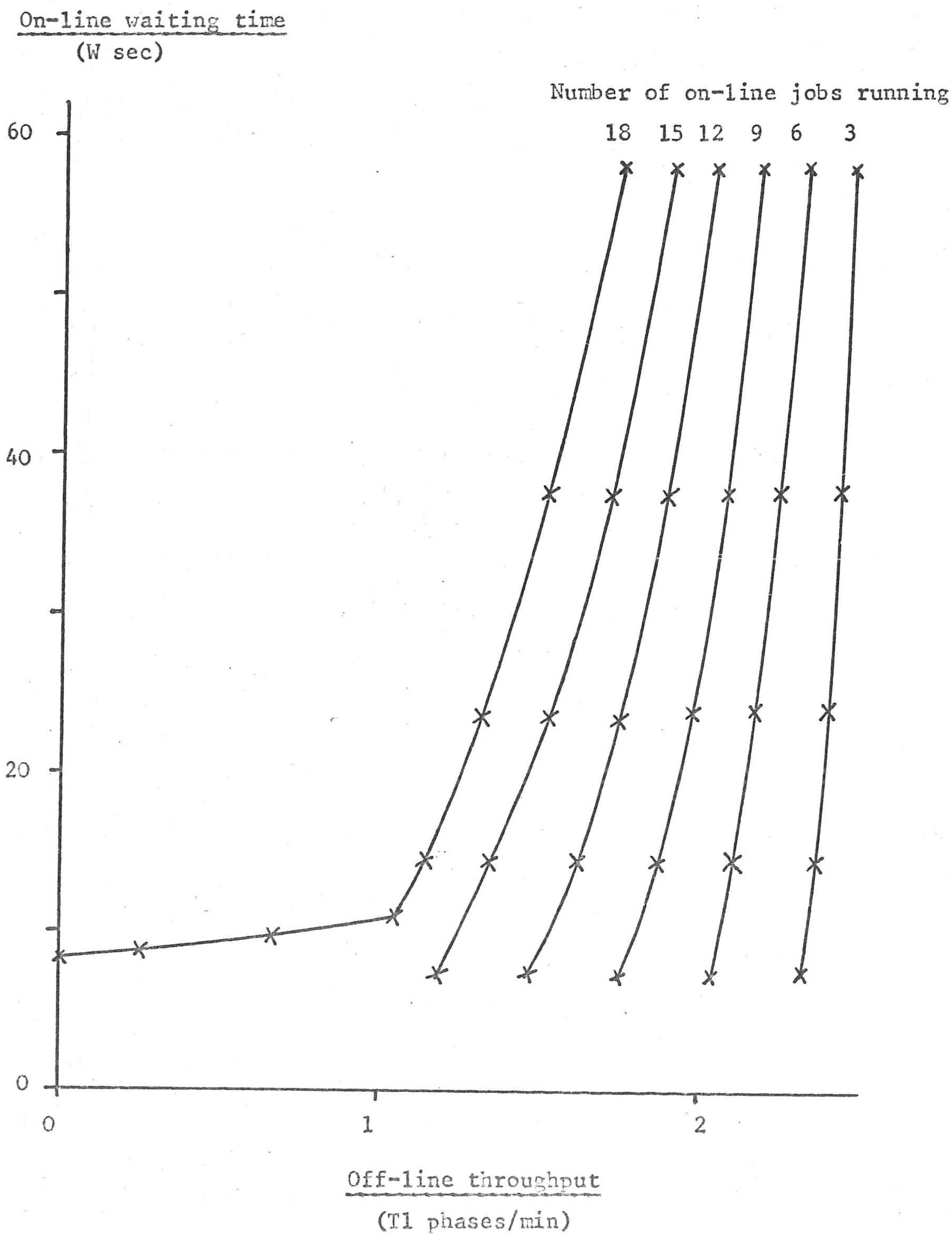
N	T1	T2	T2/N	W	Q	CU1	CU2	DU1	DU2
3	2.47	1.80	0.60	58.00	1.74	0.95	0.05	0.12	0.08
3	2.44	2.27	0.76	37.37	1.41	0.93	0.07	0.11	0.11
3	2.40	2.74	0.91	23.79	1.08	0.92	0.08	0.11	0.13
3	2.36	3.20	1.07	14.18	0.76	0.90	0.10	0.11	0.15
3	2.33	3.64	1.21	7.41	0.45	0.89	0.11	0.11	0.17
6	2.33	3.60	0.60	58.00	3.48	0.89	0.11	0.11	0.17
6	2.26	4.54	0.76	37.37	2.82	0.86	0.14	0.11	0.21
6	2.18	5.47	0.91	23.79	2.17	0.83	0.17	0.10	0.26
6	2.11	6.41	1.07	14.18	1.51	0.81	0.19	0.10	0.30
6	2.04	7.29	1.21	7.41	0.90	0.78	0.22	0.10	0.34
9	2.19	5.40	0.60	58.00	5.22	0.84	0.16	0.10	0.25
9	2.08	6.80	0.76	37.37	4.24	0.79	0.21	0.10	0.32
9	1.97	8.21	0.91	23.79	3.25	0.75	0.25	0.09	0.38
9	1.86	9.61	1.07	14.18	2.27	0.71	0.29	0.09	0.45
9	1.75	10.93	1.21	7.41	1.35	0.67	0.33	0.08	0.51
12	2.05	7.20	0.60	58.00	6.96	0.78	0.22	0.10	0.34
12	1.90	9.07	0.76	37.37	5.65	0.73	0.27	0.09	0.42
12	1.75	10.94	0.91	23.79	4.34	0.67	0.33	0.08	0.51
12	1.60	12.82	1.07	14.18	3.03	0.61	0.39	0.07	0.60
12	1.46	14.57	1.21	7.41	1.80	0.56	0.44	0.07	0.68
15	1.91	9.00	0.60	58.00	8.70	0.73	0.27	0.09	0.42
15	1.72	11.34	0.76	37.37	7.06	0.66	0.34	0.08	0.53
15	1.54	13.68	0.91	23.79	5.42	0.59	0.41	0.07	0.64
15	1.35	16.02	1.07	14.18	3.79	0.52	0.48	0.06	0.75
15	1.18	18.21	1.21	7.41	2.25	0.45	0.55	0.05	0.85
18	1.76	10.80	0.60	58.00	10.44	0.67	0.33	0.08	0.50
18	1.54	13.61	0.76	37.37	8.47	0.59	0.41	0.07	0.64
18	1.32	16.42	0.91	23.79	6.51	0.50	0.50	0.06	0.77
18	1.10	19.22	1.07	14.18	4.54	0.42	0.58	0.05	0.90
18	0.00	21.43	1.19	8.40	3.00	0.00	0.65	0.00	1.00

Key

N	Number of on-line jobs running
T1	Off-line throughput (phases/min)
T2	On-line throughput (phases/min)
W	Mean response time for on-line jobs
Q	Mean number of on-line jobs awaiting response
CU1	Processor use by off-line jobs (including overhead)
CU2	Processor use by on-line jobs (including overhead)
DU1	Disc use by off-line jobs
DU2	Disc use by on-line jobs

Figure 4.2

On-line waiting time against off-line throughput for deterministic model



$T1 = 2.22$ phases/min

$T2 = 4.93$ phases/min

$N = 4.61$ on-line jobs

from which we have $T2/N = 1.07$.

By interpolating in Table 4.1 between $N = 3$ and $N = 6$ we get $T1 = 2.24$ phases/min which is remarkably close to the actual figure. We also get by the same method the disc usage $DU = 0.33$ which may seem remarkably low until we remember that in practice the number of on-line jobs varies a lot and we have to be able to cope with the peaks as well. Similarly in practice the waiting time $W = 14.18$ sec will vary when we take into account the statistical nature of the real system.

Considering the amount of computer time used we see by interpolation that $CU1 = 0.85$, $CU2 = 0.15$. If we exclude overhead the fractions of processor time devoted to off-line and on-line work are 0.71 and 0.125 respectively, which is in exact agreement with the observed values. The extent of these agreements and the one noted above are coincidental, but they do indicate the role of this type of model when the performance of the whole system is being considered. However it cannot describe in detail the effect on individual jobs, the value of W being only an approximate measure of the mean response time.

One of the recurrent questions about such systems is whether the on-line response time or the number of on-line jobs running or both can be improved without too serious a drop in the amount of off-line work, or whether these objectives are incompatible.

This model can partially answer these questions by telling us the theoretical optimum results and how much trade-off between objectives is required. It does not tell us whether this can be achieved, or how

to achieve it, because the problems of scheduling and store allocation can only be investigated by analysis or simulation with more sophisticated models.

The reason it gives a theoretical optimum is that randomness in arrival and service times and waiting for core store will produce delays and result in idle time which the model does not account for.

As we can see from Table 4.1 & Figure 4.2, even if on-line jobs are given the maximum possible amount of processor time, i.e. use of the processor whenever there are any waiting, saturation of the disc does not occur until there are 17 on-line jobs running and thus the table provide a valid picture over all this range. When the disc becomes saturated the two expressions for T1 given earlier are equal, i.e.

$$\frac{2}{D} - T2 = V - \frac{T2 \times C2}{C1} = T1$$

This happens when T2 = 21.4, T1 = 1.05. The resulting discontinuity can be seen in Figure 4.2.

If we first consider how much we can improve the response time keeping the number of on-line jobs the same we see that we can cut W almost by half, from 14.18 to 7.41 with CU1 and CU2 being 0.83 and 0.17 respectively corresponding to 0.69 and 0.14 after overhead has been deducted. It seems remarkable that such a change can be made by reducing the time devoted to off-line jobs by only 3% of its value, but this reflects the fact that in its mean state the on-line system is nowhere near saturation.

The mean state however is not the typical state, as a glance at Table 3.4 and Figure 3.2 giving the number of jobs running will show, since at some times during the day there are twelve people

logged in but during the night there are very few. At peak times the degradation required to give this increase in performance is more severe, for example with 15 users the fraction of actual time given to off-line users decreases from 0.43 to 0.38, over 11% of its value. Whether off-line users are prepared to accept the resulting degradation in their turn-round during the day is essentially a political decision.

A similar situation arises when changes to the maximum number of consoles are considered. If the mean rose to 9, which would roughly be the effect of increasing the maximum to 18, keeping the response time the same, the resultant CU1 and CU2 would be 0.71 and 0.29, that is 0.59 and 0.24 after deducting overhead. With 18 consoles actually in operation the figures for CU1 and CU2 are 0.42 and 0.58 respectively and the consoles are now consuming the lion's share of the time in peak periods, though the change in the mean time devoted to off-line work is only 14% of its previous value.

There is an additional danger that once the off-line response time becomes too bad more and more people will do more of their work on-line, thus causing a vicious circle. No computer can serve two masters.

The situation is also slightly altered by the fact that the supply of off-line work is not in fact infinite but depends to a large extent on people getting the results of their old runs back before initiating new ones, thus an increase in the turn-round time will not produce an infinite backlog but stabilise with a larger backlog, the most important effect being a less satisfactory service.

There are two reasons for giving the on-line user priority, first because his requests for service are usually short, and therefore

logged in but during the night there are very few. At peak times the degradation required to give this increase in performance is more severe, for example with 15 users the fraction of actual time given to off-line users decreases from 0.43 to 0.38, over 11% of its value. Whether off-line users are prepared to accept the resulting degradation in their turn-round during the day is essentially a political decision.

A similar situation arises when changes to the maximum number of consoles are considered. If the mean rose to 9, which would roughly be the effect of increasing the maximum to 18, keeping the response time the same, the resultant CU1 and CU2 would be 0.71 and 0.29, that is 0.59 and 0.24 after deducting overhead. With 18 consoles actually in operation the figures for CU1 and CU2 are 0.42 and 0.58 respectively and the consoles are now consuming the lion's share of the time in peak periods, though the change in the mean time devoted to off-line work is only 14% of its previous value.

There is an additional danger that once the off-line response time becomes too bad more and more people will do more of their work on-line, thus causing a vicious circle. No computer can serve two masters.

The situation is also slightly altered by the fact that the supply of off-line work is not in fact infinite but depends to a large extent on people getting the results of their old runs back before initiating new ones, thus an increase in the turn-round time will not produce an infinite backlog but stabilise with a larger backlog, the most important effect being a less satisfactory service.

There are two reasons for giving the on-line user priority, first because his requests for service are usually short, and therefore

easy to satisfy, and second because he is sitting waiting for the results, presumably idle. Because of this latter fact he implicitly states how important a particular set of results is to him by choosing to wait for it on-line, since he can always create the equivalent of an off-line job from a console to run in its own time, provided it does not use paper tape input. If the response time worsens he does more of his work in this way and thus the situation improves again. It is not easy to compare the needs of on-line and off-line users directly, but at least using analysis we know the alternatives open to us .

Another interesting solution to this problem, though rather improbable in practice, is the installation of a second processor. We have seen that it is nearly always the processor that is saturated rather than the disc, and that the system is theoretically capable of providing a much better service to on-line users, but at the expense of off-line users. Since the main resource that these use is processor time, a second processor seems the natural solution.

Table 4.2 gives the results of this model with two processors, each with the same power as the current one. These show that in fact the main beneficiaries are off-line jobs, and if we compare it with Table 4.1 we see that the throughput of these is relatively so much higher that we can afford to give maximum priority for on-line jobs, except when there are over 17 of these running, when again we run into trouble with the discs. Even with 15 on-line jobs running, if we give them maximum priority the disc is fully used and the processor is only used 89% of the time including overhead. This assumes the same overhead, which is reasonable since the extra administrative needs would be balanced by the fact that only one processor need handle interrupts and the

Table 4.2

Deterministic model of two processor system

N	T1	T2	T2/N	W	Q	CU1	CU2	DU1	DU2
3	5.09	1.80	0.60	58.00	1.74	0.97	0.03	0.24	0.08
3	5.05	2.27	0.76	37.37	1.41	0.97	0.03	0.24	0.11
3	5.02	2.74	0.91	23.79	1.08	0.96	0.04	0.23	0.13
3	4.98	3.20	1.07	14.18	0.76	0.95	0.05	0.23	0.15
3	4.95	3.64	1.21	7.41	0.45	0.94	0.06	0.23	0.17
6	4.95	3.60	0.60	58.00	3.48	0.95	0.05	0.23	0.17
6	4.87	4.54	0.76	37.37	2.82	0.93	0.07	0.23	0.21
6	4.80	5.47	0.91	23.79	2.17	0.92	0.08	0.22	0.26
6	4.73	6.41	1.07	14.18	1.51	0.90	0.10	0.22	0.30
6	4.66	7.29	1.21	7.41	0.90	0.89	0.11	0.22	0.34
9	4.81	5.40	0.60	58.00	5.22	0.92	0.08	0.22	0.25
9	4.70	6.80	0.76	37.37	4.24	0.90	0.10	0.22	0.32
9	4.58	8.21	0.91	23.79	3.25	0.88	0.12	0.21	0.38
9	4.47	9.61	1.07	14.18	2.27	0.85	0.15	0.21	0.45
9	4.37	10.93	1.21	7.41	1.35	0.83	0.17	0.20	0.51
12	4.66	7.20	0.60	58.00	6.96	0.89	0.11	0.22	0.34
12	4.52	9.07	0.76	37.37	5.65	0.86	0.14	0.21	0.42
12	4.37	10.94	0.91	23.79	4.34	0.83	0.17	0.20	0.51
12	4.22	12.82	1.07	14.18	3.03	0.81	0.19	0.20	0.60
12	4.08	14.57	1.21	7.41	1.80	0.78	0.22	0.19	0.68
15	4.52	9.00	0.60	58.00	8.70	0.86	0.14	0.21	0.42
15	4.34	11.34	0.76	37.37	7.06	0.83	0.17	0.20	0.53
15	4.15	13.68	0.91	23.79	5.42	0.79	0.21	0.19	0.64
15	3.97	16.02	1.07	14.18	3.79	0.76	0.24	0.19	0.75
15	3.21	18.21	1.21	7.41	2.25	0.61	0.28	0.15	0.85
18	4.38	10.80	0.60	58.00	10.44	0.84	0.16	0.20	0.50
18	4.16	13.61	0.76	37.37	8.47	0.79	0.21	0.19	0.64
18	3.94	16.42	0.91	23.79	6.51	0.75	0.25	0.18	0.77
18	2.20	19.22	1.07	14.18	4.54	0.42	0.29	0.10	0.90
18	0.00	21.43	1.19	8.40	3.00	0.00	0.32	0.00	1.00

Key

- N Number of on-line jobs running
- T1 Off-line throughput (phases/min)
- T2 On-line throughput (phases/min)
- W Mean response time for on-line jobs
- Q Mean number of on-line jobs awaiting response
- CU1 Processor use by off-line jobs (including overhead)
- CU2 Processor use by on-line jobs (including overhead)
- DU1 Disc use by off-line jobs
- DU2 Disc use by on-line jobs.

number of phases would be less than doubled because of the higher proportion of off-line work with a longer time per phase. 128K of core store would probably suffice for both processors, since one would be running mainly off-line jobs without much time sharing. This would represent a system where the disc and the processors were more evenly balanced for the given policy on the ratio of off-line to on-line work. More off-line work would soon appear to fill the extra capacity.

The next type of model uses the theory of Markov processes and again considers the system as a whole with jobs flowing through it, though at a variable rate and spending variable lengths of time in the different stages of the system. The system is considered as having a number of distinct states and is able to make transitions between the various states with the passage of time. To each transition from one state to another and from a state to itself is assigned a transition probability, which can be calculated from a knowledge of the system.

The system is called a Markov process if these transition probabilities depend only on the current state of the system and not on its past history. As mentioned earlier the exponential distribution possesses this memoryless property and most models assume this. Other distributions can be derived from the exponential by introducing notional extra stages of service, in series, parallel or a series-parallel combination, with different means. The use of the hyper-exponential distribution for comp. times has been mentioned already. Though these can represent a better approximation to the real distribution, and the methods of solving them are similar, they complicate the analysis and require more computer time for solution, or only permit numerical solution.

The result is a series of linear equations in the probabilities, which in certain cases can be solved analytically if there is sufficient symmetry. Because there may be a number of equations numerical solution of the other cases is difficult and requires recursion. However once numerical methods are necessary simulation with a more accurate model becomes attractive and numerical solution was not used in this work.

These have been studied by statisticians such as Jackson[30] and Koenisberg[38] in the last few years but there have only been two previous applications to computer systems. The simplest case is analogous to a machine minding problem with the consoles representing machines, the requests for processor time representing breakdowns, and the processor representing the engineer. The solution of this has been known for some time and this model was used by Scherr[54,55] at MIT to model CTSS and is derived in detail later. It has been developed further by Kleinrock.[35,36]

The other application was the numerical solution by J.L. Smith[58] of a system containing about 5000 states, using a recursive method developed by Wallace and Rosenberg. This method has also been used by J.M. Smith.[59] None of this work has distinguished between off-line and on-line jobs or dealt with requests for specific processing units.

All the service times used in the models are assumed to be exponentially distributed because of the convenient memoryless property this gives. If a random variable y with mean m is exponentially distributed its probability density function is given by:

$$PR(x \leq y \leq x+\Delta x) = \frac{e^{-x/m}}{m} \Delta x$$

Consider the distribution of the remainders of those y that are

greater than z. The probability of a random y being greater than z is:

$$PR\ (y > z) = \int_z^{\infty} \frac{e^{-x/m}}{m}\, dx = e^{-z/m}$$

Hence the conditional distribution of $(y - z)$ granted that it is positive or zero is given by:

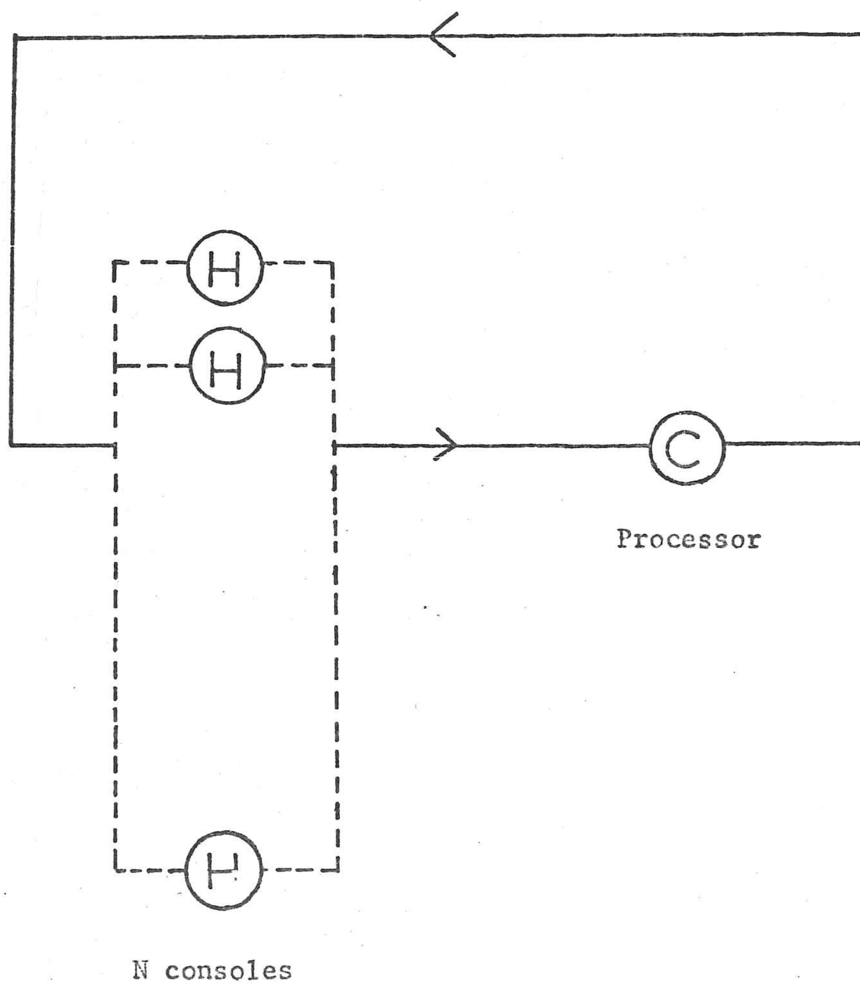
$$PR(x-z \leq y-z \leq x+\Delta x-z) = \frac{e^{-x/m}}{me^{-z/m}} \Delta x$$

where $0 \leq x-z \leq \infty$. If we change the variable to $x' = x - z$ this reduces once more to the exponential distribution. Thus however long a job has been in any stage its mean rate of leaving that stage is still $1/M$ where M is the mean effective service time for that stage. We shall consider first the simplest model, partly for illustration and partly because it is the only one that has previously been used for analytic solution when applied to computer systems. This is shown in Figure 4.3 and consists of N consoles with exponentially distributed thinking time H using one processor with mean computation time C. The notation is the same as before except that stages with exponentially distributed service times are represented by a circle with the mean time inside.

Because this is a closed system with a fixed number of jobs and because the memoryless property means that the amount of service they have had so far is irrelevant we concentrate our attention on the finite number of possible states of the system instead of on the individual jobs. The state (i,j) of the system can be described by the number of consoles thinking i and the number of jobs computing j. Since $i + j = N$ we only need one of these in principle,

Figure 4.3

Simplest Markov model of multi-access system



but the equations for the more general cases are more symmetrical and easier to follow if both are used. To each state we ascribe a probability P_{ij} of the system being in that state. We then consider the transition rates between various states.

We must also introduce the Heaviside unit function, to give it one of its names:

$$U(i) = \begin{cases} 1 & \text{if } i > 0 \\ 0 & \text{if } i = 0 \end{cases}$$

The mean rate of exit from the state where a job is thinking to the state where it is computing is $1/H$ and since there are i such jobs which all think in parallel the rate of transition from state (i,j) to state $(i-1,j+1)$ is i/H. We assume that no two jobs stop thinking exactly simultaneously and that "thinking" includes the processes of console input and output, the validity of the latter assumption has already been discussed. Similarly the rate of transition from state (i,j) to state $(i+1,j-1)$ is $U(j)/C$, since only one job can be computing at a time.

It does not in fact matter what the scheduling algorithm is, since this will merely reorder the service completions rather than change the rate of completion. Although it affects the distribution of waiting time conditional on processor time it does not affect its mean.

In equilibrium, which must be reached since this is a closed aperiodic system, the rate of transfer out of state (i,j) must be equal to the rate of transfer into it from other states for all possible (i,j). Hence we have:

$$\left(\frac{i}{H} + \frac{U(j)}{C}\right) P_{ij} = \frac{(i+1)}{H} P_{i+1,j-1} + \frac{1}{C} P_{i-1,j+1}$$

Rate of leaving (i,j) Rate of entering (i,j)

These $N + 1$ equations in $N + 1$ unknowns are homogeneous with zero determinant, since each term occurs both on a right-hand and left-hand side, and are soluble in several ways. One way is to note that the output of a queue with exponential input is also exponential, and apply the normal queueing theory solutions to each queue. Alternatively one can assume that the solution is of the form $P_{ij} = Q(i) \times R(j)$ and get the solutions by inspection or from the recurrence relations:

$$Q(i) = \frac{HQ(i-1)}{i} = \frac{H^i}{i!}$$

and $$R(j) = CR(j-1) = C^j$$

giving $$P_{ij} = \frac{FH^iC^j}{i!}$$

where F is a normalizing constant. This can be found from the fact that

$$\sum_{i=0}^{N} P_{i,N-i} = 1$$

Hence we have

$$P_{ij} = (H^i \times C^j)/(i! \times \sum_{k=0}^{N} (H^k \times C^{N-k}/k!))$$

Given the P_{ij} we can easily work out everything else such as the mean processor queue size $Q = \sum_{k=0}^{N} k\, P_{N-k,k}$, the processor utilisation $CU = 1 - P_{N,0}$ and the mean waiting time $W = \frac{QxC}{CU}$. This value for W comes from the mean queue size conditional on the stage being busy multiplied by the mean service time and will be used in the later queues we consider. The throughput can be expressed as $T = N/(W+H)$ or alternatively as $T = Q/W$ and hence we can get explicit relations

for Q and W in terms of CU namely

$$Q = N - (H \times CU)/C$$

$$W = N \times C/CU - H$$

Results from this model will not be given since they are not appropriate to a system with a heavy off-line load and a disc, and can be found elsewhere.[36,54]

The first extension we shall consider is the inclusion of the discs. This is complicated by the fact that there are effectively two of them. If we consider the model in Figure 4.4 with only on-line jobs again for simplicity and i, j and k jobs thinking, waiting for the discs, and waiting for the processor respectively, and M discs the rate of exit from state (i,j,k) to state (i,j-1,k+1) is $\frac{MIN(j,M)}{D}$, the numerator giving the number of disc units in use. We thus get the following set of equations:

$$\left(\frac{i}{H} + \frac{MIN(j,M)}{D} + \frac{U(k)}{C}\right) P_{ijk} = \frac{i+1}{H} + P_{i+1,j-1,k} + \frac{MIN(j+1,M)}{D} P_{i,j+1,k-1} + \frac{1}{C} P_{i-1,j,k+1}$$

where terms with any suffix negative are ignored. To simplify subsequent equations we use the notation

$$P_k^j = P_{i,j+1,k-1} \quad \text{etc.}$$

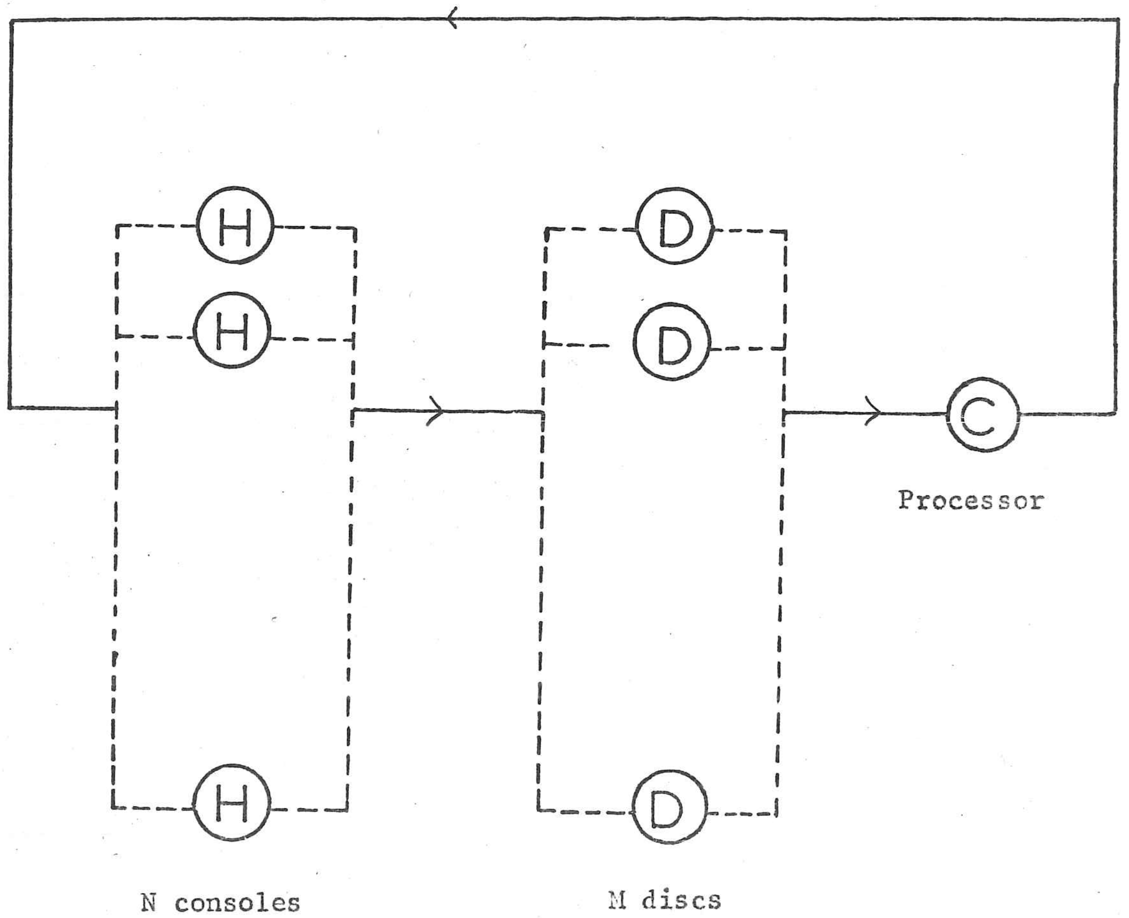
The upper subscript is increased by one and the lower is decreased by one, all others remaining the same. Hence the equation just given could be written:

$$\left(\frac{i}{H} + \frac{MIN(j,M)}{D} + \frac{U(k)}{C}\right) P_{ijk} = \frac{i+1}{H} P_j^i + \frac{MIN(j+1,M)}{D} P_k^j + \frac{1}{C} P_i^k$$

Proceeding to solve these in the same way we get

Figure 4.4

Markov model with several discs



$$P_{ijk} = \frac{H^i D^j C^k F}{i!j!} \qquad j \leq M$$

$$P_{ijk} = \frac{H^i D^j C^k F}{i!M!M^{(j-M)}} \qquad j \geq M$$

$$\sum_{i+j+k=n} P_{ijk} = 1$$

the factor F being derived from the last equation in the same way. This can obviously be extended to deal with multiple processors.

A closer approximation to the truth is to assume that disc activity is spread out into several bursts during the phase and see what difference that makes. In the model in Figure 4.5 we assume that each phase starts and ends with a disc access and also has disc accesses in the middle of it. After each disc access the probability of returning for more computation is q and the probability of finishing is p where $p + q = 1$. The mean number of disc accesses per phase is $1/p$ and the mean number of processor bursts is q/p. Thus we have for the new disc time per burst $D' = p \times D$ and the new processor time per burst $C' = p \times C/q$. Typical values of p and q would be $p = 0.125$, $q = 0.875$ corresponding to 8 disc bursts and 7 processor bursts per phase. It must be remembered that this includes program loading, backing store transfers, stream handling, filing and supervisor activities. The new equations are:

$$\left(\frac{i}{H} + \frac{MIN(j,2)}{D'} + \frac{U(k)}{C'}\right) P_{ijk} = \frac{i+1}{H} P^i_j + p\,\frac{MIN(j+1,2)}{D'} P^j_i$$
$$+ q\,\frac{MIN(j+1,2)}{D'} P^j_k + \frac{1}{C'} P^k_j$$

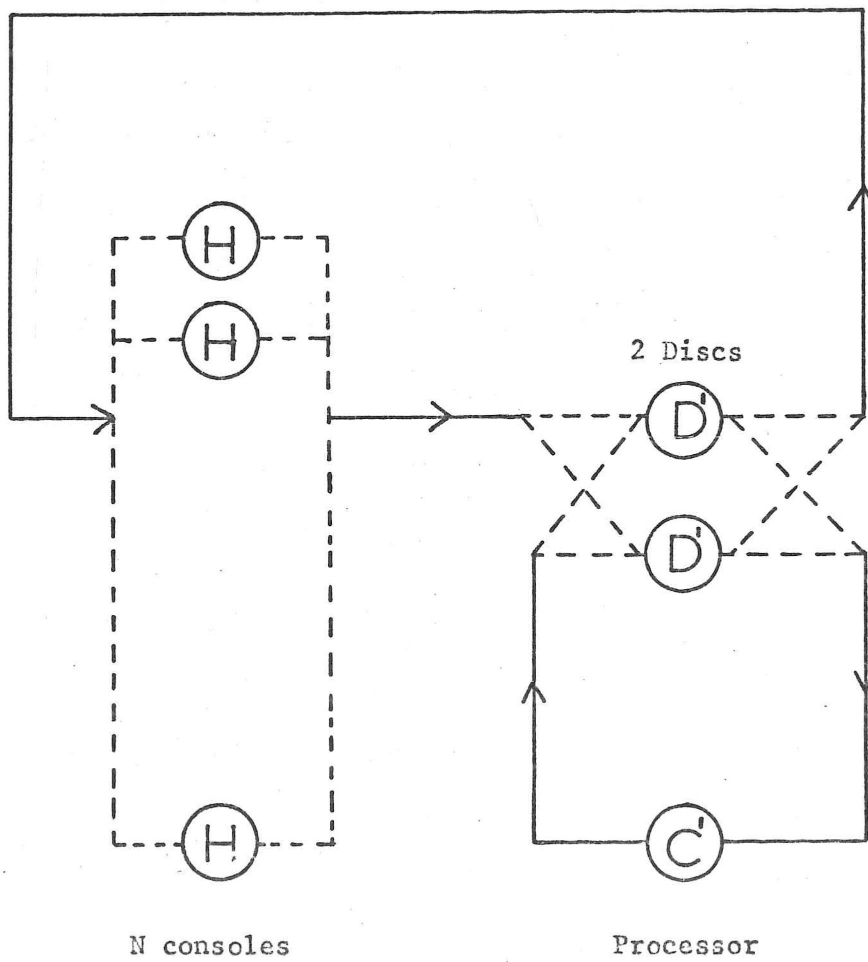
These give by inspection:

$$P_{ijk} = \frac{(pH)^i}{i!}\,\frac{(D')^j}{j!}\,(qC')^k\,F' \qquad j \leq 2$$

$$P_{ijk} = \frac{(pH)^i}{i!}\,\frac{(D')^j}{2^{j-1}}(qC')^k\,F' \qquad j \geq 2$$

Figure 4.5

Markov model with bursts of processing and disc activity



If we substitute in the values of $D' = p \times D$ and $C' = p \times C/q$ we find that the equations are the same as the original ones provided that $p^{(i+j+k)} \times F' = F$. But $i + j + k = N$ always and since F and F' are just normalising factors it is the ratios between the P_{ijk} that matter. Hence the assumption that all the disc activity is concentrated at the beginning of each phase makes no difference, and will be used in future .

The next problem is how to include the off-line work and apportion the processor and disc time between the two. Here the equations resist solution for the general case. However one of the most natural algorithms, namely if there are k jobs of one type and m jobs of another type waiting for service at a given stage, then the first type get a fraction $k/(k+m)$ and the second type a fraction $m/(k+m)$ of the server's time, is soluble. We can apply this algorithm both to the disc and the processor.

In order to apply this algorithm we must assume a fixed total (N1) of off-line jobs with ℓ waiting for the disc and m waiting for the processor, and $\ell + m = N1$. This corresponds reasonably well to reality where the number is effectively limited by the size of the core store and the scheduling policy of the supervisor. For a first come first served queue such as for the disc, or a round robin system, which is almost the case for the processor, this algorithm for time apportioning is the obvious one to use. This gives the following equations:

$$\left(\frac{i}{H} + \frac{\mathrm{MIN}(j+1,z)}{D} + \frac{k}{(k+m)C} + \frac{m}{(k+m)C}\right)P_{ijk\ell m} = \frac{i+1}{H}P^{i}_{j} + \frac{(j+1)\mathrm{MIN}(j+\ell+1,2)}{(j+\ell+1)D}P^{j}_{k}$$

$$+ \frac{(k+1)}{(k+m+1)C2}P^{k}_{i} + \frac{(\ell+1)\mathrm{MIN}(j+\ell+1,2)}{(j+\ell+1)D}P^{\ell}_{m} + \frac{m+1}{(k+m+1)C1}P^{m}_{\ell}$$

The solution of these is

$$P_{ijk\ell m} = \frac{H^i}{i!} \quad \frac{D^{j+\ell}}{j!\ell!} \quad C2^k C1^m \quad \frac{(k+m)!}{k!m!} \; F \qquad j+\ell \leq 2$$

$$P_{ijk\ell m} = \frac{H^i}{i!} \quad \frac{D^{j+\ell}(j+\ell)!}{j!\ell!2^{j+\ell-1}} \; C2^k C1^m \; \frac{(k+m)!}{k!m!} \; F \qquad j+\ell \geq 2$$

where N1 and N2 are the total numbers of off-line and on-line jobs respectively and C1 and C2 are their mean processor times. F is found from the normalising condition in the usual way.

$$\sum_{i+j+k\ =\ N2} \quad \sum_{\ell+m\ =\ N1} P_{ijk\ell m} = 1$$

In such models the actual values of the probabilities are not usually of interest, and would occupy too much paper in any case, since the number of possible states is in the thousands for large N1 and N2. Instead we are interested primarily in the derived quantities like queue sizes and waiting time and item usage and throughput, the method of calculating which was described earlier for the simple case. In this case the situation is much the same only the sum must be taken over a larger range of possibilities.

The results for this model are given in Table 4.3 and Figure 4.6 for the one processor case and Table 4.4 or the two processor case, both processors being of the same speed as the current one, as before. The first thing to notice about these results is the remarkable measure of agreement with the non-statistical model. Although the latter did not include the number of off-line jobs we can compare results with equal values of the throughputs. One might have thought that the statistical variation of times would have changed the performance characteristics of the system, but this appears not to be so.

Figure 4.6

Mean on-line waiting time against off-line throughput for Markov model

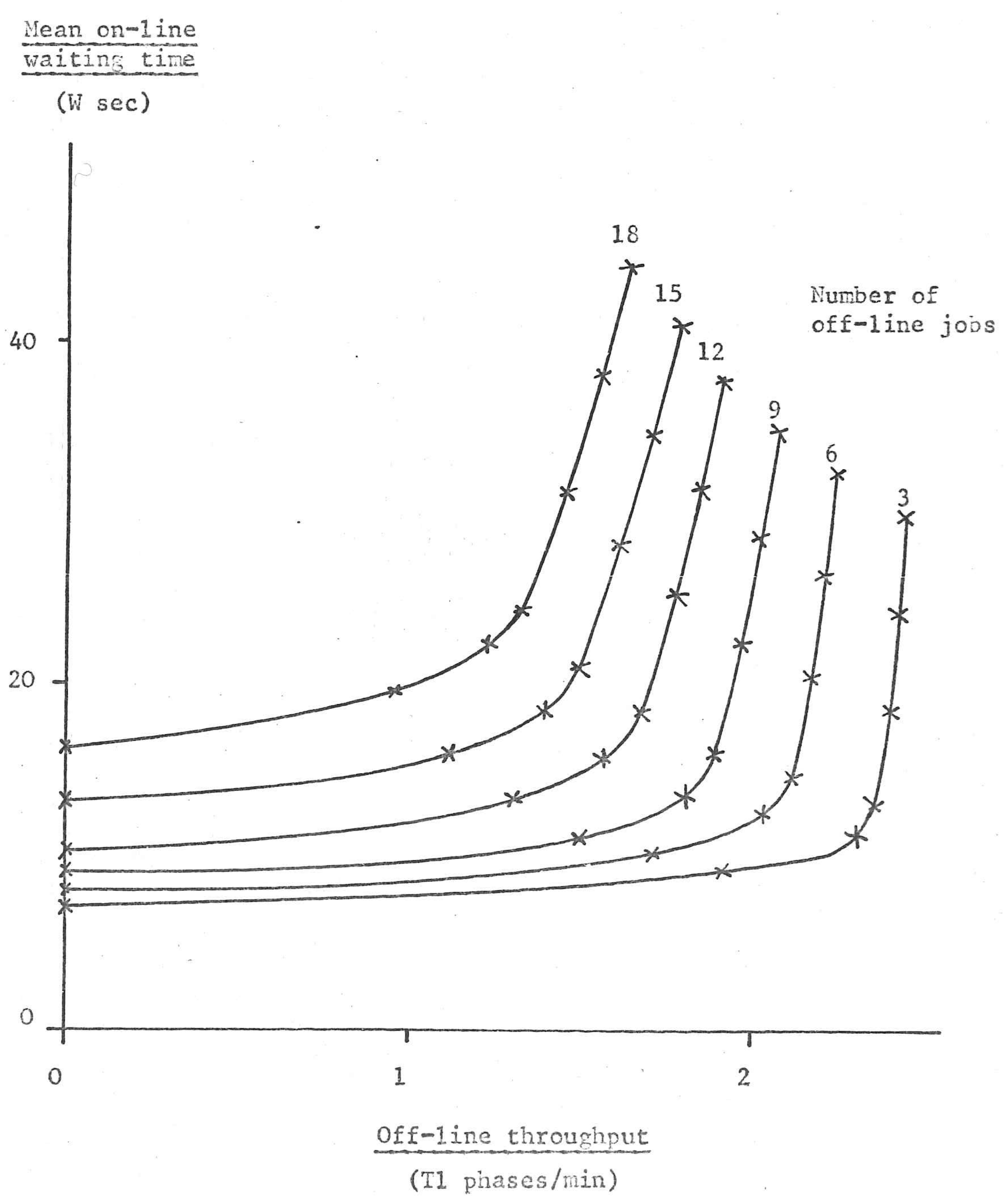


Table 4.3

Markov model of Cambridge system

N2	N1	T1	T2	T2/N2	W	Q	CU1	CU2	DU1	DU2
3	0	0.00	3.63	1.21	7.59	0.46	0.00	0.11	0.00	0.17
3	3	2.35	3.27	1.09	13.13	0.71	0.90	0.10	0.11	0.15
3	6	2.38	2.96	0.99	18.84	0.93	0.91	0.09	0.11	0.14
3	9	2.40	2.71	0.90	24.51	1.11	0.92	0.08	0.11	0.13
3	12	2.42	2.50	0.83	30.14	1.25	0.92	0.08	0.11	0.12
6	0	0.00	7.18	1.20	8.16	0.98	0.00	0.22	0.00	0.33
6	3	2.11	6.38	1.06	14.39	1.53	0.80	0.19	0.10	0.30
6	6	2.16	5.76	0.96	20.50	1.97	0.83	0.17	0.10	0.27
6	9	2.20	5.26	0.88	26.49	2.32	0.84	0.16	0.10	0.25
6	12	2.23	4.84	0.81	32.39	2.61	0.85	0.15	0.10	0.23
9	0	0.00	10.56	1.17	9.15	1.61	0.00	0.32	0.00	0.49
9	3	1.87	9.30	1.03	16.06	2.49	0.72	0.28	0.09	0.43
9	6	1.96	8.37	0.93	22.54	3.14	0.75	0.25	0.09	0.39
9	9	2.01	7.63	0.85	28.82	3.66	0.77	0.23	0.09	0.36
9	12	2.06	7.02	0.78	34.95	4.09	0.79	0.21	0.10	0.33
12	0	0.00	13.66	1.14	10.70	2.44	0.00	0.41	0.00	0.64
12	3	1.66	11.95	1.00	18.23	3.63	0.64	0.36	0.08	0.56
12	6	1.77	10.75	0.90	25.00	4.48	0.68	0.32	0.08	0.50
12	9	1.84	9.80	0.82	31.50	5.14	0.70	0.30	0.09	0.46
12	12	1.90	9.02	0.75	37.82	5.69	0.73	0.27	0.09	0.42
15	0	0.00	16.36	1.09	13.02	3.55	0.00	0.49	0.00	0.76
15	3	1.48	14.28	0.95	21.00	5.00	0.56	0.43	0.07	0.67
15	6	1.60	12.87	0.86	27.92	5.99	0.61	0.39	0.07	0.60
15	9	1.69	11.75	0.78	34.57	6.77	0.64	0.36	0.08	0.55
15	12	1.76	10.84	0.72	41.02	7.41	0.67	0.33	0.08	0.51
18	0	0.00	18.49	1.03	16.40	5.05	0.00	0.56	0.00	0.86
18	3	1.32	16.23	0.90	24.52	6.64	0.50	0.49	0.06	0.76
18	6	1.45	14.72	0.82	31.37	7.70	0.56	0.44	0.07	0.69
18	9	1.55	13.49	0.75	38.04	8.55	0.59	0.41	0.07	0.63
18	12	1.63	12.48	0.69	44.54	9.26	0.62	0.38	0.08	0.58

N1 Number of off-line jobs running
N2 Number of on-line jobs running
T1 Off-line throughput (phases/min)
T2 On-line throughput (phases/min)
W Mean response time for on-line jobs
Q Mean number of on-line jobs awaiting response
CU1 Processor use by off-line jobs (including overhead)
CU2 Processor use by on-line jobs (including overhead)
DU1 Disc use by off-line jobs
DU2 Disc use by on-line jobs

Table 4.4

Markov model of two processor system

N2	N1	T1	T2	T2/N2	W	Q	CU1	CU2	DU1	DU2
3	0	0.00	3.64	1.21	7.45	0.45	0.00	0.05	0.00	0.17
3	3	4.71	3.50	1.17	9.49	0.55	0.90	0.05	0.22	0.16
3	6	4.97	3.31	1.10	12.35	0.68	0.95	0.05	0.23	0.15
3	9	4.98	3.15	1.05	15.13	0.79	0.95	0.05	0.23	0.15
3	12	5.00	3.00	1.00	17.91	0.90	0.95	0.05	0.23	0.14
6	0	0.00	7.23	1.21	7.79	0.94	0.00	0.11	0.00	0.34
6	3	4.42	6.87	1.14	10.43	1.19	0.85	0.10	0.21	0.32
6	6	4.71	6.50	1.08	13.42	1.45	0.90	0.10	0.22	0.30
6	9	4.74	6.18	1.03	16.27	1.68	0.91	0.09	0.22	0.29
6	12	4.77	5.89	0.98	19.11	1.88	0.91	0.09	0.22	0.27
9	0	0.00	10.69	1.19	8.52	1.52	0.00	0.16	0.00	0.50
9	3	4.13	10.03	1.11	11.86	1.98	0.79	0.15	0.19	0.47
9	6	4.47	9.48	1.05	14.97	2.37	0.85	0.14	0.21	0.44
9	9	4.52	9.02	1.00	17.85	2.68	0.86	0.14	0.21	0.42
9	12	4.55	8.61	0.96	20.70	2.97	0.87	0.13	0.21	0.40
12	0	0.00	13.90	1.16	9.80	2.27	0.00	0.21	0.00	0.65
12	3	3.83	12.85	1.07	14.01	3.00	0.73	0.19	0.18	0.60
12	6	4.25	12.16	1.01	17.23	3.49	0.81	0.18	0.20	0.57
12	9	4.31	11.61	0.97	20.03	3.87	0.82	0.18	0.20	0.54
12	12	4.36	11.11	0.93	22.82	4.22	0.83	0.17	0.20	0.52
15	0	0.00	16.71	1.11	11.87	3.31	0.00	0.25	0.00	0.78
15	3	3.53	15.20	1.01	17.21	4.36	0.68	0.23	0.16	0.71
15	6	4.05	14.39	0.96	20.52	4.92	0.77	0.22	0.19	0.67
15	9	4.14	13.82	0.92	23.13	5.33	0.79	0.21	0.19	0.64
15	12	4.18	13.29	0.89	25.71	5.70	0.80	0.20	0.20	0.62
18	0	0.00	18.90	1.05	15.13	4.77	0.00	0.29	0.00	0.88
18	3	3.23	16.93	0.94	21.80	6.15	0.62	0.26	0.15	0.79
18	6	3.89	16.05	0.89	25.27	6.76	0.74	0.24	0.18	0.75
18	9	4.00	15.54	0.86	27.51	7.12	0.76	0.23	0.19	0.73
18	12	4.04	15.06	0.84	29.71	7.46	0.77	0.23	0.19	0.70

N1 Number of off-line jobs running
N2 Number of on-line jobs running
T1 Off-line throughput (phases/min)
T2 On-line throughput (phases/min)
W Mean response time for on-line jobs
Q Mean number of on-line jobs awaiting response
CU1 Processor use by off-line jobs (including overhead)
CU2 Processor use by on-line jobs (including overhead)
DU1 Disc use by off-line jobs
DU2 Disc use by on-line jobs

The values for N1 = 0 give some idea of the extent that the response time and on-line throughput are degraded by clashes of requests, for small N2 we approach the theoretical optimum, but by the time we get 18 consoles the response time has doubled, the bottleneck here being the disc which is in use 86% of the time. Once the off-line work is included the waiting time is much the same for a given throughput per console, this is because the scheduling algorithm gives priority for on-line jobs when there are a lot waiting.

It can be seen that 3 off-line jobs are sufficient to keep one processor busy over 99% of the time, however many on-line jobs there are. The only reason one might want more is to give a better turn-round for short jobs while long jobs are running, or to encourage console users to do some of their work by creating off-line jobs, or because the scheduling algorithm requires more off-line jobs in order to give them more time. The disadvantage of having several off-line jobs going is the amount of core store they use, and effectively deny to on-line jobs. The present scheduling algorithm could easily be altered, for example by halving the amount of comp. time used in each second by off-line jobs before making the comparison to see which one should be demoted. In that way three off-line jobs could use as much time as six would have done, with a possible improvement in on-line response time in the worst cases because waiting for core store is reduced.

If another processor were added then more off-line jobs would be needed to keep the processor as busy and absorb the extra time, thus this change would probably not be desirable without adding a drum for swapping or more core store. A compromise solution to this problem

which has been used elsewhere and has proved quite satisfactory is to have a small computer to deal with the more basic needs of the on-line jobs, such as input, editing and filing, and only use the main rpocessor for program runs. From the tables given earlier it can be seen that these operations represent almost half the total amount of computer time used by on-line jobs, and thus this would give a substantial improvement, the limiting factor then being the discs.

As mentioned earlier the disc units can be considered as being between the two extremes of either having a request met by any unit that is free or having each request for a specific unit. The former case which is the easiest has already been dealt with and the latter will now be considered. We solve this problem by introducing new variables to describe the state, and instead of considering the total number of off-line and on-line jobs waiting for the disc, ℓ and j respectively in the previous notation, we consider the number waiting for disc unit one (ℓ_1 and j_1) and the number waiting the disc unit two (ℓ_2 and j_2) separately. Using the previous notation for the other variables the equations become:

$$\left(\frac{i}{H} + \frac{U(j_1+\ell_1)}{D} + \frac{U(j_2+\ell_2)}{D} + \frac{k}{(k+m)C_2} + \frac{m}{(k+m)C_1}\right)P_{i,j_1,j_2,k,\ell_1,\ell_2,m}$$

$$= \frac{i+1}{H}\left(\tfrac{1}{2}\left(P^{i}_{j_1} + P^{i}_{j_2}\right)\right) + \frac{j_1+1}{(j_1+\ell_1+1)D}\,P^{j_1}_{k} + \frac{j_2+1}{(j_2+\ell_2+1)D}\,P^{j_2}_{k} + \frac{k+1}{C_2(k+m+1)}\,P^{k}_{i}$$

$$+ \frac{\ell_1+1}{(j_1+\ell_1+1)D}\,P^{\ell_1}_{m} + \frac{\ell_2+1}{(j_2+\ell_2+1)D}\,P^{\ell_2}_{m} + \frac{m+1}{(k+m+1)C_1}\left(\tfrac{1}{2}\left(P^{m}_{\ell_1} + P^{m}_{\ell_2}\right)\right)$$

The solution is

$$P_{ij_1j_2k\ell_1\ell_2m} = \frac{H^{i}(j_1+\ell_1)!\,(j_2+\ell_2)!\,D^{j_1+j_2+\ell_1+\ell_2}\,(k+m)!\,C_2^{\,k}C_1^{\,m}}{i!\,j_1!\,\ell_1!\,j_2!\,\ell_2!\,2^{j_1+j_2+\ell_1+\ell_2}\,k!\,m!}$$

The results are given in Table 4.5. As might be expected they show a degradation in performance, which is small at first but increases as the number of consoles increases. However it is sufficiently small in magnitude to be ignored when doing simulations.

The main interest in this model lies not so much in the results but in the method of extension of the previous model and the method used for numerical computation of the answers. This type of extension can be applied also to the case where there are two types of processor, one used for input-output, filing and editing and one for serious calculation. There is no need for either the two service times or the probabilities of a request being for a given unit to be the same.

When the number of states is over 10,000 as in this model, straightforward techniques of computing each term as required and adding the sum together are very time consuming. The basic method is to store as much as possible in arrays. The factorials and the powers are obvious candidates for this since they can be generated conveniently from the previous one and expressions like $H^i/i!$ can be computed all at once. Since we are not interested in the distribution between the two discs except as far as disc usage is concerned we can calculate a function:

$$F(j,\ell) = \sum_{j_1+j_2=j} \sum_{\ell_1+\ell_2=\ell} \frac{(j_1+\ell_1)!(j_2+\ell_2)!}{j_1!\ell_1!j_2!\ell_2!}$$

Taking advantage of the symmetry between j and ℓ to halve the work and calculating the corresponding on-line and off-line disc usage simultaneously with $F(j,\ell)$ we can reduce the work so much that it only takes 20 seconds including compilation for the results in Table 4.5.

Table 4.5

Markov model of Cambridge system with requests for specific discs

N2	N1	T1	T2	T2/N2	W	Q	CU1	CU2	DU1	DU2
3	0	0.00	3.58	1.19	8.21	0.49	0.00	0.11	0.00	0.17
3	3	2.35	3.20	1.07	14.26	0.76	0.90	0.10	0.11	0.15
3	6	2.39	2.91	0.97	19.93	0.97	0.91	0.09	0.11	0.14
3	9	2.41	2.66	0.89	25.55	1.13	0.92	0.08	0.11	0.12
3	12	2.42	2.46	0.82	31.15	1.28	0.93	0.07	0.11	0.11
6	0	0.00	6.96	1.16	9.72	1.13	0.00	0.21	0.00	0.32
6	3	2.11	6.18	1.03	16.28	1.68	0.81	0.19	0.10	0.29
6	6	2.17	5.60	0.93	22.26	2.08	0.83	0.17	0.10	0.26
6	9	2.21	5.13	0.86	28.14	2.41	0.84	0.16	0.10	0.24
6	12	2.24	4.74	0.79	33.95	2.68	0.86	0.14	0.10	0.22
9	0	0.00	10.06	1.12	11.70	1.96	0.00	0.30	0.00	0.47
9	3	1.90	8.89	0.99	18.76	2.78	0.72	0.27	0.09	0.41
9	6	1.98	8.06	0.90	24.98	3.36	0.76	0.24	0.09	0.38
9	9	2.03	7.39	0.82	31.06	3.83	0.78	0.22	0.09	0.34
9	12	2.08	6.83	0.76	37.04	4.22	0.79	0.21	0.10	0.32
12	0	0.00	12.78	1.07	14.33	3.05	0.00	0.39	0.00	0.60
12	3	1.70	11.29	0.94	21.78	4.10	0.65	0.34	0.08	0.53
12	6	1.81	10.26	0.86	28.15	4.82	0.69	0.31	0.08	0.48
12	9	1.87	9.43	0.79	34.36	5.40	0.72	0.28	0.09	0.44
12	12	1.93	8.73	0.73	40.45	5.89	0.74	0.26	0.09	0.41
15	0	0.00	15.06	1.00	17.77	4.46	0.00	0.45	0.00	0.70
15	3	1.54	13.34	0.89	25.46	5.66	0.59	0.40	0.07	0.62
15	6	1.65	12.19	0.81	31.82	6.47	0.63	0.37	0.08	0.57
15	9	1.73	11.24	0.75	38.05	7.13	0.66	0.34	0.08	0.52
15	12	1.79	10.44	0.70	44.19	7.69	0.68	0.32	0.08	0.49
18	0	0.00	16.83	0.93	22.18	6.22	0.00	0.51	0.00	0.79
18	3	1.40	15.03	0.83	29.87	7.48	0.53	0.45	0.07	0.70
18	6	1.52	13.84	0.77	36.03	8.31	0.58	0.42	0.07	0.65
18	9	1.60	12.83	0.71	42.17	9.02	0.61	0.39	0.07	0.60
18	12	1.67	11.96	0.66	48.28	9.63	0.64	0.36	0.08	0.56

N1 Number of off-line jobs running
N2 Number of on-line jobs running
T1 Off-line throughput (phases/min)
T2 On-line throughput (phases/min)
W Mean response time for on-line jobs
Q Mean number of on-line jobs awaiting response
CU1 Processor use by off-line jobs (including overhead)
CU2 Processor use by on-line jobs (including overhead)
DU1 Disc use by off-line jobs
DU2 Disc use by on-line jobs

Table 4.5

Markov model of Cambridge system with requests for specific discs

N2	N1	T1	T2	T2/N2	W	Q	CU1	CU2	DU1	DU2
3	0	0.00	3.58	1.19	8.21	0.49	0.00	0.11	0.00	0.17
3	3	2.35	3.20	1.07	14.26	0.76	0.90	0.10	0.11	0.15
3	6	2.39	2.91	0.97	19.93	0.97	0.91	0.09	0.11	0.14
3	9	2.41	2.66	0.89	25.55	1.13	0.92	0.08	0.11	0.12
3	12	2.42	2.46	0.82	31.15	1.28	0.93	0.07	0.11	0.11
6	0	0.00	6.96	1.16	9.72	1.13	0.00	0.21	0.00	0.32
6	3	2.11	6.18	1.03	16.28	1.68	0.81	0.19	0.10	0.29
6	6	2.17	5.60	0.93	22.26	2.08	0.83	0.17	0.10	0.26
6	9	2.21	5.13	0.86	28.14	2.41	0.84	0.16	0.10	0.24
6	12	2.24	4.74	0.79	33.95	2.68	0.86	0.14	0.10	0.22
9	0	0.00	10.06	1.12	11.70	1.96	0.00	0.30	0.00	0.47
9	3	1.90	8.89	0.99	18.76	2.78	0.72	0.27	0.09	0.41
9	6	1.98	8.06	0.90	24.98	3.36	0.76	0.24	0.09	0.38
9	9	2.03	7.39	0.82	31.06	3.83	0.78	0.22	0.09	0.34
9	12	2.08	6.83	0.76	37.04	4.22	0.79	0.21	0.10	0.32
12	0	0.00	12.78	1.07	14.33	3.05	0.00	0.39	0.00	0.60
12	3	1.70	11.29	0.94	21.78	4.10	0.65	0.34	0.08	0.53
12	6	1.81	10.26	0.86	28.15	4.82	0.69	0.31	0.08	0.48
12	9	1.87	9.43	0.79	34.36	5.40	0.72	0.28	0.09	0.44
12	12	1.93	8.73	0.73	40.45	5.89	0.74	0.26	0.09	0.41
15	0	0.00	15.06	1.00	17.77	4.46	0.00	0.45	0.00	0.70
15	3	1.54	13.34	0.89	25.46	5.66	0.59	0.40	0.07	0.62
15	6	1.65	12.19	0.81	31.82	6.47	0.63	0.37	0.08	0.57
15	9	1.73	11.24	0.75	38.05	7.13	0.66	0.34	0.08	0.52
15	12	1.79	10.44	0.70	44.19	7.69	0.68	0.32	0.08	0.49
18	0	0.00	16.83	0.93	22.18	6.22	0.00	0.51	0.00	0.79
18	3	1.40	15.03	0.83	29.87	7.48	0.53	0.45	0.07	0.70
18	6	1.52	13.84	0.77	36.03	8.31	0.58	0.42	0.07	0.65
18	9	1.60	12.83	0.71	42.17	9.02	0.61	0.39	0.07	0.60
18	12	1.67	11.96	0.66	48.28	9.63	0.64	0.36	0.08	0.56

N1 Number of off-line jobs running
N2 Number of on-line jobs running
T1 Off-line throughput (phases/min)
T2 On-line throughput (phases/min)
W Mean response time for on-line jobs
Q Mean number of on-line jobs awaiting response
CU1 Processor use by off-line jobs (including overhead)
CU2 Processor use by on-line jobs (including overhead)
DU1 Disc use by off-line jobs
DU2 Disc use by on-line jobs

A deficiency of these types of models is that one can only get the mean of the waiting time rather than the distribution, except for simple scheduling algorithms, and no account is taken of the delays waiting for store. When a good paging system is used or the store is big enough for the central processor never to be idle waiting for it this is a satisfactory assumption, as the main difference it makes is to the distribution of waiting times, for example by several jobs being held up by a large one. The mean waiting time is not affected. To study the distribution we have to use another type of model or simulation.

The next type of model concentrates on individual jobs arriving at the various stages of service and follows them through according to the scheduling algorithm. Examples of this are Coffman[9], Coffman and Kleinrock[10], Fife[15] and Gaver[19]. Fife's work is particularly interesting since he considers a three queue system and evaluates systematically all the scheduling policies to see which is best according to certain criteria. Although this is only possible for a limited range of scheduling policies it nevertheless represents the only attempt to derive them systematically. Such a study would take much too long using simulation.

For more complicated scheduling systems such as are used in the Cambridge system this sort of analysis becomes impossible, and no new work was done in this field. However, another analytical model was developed to give some insight into the problems of swapping. This is based on the concept of a pipeline, suggested by Wilkes[65], through which jobs pass on the way to a swapping region where they are swapped in and out of core. When in the pipeline the jobs remain in core, and

in the original model they were in the pipeline for a fixed real time. In this model however they remain in the pipeline (i.e. in core) until they have had a certain quantum of processor time. The object in both models is the same, to avoid swapping out jobs that have just arrived. This variation ensures that short running jobs will not get swapped out just because there is a long queue and they are held up. To simplify the analysis we assume an infinite supply of jobs for the pipeline, as we are concentrating primarily on what happens in the swapping region.

In order to develop equations for this mode we need some notation as follows:

C mean processor time (exponentially distributed)

T quantum of processor time given to each job before being transferred to swapping region

P proportion of processor time given to jobs in pipeline

F proportion of jobs with processor time T

S proportion of processor time given to jobs in swapping region when there are any

R_0 probability of no jobs in swapping region

R_1 probability of 1 job in swapping region

⋮

Q mean number of jobs in swapping region

W mean waiting time in swapping region

$$\text{Mean processor time in pipeline} = \int_0^T e^{-x/C}\frac{x}{C}\,dx + \int_T^\infty e^{-x/C}\frac{T}{C}\,dx$$

$$= C - Ce^{-T/C} - Te^{-T/C} + Te^{-T/C} = C(1-e^{-T/C})$$

$$\left(\text{since } \int_a^b \frac{x}{C}e^{-x/C} = \left[-xe^{-x/C}\right]_a^b - \int_a^b -e^{-x/C}\,dx = \left[-(C+x)e^{-x/C}\right]_a^b\right)$$

$$\text{Rate of exit from pipeline} = \frac{P}{C(1-e^{-T/C})}$$

A fraction F of jobs leaving the pipeline will be doing so because their quantum has expired and will enter the swapping region

$$F = \int_T^\infty \frac{1}{C} e^{-x/C} dx = e^{-T/C}$$

This relationship will be used as a shorthand hereafter.

Rate of entry to swapping region = Rate of exit from pipeline owing to quantum expiring $= \frac{PF}{C(1-F)}$

Rate of exit from swapping region when jobs present there $= \frac{S}{C}$

(since mean residual length is C from properties of exponential distribution).

Now P depends on whether there are jobs in the swapping region or not

$$P = 1 - S \quad \text{if there are}$$

$$P = 1 \quad \text{if not}$$

Hence for the swapping region we have a queue dependent arrival rate.

Let r_i be the probability that there are i jobs in the queue

a_i be the arrival rate when there are i jobs in the queue

s_i be the service rate when there are i jobs in the queue

Then from queueing theory

$$r_i = \frac{a_o a_1 \ldots a_{i-1} r_o}{s_1 s_2 \ldots s_i}$$

$$r_o = \frac{1}{1 + \frac{a_o}{s_1} + \frac{a_o a_1}{s_1 s_2} \ldots}$$

For stability the denominator of r_o must be finite

$$s_1 = s_2 \ldots = \frac{S}{C}$$

$$a_o = \frac{F}{C(1-F)} \quad a_1 = a_2 = (1-S)a_o$$

$$\text{Let} \quad Y = \frac{r_{i+1}}{r_i} \quad \text{for} \quad i \geq 1$$

For convergence the ratio of successive terms Y must be less than unity

$$Y = (1-S) \times \frac{F}{C(1-F)} / \frac{S}{C} = \frac{F(1-S)}{S(1-F)} < 1$$

$$F - SF < S - SF \qquad \underline{\underline{S > F}}$$

$$r_o = \frac{1}{1 + \frac{F}{S(1-F)} \times \frac{1}{1-Y}}$$

$$\frac{1}{1-Y} = \frac{S(1-F)}{S(1-F) - (1-S)F} = \frac{S(1-F)}{S-F}$$

$$r_o = \frac{1}{1 + \frac{F}{S-F}} = \frac{S-F}{S}$$

$$r_i = \frac{Y^i r_o}{1-S} = \frac{(S-F)Y^i}{S(1-S)}$$

$$r_1 = \frac{(S-F)F}{(1-F)S^2}$$

Mean queue size $Q = \Sigma i r_i$

But $\Sigma iY^i = Y \frac{d}{dy} \Sigma Y^i = \frac{Y}{(1-Y)^2}$

$$Q = \frac{(S-F)}{S(1-S)} \times \frac{F(1-S)}{S(1-F)} \times \frac{S^2(1-F)^2}{(S-F)^2} = \frac{F(1-F)}{S-F}$$

Average $P = r_o + (1-S)(1-r_o) = \frac{S-F}{S} + \frac{(1-S)F}{S} = 1 - F$

A fraction F of jobs enter the swapping region with mean residual length C (from properties of exponential distribution).

Service rate for swapping region $= \frac{F}{C} = \frac{Q}{W}$ from queueing theory (43)

$$\therefore \quad W = \frac{C(1-F)}{S-F}$$

Table 4.6 gives some of these quantities for some typical values for C = 8 seconds.

Table 4.6

Pipeline swapping model for C = 8 sec

T	P	F	S	R_o	R_1	Y	Q	W	C+WF
1	0.12	0.88	1.00	0.12	0.88	0.00	0.88	8.00	16.04
1	0.12	0.88	0.98	0.09	0.73	0.19	1.12	10.42	17.16
1	0.12	0.88	0.95	0.07	0.56	0.40	1.54	14.66	20.88
1	0.12	0.88	0.92	0.05	0.37	0.61	2.44	23.91	29.10
1	0.12	0.88	0.90	0.02	0.16	0.83	5.92	59.67	60.50
2	0.22	0.78	1.00	0.22	0.78	0.00	0.78	8.00	14.25
2	0.22	0.78	0.95	0.18	0.67	0.19	1.01	10.88	16.44
2	0.22	0.78	0.90	0.13	0.53	0.39	1.42	16.22	20.68
2	0.22	0.78	0.85	0.08	0.35	0.62	2.42	29.24	30.85
2	0.22	0.78	0.80	0.03	0.12	0.88	8.13	104.34	89.50
4	0.39	0.61	1.00	0.39	0.61	0.00	0.61	8.00	13.68
4	0.39	0.61	0.90	0.33	0.56	0.17	0.81	11.92	15.28
4	0.39	0.61	0.80	0.24	0.47	0.39	1.23	20.34	20.62
4	0.39	0.61	0.70	0.13	0.29	0.66	2.55	48.11	37.40
8	0.63	0.37	1.00	0.63	0.37	0.00	0.37	8.00	10.96
8	0.63	0.37	0.80	0.54	0.39	0.15	0.54	14.63	13.43
8	0.63	0.37	0.60	0.39	0.38	0.39	1.00	36.31	21.45
8	0.63	0.37	0.40	0.08	0.12	0.87	7.24	393.59	114.00
16	0.86	0.14	1.00	0.86	0.14	0.00	0.14	8.00	9.12
16	0.86	0.14	0.80	0.83	0.16	0.04	0.18	13.01	9.82
16	0.86	0.14	0.60	0.77	0.20	0.10	0.25	24.81	11.48
16	0.86	0.14	0.40	0.66	0.26	0.23	0.44	65.34	17.15
16	0.86	0.14	0.20	0.32	0.25	0.63	1.81	534.86	82.70

C mean processor time (exponentially distributed)
T quantum of processor time given to each job before being transferred to swapping region
P proportion of processor time given to jobs in pipeline
F proportion of jobs with processor time T
S proportion of processor time given to jobs in swapping region when there are any
R_o probability of no jobs in swapping region
R_1 probability of 1 job in swapping region
Y $\frac{R_{i+1}}{R_i}$ (i > 1)
Q mean number of jobs in swapping region
W mean waiting time in swapping region

The surprising feature about this model is that the mean waiting time in the pipeline is not affected by the proportion of processor time given to jobs in the swapping region when there are any, since if this is high the swapping region is emptied more quickly and the average proportion given to each region is unaffected. With n jobs in the pipeline and a round robin with a small quantum operating for these the mean total waiting time (V) for a job of length x is given by:

$$V = \frac{nx}{1-F} \qquad x \leqslant T$$

$$V = \frac{nT}{1-F} + \frac{C(1-F)}{S-F} + x-T-C \qquad x \geqslant T$$

The second of these is derived by adding together the mean time in the pipeline and the mean time in the swapping region and the difference between the actual residue of time required when it enters the swapping region $(x-T)$ and its expected value (C). This assumed that the swapping region is served on a first come first served basis.

From examining Table 4.6 alone one might deduce that it was best to have S as high as possible since nothing seems to suffer, in fact however this results in a much higher variance in the waiting time for short jobs which might get held up for a long time by a job in the pipeline. The answer is to pick from the table a T, RO, W triplet that gives a reasonable compromise. To get the mean overall waiting time we integrate the expression for V using the formula:

$$\int_a^b \frac{x e^{-x/C}}{C} = \left[-xe^{-x/C}\right]_a^b - \int_a^b (-e^{-x/C})\,dx = \left[(C+x)e^{-x/C}\right]_b^a$$

$$\bar{V} = \int_{o}^{T} \frac{nx}{1-F} e^{-T/C} dx + \int_{T}^{\infty} \left(\frac{nT}{1-F} + W + (x-T-C)\right)e^{-T/C} dx$$

$$= \frac{n(C-CF-TF)}{1-F} + \frac{nTF}{1-F} + WF = nC + WF$$

We can take $n = 1$ as corresponding to a situation with an ever present background load and a relatively light on-line load which always received priority, and the results for this are included in Table 4.6.

Chapter 5

SIMULATION LANGUAGES

Simulation describes the process of building a model of a system, usually simplified in some ways, and observing its behaviour under certain conditions. As there are several different types of simulation, which require different computing techniques, we shall consider first how the simulation of computer systems differs from other simulations.

Because of the complexity of computer systems we must use a digital computer to simulate them. For simpler systems a variety of other techniques are available. Hand simulation, mechanical models, electrical models and analogue computer simulation may be used depending on the type of system. We are also concerned with the operation of the system over time, whereas for some simulations all that is required is the instantaneous state of the system under certain conditions.

We use random numbers to simulate typical user and program behaviour, and collect statistics about various aspects of the system. Hence there is an inherent variability in the results of the simulations compared with a deterministic one, in addition to any inaccuracy owing to the simplications of the model. However this reflects the fact that there is no such thing as a standard user session.

Some simulations are based on a fixed time interval. These usually represent a regular or continuous process in contrast to the irregular and unpredictable pattern of events in a multi-access computer system. A fixed time interval simulation can always be treated as a special case of a variable time interval one.

Whereas the mathematical models described in the previous chapter are abstract and deduce the performance of the system from theoretical considerations, the simulation models resemble the system much more closely, all important components of the system such as jobs, phases, processors, discs, etc. having their analogues in the simulation model. The operation of the system is imitated as closely as is possible and necessary and its performance is observed.

The various components of a simulation model can be active or passive. Active components are represented by activities, otherwise known as processes or transactions, and the same section of program may be used by several activities. At any moment there is one activity running and the others have either not yet started, or halted for various reasons, or finished completely. These activities operate on the passive components of the system, which are represented by variables, queues, lists and data structures.

Whether a particular component of a system, such as a job, a processor, or a scheduler, is active or passive depends on its complexity and particular features of the model and language chosen. The fewer active components there are, the simpler the program is, since fewer routines and facilities for communicating between them need be provided. However, if a component requires complicated decisions to be taken then it will need to be active and the corresponding activity takes the decisions. Usually a problem splits fairly naturally into active and passive components with a few borderline cases.

Simulations of multi-access systems consist of several inter-dependent activities. These are simultaneous in simulated time, but

in an actual simulation on a one processor computer they must occur sequentially. Thus the pattern of control transfers between activities is not hierarchical like subroutine calls but lateral, so that when an activity halts it does not know which activity will run next. Instead it transfers control back to a master routine which decides which activity should run next and transfers control to this activity at the point where it last halted.

Originally most simulations were written in machine code or a general purpose language such as Fortran. Machine code makes it possible to minimize running time and the use of store, both of which can be critical resources in a simulation. However it is extremely difficult to write and debug simulation programs in machine code because of their complicated structure.

A general purpose language makes this easier and has the advantage that complete reprogramming is not necessary if the simulation is transferred to another machine, assuming that the language is available on both machines. However it is usually necessary to include machine code routines in order to deal with the pattern of control transfers described earlier and provide other simulation facilities which cannot conveniently be provided in the general purpose language. A tortuous or inefficient structure and syntax may be necessary.

In order to provide more natural languages for writing simulation programs, with a simpler and more appropriate syntax and a wider range of built-in facilities, special simulation languages were developed. In principle these have the additional advantage that they are machine independent and simulation programs can easily be transferred. In practice however there are so many different simulation

languages that few are implemented on more than two different machines.

It is interesting to consider whether the same type of language could be used for simulation and real-time programming, such as supervisors, process control, seat reservation, etc. This would have the desirable feature that programs could be partially debugged by simulation, since debugging is always a serious problem for systems where a given set of conditions may not be repeatable.

Ideally a modular approach could be used, sections of the real system could be tested in turn with other parts being simulated to generate conditions which might take several months running to occur in practice. Also simulations of proposed modifications could be done without reprogramming in a simulation language and new algorithms developed with simulation could be used directly. This would also reduce the problem of communication between the software writers and each other and the outside world and ensure that the simulation represents the real system adequately.

While such a language might be useful for developing a new system where plenty of machine time is available, for example with a computer manufacturer, it could never be appropriate for all simulations. Many of the operations necessary in real time systems are needed because events are not completely under the programmer's control or because a response must be given within a certain time. Neither of these is true for simulation, and inclusion of too much detail will slow the system down and create unnecessary confusion.

Often an existing general purpose language is used as the

basis of a simulation language. This can be done by providing a set of subroutines, written either in the language itself or in machine code, to perform the basic simulation functions, and compiling the simulation language either into the general purpose language or straight into machine code by modifying the original compiler.

The full power of the general purpose language can be used which makes the language more flexible, and the language can be transferred to another machine with relative ease, particularly if the compiler is not modified. It is much easier to understand and write programs in a language if it is very similar to a well-known one such as FORTRAN or ALGOL, the two most used as bases for simulation languages.

However the syntax is outside the control of the designer and desirable features may prove impossible or too inefficient or tortuous to incorporate. Also inefficiency may be introduced by unnecessary generality in the base language, particularly in the case of ALGOL-based languages. The importance of these factors depend on the language, its implementation, and the type of simulation.

Most simulation languages are either activity-based or event-based. An activity corresponds to a section of program being obeyed, whereas an event is instantaneous, such as a variable changing or a program halting. Any simulation contains both activities and events and the distinction is more one of approach than of types of problem.

In an activity-based language each activity is preceded by a series of leading tests which may involve time cells, these are set like normal variables and when there is no activity free to go at the

present time the list of time cells is scanned to find the lowest positive value. This value is then subtracted from all the positive ones, simulation time is advanced by this amount, and the tests are repeated. In general all the leading tests of each activity are made at each stage, though a more sophisticated system would only perform tests, involving variables that are known to have changed since the last test.

This approach is satisfactory in multi-resource simulations, where several conditions which can be specified in advance must hold before an activity can run. Rescheduling an event merely involves changing the value of the appropriate time cell. It does however involve much unnecessary testing of conditions and searching through the whole list of time cells twice on each occasion that simulated time is advanced. This can be reduced by holding absolute rather than relative time in the time cells, or dividing the list of time cells into sublists.

In an event-based language a list is kept of all the activities that are free to go in order of the time that they may start, possibly combined with a list of those activities that are halted together with the reason. When an activity is halted simulation time is advanced to the earliest time that enables another activity to start, which may be the present. All activities must be caused or freed explicitly, and no leading tests are permitted.

When changing from one activity to another several things have to be done. First of all some test may be necessary to decide whether to continue the current activity or change to a new one. If the latter

then the link must be preserved in order to know where to restart the activity and the variables local to that activity must be preserved or a pointer to them stored. The next activity to proceed must then be chosen according to certain rules, simulation time advanced if necessary and the local variables restored or a pointer to them reset. Only then can control be transferred to the new activity at the address stored in the link.

If an activity is caused by several events each one must test for the others or the activity itself must test for them all and halt itself if they have not all happened, which gives the equivalent effect to a leading test. This has the advantage that the identity of the event freeing the activity can be preserved and only possible activities need be attempted, and by keeping the list in order of time only part of the list need be searched when inserting and removing items, whereas with a random list as for time cells the whole list must be searched whenever simulated time is advanced. There is no reason in principle why time cells could not be in an ordered list, though the list would then have to be reordered every time one was set.

An activity-based language may be easier to program and more efficient when the relationship of events to activities is many-one or many-many, but for simulating computer systems an event-based system is preferable. Each activity is usually caused by a single event such as a clock interrupt or a job finishing, and it is easier and more efficient to treat schedulers as activities that determine which activity (job) is to run next than to have a test in front of each activity which is obeyed on every event. The event-based system also fits more easily into existing algorithmic languages, since the table

of activities gets updated automatically as we have to cause each event explicitly, and the number of activities can change dynamically corresponding to the flow of jobs and phases which avoids having several copies of a basically identical routine.

A distinction is often made between machine-based and material-based languages, though not all fall into one of these categories. The terms derive from the "job shop" problem where there is a flow of material which is processed by machines. In a machine-based language the material is basically homogeneous and is a passive component whereas the machines are the active components and are responsible for deciding the scheduling of which material to process when.

In a material-based language the machines are homogeneous passive components and the material is the active component which decides which machines to be processed by. In the multi-access system the machines would be processors, disc units and users thinking and the material would be jobs. This does not fall clearly into either category, since both may be non-homogeneous and so any language which is oriented solely towards one view will be artificial in some way.

The main difference in practice between the two types of languages are the automatic statistics gathering facilities or standard machines that they provide. A language which leaves this for the user to specify can be used in either way.

At Cambridge there was no existing simulation language implemented and so the choice lay between writing a simulation system for either an existing or a new simulation language. Writing the

model in machine code or a general purpose language had been ruled out for the reasons stated above.

There are many simulation languages in existence, some of the better known and more interesting ones being CSL[5,6,8], Simula[13,48], Simscript[14], OPS[22,31,32], GPSS[27], SOL[37]. This list excludes those that are unsuitable for multi-access system simulation, for example those with a fixed time increment. Detailed comparison of the various languages can be found in (44,61) and most of the other papers on simulations describe and compare some of the languages, so only the more important points will be discussed.

CSL is an activity-based language that is designed for compilation into Fortran. It was one of the earlier languages and lacks some of the facilities and improvements in later ones. Because it uses the Fortran subroutine structure, each time that an activity halts it must either specify the next activity explicitly or else leading tests must be made at the beginning of each activity in turn until one is found that is free to go. It is impossible to restart the halted activity at the point where it halted unless this is treated as two separate activities which results in extra testing. It is thus more suitable for simulations with activities which occur regularly or have complicated conditions before they can start than for simulations of multi-access systems where events happen randomly in time and can be caused explicitly.

Simula is an Algol-based language. It contains Algol as a subset and a Simula block can be enclosed in an Algol program without requiring the whole program to be compiled by Simula. A good range of

facilities are provided, particularly for scheduling events, which can be scheduled before or after other events at the same or a different time, or rescheduled at a different time. Its main disadvantage is the block structure of Algol, which makes it impossible to include machine code or precompiled subroutines and means that if any change is made the whole program has to be recompiled, which can take a lot of central processor time. It would also be unsuitable for on-line use, since Algol uses a larger amount of store and tends to run slower because of its greater generality compared with Fortran.

Simscript is a Fortran-based language containing Fortran as a subset. It is event-oriented and each event must be caused explicitly. However it is impossible to schedule events relative to other previously scheduled events, or find out when events are due to occur or reschedule them. There are also no debugging facilities and again if any change is made the program has to be recompiled.

OPS-3 was written at MIT for CTSS and is now being extended considerably to give OPS-4 which is being implemented as part of the Multics system[11]. OPS-4 is based on PL/I with the compiler also written in PL/I, however it uses features in Multics which will make it difficult to implement elsewhere. Apart from this fact and the lack of any evidence on the size of the compiler and the store and processor time required it sounds an excellent language, which combines the best features of all the others and adds several more. It is designed for on-line use, and so sections of the model can be tested interpretively or be portrayed by the console user, and then compiled for greater efficiency. It is not necessary to recompile the whole program when changes are made and a mixture of compiled,

interpreted and user-portrayed sections can be used. Comprehensive on-line and off-line debugging facilities are available and a wide variety of output is possible.

GPSS was designed for non-programmers and is based on a flowchart approach, each box on the flowchart corresponding to one instruction in GPSS, typically joining a queue or waiting for a specified time. While operations of this type are easy because of the built-in structure, no general mathematical facilities are available. Good debugging facilities are available and the language is easy to follow for a simple simulation, though more complicated ones are difficult or impossible. The language is rather inefficient because it is executed in a partially interpretive fashion, and automatic statistics are gathered for queues, facilities, and stores. Routines in other languages cannot be included and compilation is necessary if any changes are made.

SOL takes the basic concepts of GPSS and includes them in an algebraic language similar to Algol. A sophisticated system of priorities for queues and deciding which process to execute next is available, and it is possible to create and destroy activities dynamically. However it is inefficient for the same reasons as GPSS.

None of these languages were really suitable for the task in hand, except OPS-4 which would have required a PL/I compiler and had not been published at the time that a language was being chosen (May 1967), and there appeared to be no demand among users for a simulation language. A new language was therefore designed and implemented based on Fortran and called FOSSIL - Fortran Systems Simulation Language. The name does not reflect the philosophy behind it!

Chapter 6

FOSSIL - Fortran Systems Simulation Language

Although old-fashioned, FORTRAN is still very popular because of the simple reason that it serves scientific users' basic needs well and almost every machine has a compiler for some version of it. It is capable of being compiled quickly and producing efficient code and does not usually use as much store or time for compilation or execution as ALGOL and similar languages.

The Cambridge T3 FORTRAN Compiler[40] is part of MLS (Mixed Language System).[41] This makes it easy to incorporate machine code and precompiled routines, and provides a library facility whereby only routines which have been changed need be recompiled, and all other routines needed are loaded automatically. The routines that are responsible for changing the current activity and advancing simulation time and dealing with interlocks are written in machine code for efficiency, and yet they are called like ordinary FORTRAN subroutines. No other high level language offers such a facility.

The system was tailor made for the applications and easy to change when necessary. If an existing system had been implemented many of the features that were time-consuming to implement might have proved redundant for this type of application, and difficult to change if necessary, and the syntax would not be so easy to understand. It is of course always easier to follow something one has designed oneself, and not least among the advantages was the valuable experience gained.

The ML/I macrogenerator[3] is used to remove some of the inelegancies of FORTRAN syntax and introduce various new features

necessary or desirable for simulation. This could be avoided by modifying the FORTRAN compiler, but this would mean another version would have to be maintained, and give a negligible gain in efficiency, since the macrogeneration time is relatively small.

The most important features in simulation languages once the basic type has been settled can be summarised as follows:

1) Control and timing of activities
2) Interlocks and queues
3) Data structures
4) List processing facilities
5) Statistics output
6) Debugging facilities

These will be discussed in turn with reference to FOSSIL. Vertical bars each end denote an item of a given type, for example |routine name| means the name of the routine concerned.

The reasons for choosing an event-based language have been given in Chapter 5. Usually in an activity-based language all the activities are active all the time and control statements are unnecessary except for stopping the simulation. Originally the system started attempting to obey all the activations of all the routines at once. This soon proved unsatisfactory and was replaced by a flexible system where it was possible to have a variable number of concurrent activations of the same routine which could be begun and finished at will.

Initially only the main program is active. This can do any necessary initialisation, start the simulation, intervene at intervals

to print out debugging information, test for convergence, tabulate results or restart the simulation with different parameters, and finally stop the simulation.

An agenda is kept of all current activities, sorted in ascending order of the time they are free to go if not otherwise halted, this time is known as their schedule. A BEGIN (|routine name|, |index|) instruction inserts a new activity in the agenda and marks it as free to go now. The local variable INDEX is set to the value of the second argument. This is automatically saved every time the routine is halted and restored when it is restarted and it serves to identify a specific activation of a routine, both for the purpose of referring to the activity in other instructions and for subscripting arrays.

A TERMINATE (|routine name|, |index|) instruction will remove an activity from the agenda without affecting any other activations of the same routine or the activity that obeys the instruction. A FINISH instruction removes the current activity from the agenda and proceeds with the next one that is free to go.

A RESTART instruction removes all activities from the agenda except the current one and resets simulation time to zero. This can be used where several runs are required. A STOP instruction will stop completely, though it is possible to run other jobs after returning to MLS.

The simulation time is held in the common variable TIME which is automatically declared in each routine and can be used as a value but should not be set. Special instructions are required to advance simulation time since it is necessary to test to see whether any other

activities are free to go, and RESTART is the only way of setting it back to a lower value.

The instructions DELAY (|interval|) and WAIT (|newtime|) shedule the current activity at TIME + |interval| and |newtime| respectively, if this is less than or equal to TIME the activity just continues without any change. Otherwise the first activity on the agenda that is free to go is resumed and TIME is advanced accordingly if necessary. The function SCHEDULE (|routine name|,|index|) gives the schedule of an activity on the agenda. The function NEXT (|nexttime|) sets |nexttime| to the schedule of the next activity on the agenda that is free to go. The instruction RESCHEDULE (|routine name|,|index|, |newtime|) reschedules an activity on the agenda at MAX (TIME,|newtime|), this is to avoid the possibility of scheduling an activity before the current one.

This system allows direct scheduling of other events and look ahead, and though not as flexible as SIMULA or OPS-4 which permit scheduling before and after other activities and before activities with the same scheduled time, it allows all the possibilities required in practice.

Interlocks and queues are as necessary on a simulated system as on a real one and most attempts to provide them have not dealt with all cases. The most notable is that used in SOL, where a system of FACILITIES and STORES is provided. A facility can only be held by one routine at a time with a certain control strength, and if another routine requests it with a higher control strength the holding routine is interrupted, otherwise the requesting routine queues for it according to its control strength until it is free. A STORE is similar

except that varying amounts can be requested up to a maximum capacity and no interrupts are possible since the control strength is always the same.

Originally a variation of this was implemented in FOSSIL, where a distinction was made between a pre-emptive (SEIZE) and ordinary (REQUEST) demand for a facility. The former would interrupt any activity holding the facility with a lower control strength, whereas the latter would wait until the current holder had released it but jump the queue according to its control strength. This gives a neat way of describing a multi-level scheduling system such as is used in Project Mac CTSS, where each level corresponds to a different control strength.

Under this system it is difficult to schedule jobs externally, that is to have an activity C that removes a facility from activity A and hands it to activity B. This could be overcome by a system of dynamic control strengths, but routine C would have to test and change the control strengths of routines A and B and other routines on the list might need their control strengths altering at the same time. Allowing interrupts involves keeping a list of all the facilities held by a routine, or all the routines queueing for a facility, which involves a considerable amount of list processing. Without interrupts a routine can only be queueing for one facility at a time and no list is necessary, since the timing mechanism will automatically keep the queues in order.

The instructions QUEUE (|queuename|) and NEST (|queuename|) examine the value of variable |queuename|. If it is negative or zero then the activity proceeds, otherwise it halts and is marked as queueing for |queuename| at the back or front of the queue for QUEUE and NEST

respectively. In either case the value of variable |queuename| is increased by one. If a queueing activity is rescheduled then its place in the queue may alter, and this facility may be used to shuffle a queue if necessary.

The instruction RELEASE (|queuename|) reduces the value of variable |queuename| by one and frees the activity in the front of the queue for |queuename| if any. If |queuename| is an integer or long integer variable it gives the number in the queue including the current holder. By setting |queuename| negative originally this system can be used for stores with capacity greater than one, though the previous sentence no longer applies then.

The instruction BLOCK (|routine name|,|index|) will halt an activity other than the current one and UNBLOCK (|routine name|,|index|) will free it again. This operates independently of the other interlocks and after an UNBLOCK instruction an activity may still be halted for some other reason or scheduled to begin later.

The instruction HALT (|arg|) will halt the current activity and store the argument and FREE (|arg|) will free all activities halted with that argument. Note that it is the value of the argument rather than the name that is important, which allows it to be an integer, and it should be strictly positive.

Some authors have emphasised the importance of data structures and lists for representing queues of items and attributes of an item in simulation languages. These are more important for applications where the number of attributes is large or there are several classes of entity.

Simple lists can be represented as an array where the value of each item is the index of its successor, that is the address of its successor relative to the base of the array. The attributes of an item may be held in separate arrays with mnemonic names or condensed into one array according to the convenience of the programmer, in each case the array is subscripted by the index of the item. The end of the list is marked by a zero value which avoids special action when the first item is inserted in a null list or the last item is removed from a list. For example consider a queue array of four items, jobs numbered 5, 6, 1, 4 whose contents would be:

JOB(0) = 5, JOB(1) = 4, JOB(2) = JOB(3) = JOB(4) = 0

JOB(5) = 6, JOB(6) = 1, other elements all zero.

This is a compromise between the demands of efficiency and convenient implementation and the need to be able to refer to items by convenient names. If desired attributes can have coded values for example if we set ONLINE = 0 and OFFLINE = 1 we can then use statements like:

```
IF (STATUS (4) - ONLINE) 1,2,3
```

to avoid having to remember the code. The declarations required for automatic assignment of these codes would offset any gains from this.

Arrays with a single suffix are more satisfactory because they can conveniently be accessed in one order using a modifier register whereas those with several require multiplication or table look-up. Thus having the attributes in separate arrays is convenient both from a mnemonic and efficiency point of view.

The approach described above is used in FOSSIL and the usual list processing functions are provided as FORTRAN subroutines.

T3 Fortran permits the use of zero as a suffix for one dimensional arrays. If necessary they can be programmed in machine code for greater efficiency, though this is much harder to debug and only worthwhile for those that are used very often.

Facilities are also provided for manipulating the circular queues where the last member points back to the first, though the zeroth element of the array still points to the first or is zero if the queue is empty. These are particularly suitable for Round Robin schedulers, where after dealing with a job it would have to be placed at the bottom of an ordinary queue but need not be touched in a circular queue as the pointer is merely advanced. Unfortunately the same technique cannot be used for the activities list, where we have to preserve schedules of the activities in order.

Also important are functions to insert an item in a list in a position dependent on the value of an attribute, either in increasing or decreasing order of the attribute, when the attribute is held in an array indexed by the item number.

The possibility of lists with two way pointers was considered and dismissed because the extra programming and operations required outweighed any advantages gained in putting items on the end of lists, which is the only operation that can be performed much more satisfactorily with these. The same reasoning applied to the list of activities. A compromise solution would be to keep a pointer to the last member of the list, but even this would have to be checked and updated if necessary.

An important facility of any simulation language is the ability to collect and tabulate statistics conveniently in the form of

histograms and graphs. Some languages such as GPSS and SOL provide automatic tabulation of the use of facilities, queues and stores, but this consumes unnecessary time and paper and is difficult to provide in a language where these are not explicitly declared and the maximum queue size is unknown.

In FOSSIL a TAB (|table name|,|arg1|,|arg2?|) instruction is provided to insert a value in a table which may be one of three types. The first is a conventional histogram where the number of occurrences of each range is recorded, the next records the mean of one variable |arg2| for each range of the other |arg1|, and the third records the amount of simulation time that the variable spends in each range. |arg2| is only necessary for tables of the second type. By using the third type and inserting a TAB instruction before each QUEUE, NEST or RELEASE instruction the mean length of queues can be recorded since the value of the queue variable is equal to the length of the queue if in long integer mode. Summary tables giving just the number of entries or total time, maxima, minima and standard deviations can be printed, or full details of the entries in each range of the table, in a compact or detailed form with five or one ranges to a line respectively. Out of range entries are also recorded.

Tables can also be plotted to any desired scale or displacement of axes, and it is possible to plot several tables on the same set of axes with only one instruction for each. The axes are marked out automatically in an appropriate scale with major divisions every ten minor divisions enabling them to be calibrated easily as required.

A random variable can be sampled from a table, a rectangular distribution being assumed for each cell and the out of range entries if any. One can also have several random number streams and generate exponential and other distributions from these as required. The seed for the pseudo-random number generator can be set to repeat a sequence or produce a different sequence.

Another important property of simulation languages is the ease with which programs can be debugged. With several activities in simultaneous operation it is easy to get confused about the location of errors and wonder in which order the activities were obeyed. FOSSIL has an optional trace system that can output on any stream to avoid confusion with other output. When the trace is on, each time an instruction that could change the status by freeing or halting the current or another activity is obeyed the instruction, simulation time, routine name, index, label and line number relative to it are printed. Unfortunately the line numbers refer to the macrogenerated program, but this can be preserved, and one can usually guess where it is in the original.

Facilities are provided for switching the trace on and off dynamically and it is possible to trace only those activities in which one is interested or to trace different activities on different streams, since a separate value of the trace stream, which may be unset, is kept for each activity.

There is also a STATUS instruction which will print out all the activities on the list with the TRACE details of the last time they halted and the schedule. The trace routine is carefully programmed to give a minimal overhead, particularly when it is not required. This

is important since it is entered for every simulation instruction.

The scope of variables is similar to Fortran but new facilities have been introduced by the macrogeneration. A program can be preceded by any number of GLOBAL |declaration| statements which will copy the |declaration| at the head of each FUNCTION or SUBROUTINE.

The result must obey the normal FORTRAN II conventions about the order of declarations. The declaration GLOBAL COMMON TIME,INDEX is assumed. Each routine may contain any number of declarations of the form LOCAL |arg1|,|arg2|, etc. These arguments, which should be simple variables, will then be preserved when an activation of the routine is halted, and restored when it is resumed. They are thus local to each activation of a routine. The variable INDEX is effectively local, but it is treated specially, since it is used to identify activities in the instructions which refer to them.

This facility has obvious disadvantages as far as efficiency is concerned, and when this is crucial it may be better to treat the variable as an array with subscript INDEX so that it does not have to be preserved and restored each time. The main object of using LOCAL instead is to make programs more legible and simpler to understand. If the variable is used a great deal compared to the number of halts it may actually be more efficient to avoid the subscripting each time by using LOCAL.

No real attempt is made to retrieve storage except the six word blocks used for the list of activities, and if activities using local variables are created and destroyed too often this might prove a problem, since holding storage maps and piecing together variable sized blocks was not considered worthwhile.

The six words of the agenda entry for each activity contain the pointer to the next entry on the list or zero if at the end, the address where local variables are stored if any, the start address of the routine (for identification), the trace stream if any, the routine name in characters, the label and line number, the link, the value of INDEX for the activity, the scheduled time, and the status. These are all halfwords except the routine name and the scheduled time which are full words.

The status-halfword is:

0	If free to go
-n	If halted for reason n
Address of Q	If queueing for Q
+0.1	If blocked

Only this halfword need be tested to see if a routine is free which makes the scan of the list very fast. When a routine is halted it goes behind all other routines scheduled at the same time except for a NEST instruction when it stays at the top of the list, and thus the queueing is accomplished automatically using the same list as for the timing.

An example of a FOSSIL program is given in the Appendix. One must now consider how useful the language has been, and how it could be improved. Leaving aside the utility of simulation as a technique, which is discussed elsewhere, the alternatives once simulation had been decided on were machine code or the use of a non-simulation language, which were unacceptable for the reasons discussed above.

A set of subroutines written all in Fortran might have fulfilled some of the necessary criteria, but efficiency of time and store usage would have been a major problem. Several parts of the machine code sections had to be recoded to run faster and the tabulation routines, which are all in FORTRAN, can represent serious overhead and were replaced in some models by special purpose facilities.

Even if the temptation to slip into machine code were resisted it would probably have been necessary to exploit various non-standard features which would have destroyed the advantage of compatibility. Accordingly some special purpose language was necessary.

FOSSIL never set out to be a general purpose simulation language, though some features were included which were not of direct relevance to the problem partly with this in mind. As a language for simulating computer systems it has proved very satisfactory and more appropriate than most other languages, because it was specially designed for that purpose.

An ideal language should have its queue manipulation and list processing facilities combined with its timing facilities and activity control, this would solve some of the more tortuous interlock problems and give a more flexible system of queues and delays with the standard simulation functions (e.g. WAIT, DELAY, QUEUE) as special cases. The disadvantage of such a system would be a considerable drop in efficiency if the language was based on another algorithmic language, such as FORTRAN, which did not incorporate these facilities, since it would be difficult to insert them efficiently and conveniently. SIMULA, an ALGOL-based language with a separate compiler, goes some way towards solving this problem, but though the first list processing

language LISP has been in existence a long time nobody has succeeded in incorporating its facilities satisfactorily into a general purpose algorithmic language.

An important deficiency as a result of this which is difficult to rectify within the present framework is the lack of loops for members of lists and facilities for finding members of lists given a set of conditions, such as are provided in CSL. One may wish to choose a random member of a list or see if one exists satisfying certain conditions, or to perform a set of operations on all members satisfying certain conditions. These operations can be programmed in an ad hoc fashion or even as functions, but there is no convenient means of specifying a sequence of instructions as an argument. This difficulty is not present in ALGOL-based languages, whose block structure gives them a major advantage over FORTRAN-based ones, but not sufficient to compensate for the other disadvantages mentioned above.

Another major improvement would be a statement of the form "Delay T while not halted" or "Hold facility F for time T". Both of these were implemented but the inefficiencies involved in the attendant testing and shuffling of the list of activities were too serious, and they were abandoned in favour of ad hoc recording of the time each job is in possession of the central processor, which would have been their main use. There seems to be no satisfactory compromise solution to this problem.

Large simulations generally take too long and have too many operations to make on-line interaction an attractive proportion except for testing, though the use of visual input-output could make a big difference as one could then watch queues grow longer and shorter and

get variables monitored continuously to check for errors.

One could also intervene to change the value of a parameter if things were not going as expected and continue without recompiling. The latter is practicable at the moment for COMMON variables which are stored in a fixed location, but if interaction in the source language is required then an interpretive language must be used. For most simulations running time is an important criterion and this technique is not viable. Visual interaction also puts a big load on the central processor and store and channels, and for this application the result is not sufficient to justify this. However OPS-4 will have facilities for on-line interaction and visual display.

Some languages, such as Simscript, have report generators which are useful but difficult to build into an existing language. In FOSSIL flexibility of output for TABLEs has been sacrificed to some extent to avoid having many arguments or a complicated algorithm to decide what is wanted. However the options are satisfactory for most purposes, and with a knowledge of the storage allocation FORTRAN routines can be written for special purpose styles.

Chapter 7

SIMULATION MODELS AND RESULTS

We have already seen some of the reasons for simulating computer systems and some of the difficulties that are likely to be involved. We want to complement rather than duplicate the analytical models by investigating models with more accurate representations of core store allocation and scheduling. We want not only some absolute measures of the performance of such systems but also to compare them against each other in various circumstances to see which fares better. We should thus regard simulation as an experiment rather than a means of producing accurate figures, in particular we can have more confidence in comparative figures than in absolute values, provided that we are comparing like with like.

Simulation has been a popular technique for modelling computer systems, since it can be used for systems where there is no satisfactory mathematical model, and the level of detail is completely under the control of the modeller. Fine and McIsaac[16,17] used it to model the SDC time-sharing system,[56] which is similar to CTSS with a drum and using swapping. Greenbaum[21] used another computer to produce a reproducible load for evaluating and testing multi-access systems. Hutchinson and Maguire[28,29] used Simscript to evaluate a batch processing system with three 7090s from a broader point of view, by comparing it with bureaux for different types of job taking factors like manpower and operator imposed priorities into account, and also to determine the effects of hardware and buffer size modifications to the manufacturer's software.

Katz[33] used Simscript to model an IBM 7040-7090 Direct Coupled System, where the smaller computer (7040) feeds the larger one (7090) with a stream of jobs and receives the output, the 7040 dealing with the peripherals. Katz[34] has also used Simscript to model the System/360, but although consoles are included in the model the input from them and output to them are regarded as independent and there is no paging in the model.

Nielsen[46,47] has simulated a third generation multi-access system before it was installed but he points out that the cost of doing this properly is high and efficiency and feasibility may impose constraints on the design and hence the fidelity of the model. One computer manufacturer considering a simulation model of their third generation system decided against it when they estimated that designing and running a good simulation would cost as much as developing the system.

Few people have combined simulation and analysis of the same system. Penny[49] did so for a simple system by getting analytical upper and lower bounds for the performance of the system over a range of parameter values thus producing an envelope of two curves between which the results must lie, then by simulating with a few carefully chosen parameter values he was able to estimate the performance for the whole range by interpolation. This technique could cut down the amount of time needed for simulation considerably if suitable applications of it could be found.

Scherr[54,55] used the Markov model described earlier and simulation in his own language to model CTSS and found a very good agreement between the results of the two, and the performance of the

actual system. However in this case the main use of the simulation was to provide a check on the validity of the analytical model by using a more accurate representation of the system.

One of the first problems we encounter in simulation is whether to use data from an actual run of the real system or whether to generate data from the distributions observed from several runs or convenient approximations to them. The former has the advantage that it is a genuine sample and so no unrealistic values can be generated, however a random sample is not necessarily a typical sample and is unlikely to have the same means and distributions as the results from several weeks' running, unless we make it rather long. The storage of the data also presents a difficult problem since we will not necessarily use them in the same order as they occurred in the original run. Effectively a different stream of data is needed for each job.

The choice between using an actual distribution or an analytical distribution as an approximation if generated data is used is governed by convenience and goodness of fit. For distributions that are very unbalanced, such as the negative exponential, the analytical one is better if it is a reasonable approximation, since an equal interval histogram will give inaccurate results at the ends. Routines are provided in FOSSIL to sample from histograms which have been generated earlier and stored or fed in directly, here a uniform distribution is assumed within each interval and between the minimum or maximum value recorded and the appropriate end range.

The next problem to be considered is how much detail should be included. Any model is no more accurate than its weakest link, and to simulate some parts of the system in much more detail than necessary

is pointless unless detailed insight is required into those parts, and even that is not likely to be accurate. Computer and programming time is wasted, there is more scope for programming bugs to creep in, and the model is harder to understand and deal with.

In practice however it is very difficult to tell when the required level of detail has been reached, since there is no common standard to measure detail by and there may be nothing with which to compare the results of the model. One can ensure that the time scale of all activities in the model is the same, and no activity has events which can be grouped together occurring much more frequently than anywhere else in the model. Randell and Zurcher[51] suggested starting with a modular model with each part of the system represented by a very simplified "black box" and gradually develop the detail in each simultaneously. In principle this should make it easier to debug the model and tell when the detail in any specific part makes any difference or not.

Although any simulation model is bound to change and develop a lot this approach has the disadvantage that the interfaces between the various parts are likely to require changing a lot, thus removing the advantage of compatibility with the previous versions and requiring extra programming effort in the long run. Also defects in the initial model may be perpetuated when a new approach not corresponding with the initial model is required. Looking at the final model described below it is difficult to see what initial model it could have been derived from, indeed its lack of modularity contributes to its efficiency.

Another approach is to try and isolate the critical factors by comparison with analytical or other simulation models, which can enable one to omit factors if the model proves to be insensitive to them. However it is dangerous to apply conclusions reached from analysis or one simulation model to another where different factors may be critical and the only safe comparison is between a model with a factor included and the same model using the same data without it. If these prove sufficiently similar the simpler model can be used for other parameter values and small modifications, but if any major modifications are made the comparison should be done again.

In practice it is usually the ingenuity and judgment of the analyst, the resources of computer time and store and the human time available that decide the complexity of the model, and for this reason simulation is often looked on with suspicion.

The next difficult question, in some ways very similar to the previous one, is how long to simulate for. If there are many long runs the cost in computer time and delays in developing the model can be considerable. However simulation for too short a period produces a wide variability in results, and one cannot attach much confidence in them. There is no obvious place to stop since the accuracy does not increase proportionally with time, and to aim for a mean which hardly changes when another run is done is asking for too much, since simulation results are inherently very variable just like the real systems they represent. By comparing the results from several runs one can check that one's sample is reasonably typical and take the final decision on the same sort of basis as mentioned above.

When comparing different values of parameters or different algorithms we want as far as possible to be comparing like with like. If we have a different stream of random numbers for each of the quantities to be sampled and use the same streams for each of the two cases then we get them as similar as possible. If the parameter being changed is the mean of one of the variables being sampled then they will all be reduced proportionally but occur in the same order, subject to a few more or less at the end. All unaffected variables will sample the same values in the same order.

The random numbers are basically generated by a multiplicative congruence method, which produces a uniformly distributed sequence with a very large recurrence period. These are known as pseudo-random numbers, since they are determined and thus repeatable to allow comparisons like the above and checking of results. To get a negative exponential distribution with mean 1 we take the negative logarithm of a uniformly distributed sequence between 0 and 1.

One alternative that was considered was to use the set of exponentially distributed numbers which were most representative as described in Chapter 3, permuted in a random way. If all of them are used this will produce a much better sequence to represent the distribution than a truly random sample, since it will have exactly the right mean and variance. However this would involve one set of numbers for each stream being held on disc and read down, since there would not be room in core store for a sufficiently large set. It would also be impossible to ensure that exactly all the numbers were used, since the different streams are consumed at different rates which cannot be determined in advance, and the method depends on this

for its accuracy. However there is possibly a use for this method to determine a more typical set of numbers to represent the tail of the distribution. This illustrates the fact that a representative as opposed to random sample is required.

Another method for reducing randomness and increasing representativeness can best be described by treating random numbers from a uniform distribution as marks on the edge of a circle, the distance around the circumference from a fixed point being proportional to the value of the number. If we superimpose the reflection of the circle about the diameter through this point on itself we get a set of twice as many random numbers which have exactly the right mean. This is the method of antithetic variates[23] the practical implementation of which is to have one run with random numbers $x_1, x_2, x_3 \ldots$ and then another with random numbers $1-x_1, 1-x_2, 1-x_3 \ldots$ assuming the x_i are between 0 and 1.

Because the random numbers are usually transformed to produce a negative exponential distribution and the runs finish in slightly different places if they are of the same length the mean is not exactly correct, but there is still a significant improvement. This was adopted for the present work and extended still further by taking the two rotations of the circle through 120° and 240° and then reflecting and superimposing these as well to give effectively six sets in each run, thus producing a more even distribution all the way round without prejudicing the randomness by making the runs too short.

Next we must consider the type of output that is required, both for diagnostic purposes and as results. It is important not to

mix these two together otherwise so much paper will be required to contain a few results that it is difficult to make quick comparisons and see patterns. There is a need for both detailed results and a summary of a run on one page or so, this can conveniently be done by using different streams for the different purposes. The trace facility in FOSSIL which can use different streams for different activities was designed with this in mind. Other sorts of diagnostics will be necessary both as general checks that all is functioning correctly, particularly in the more complicated routines like scheduling and core allocation, and to detect specific errors. By printing out related information in different routines one can check for consistency.

It is sometime difficult to avoid generating too much diagnostic information, and all diagnostics should have a switch so they can conveniently be removed if necessary. If the time at which an error occurs is known one can switch on the diagnostic shortly before it, and on the Cambridge system the output can be edited by successive bisection if the program is looping to find where the loop started without having to print it all.

For the results only the values of interest should be printed for the reasons stated above, since other values obscure the detail. However it is important not to have too much aggregation of results, for example it rarely makes sense to average off-line and on-line results since the two are so dissimilar. This also applies to the various possible program store sizes, since they yield very different results. It is also better to generate directly all results that will be needed even if it involves more initial effort, since it is

so tedious and error prone to go through working out means and totals and drawing graphs from other figures when these could have been produced by the computer in the first place. This is particularly true when it turns out to be necessary to have several different runs, as is often the case.

If several runs are made of the same simulation it is desirable to have both the figures for each run and also the combined totals and means for all the runs, since the former is necessary to see how much variation there is from one run to another in order to see how much confidence can be placed in the results, and the latter to be able to make quick comparisons between runs with different parameters.

Having considered some of the factors that are important in designing simulation models, we shall now examine the basic model which corresponds as far as possible to the present state of the Cambridge multi-access system. This will serve to illustrate some of the points above and the approach that has been adopted towards the simulations.

The basic model consists of about 900 lines of FOSSIL, in addition there are 400 lines of FORTRAN and 800 orders of machine code in systems subroutines. Data gathering and printing represents a significant proportion of this even though this is kept to a minimum as far as possible. Since the operation of making an entry in a general table is a lengthy one special purpose routines were used when more efficient, such as for calculating time averages. This avoids the necessity of checking for out of range entries, multiplying by the increment, etc.

Various techniques were adopted to minimize the delays inherent in running such a large program. After each compilation the binary program was dumped on the disc, thus saving further compilation and loading except for major changes to the program. If there is an error, portions of the code can be examined on-line and changed easily by using the system program PATCH. One can identify the relevant section by using the command to search for the orders setting the name of the subroutine and the line number for the diagnostics of the FORTRAN system.

The relocatable binary was also stored on the disc, so if it was necessary to recompile some subroutines the unaffected ones need not be compiled too (except for the main program). This resulted in a considerable improvement in response time on-line and also saved central processor time, as macrogeneration and compilation of the complete program took about a minute. When running on-line it was often convenient to have a short test run for a few seconds after a change had been made, since this could detect some of the more blatant errors, and only intiate a full run from the dumped version after this, rather than wait the extra time for a potentially long job to be run only to find that it had misfired almost at once.

The FOSSIL section of the program is given in the appendix, but this is intended primarily to illustrate what a simulation program in FOSSIL looks like. The text refers to the names of the most important subroutines in order to show how the program is divided but only the basic structure of the program and its most interesting features are described.

The main program's sole function is to call a subroutine called MAIN, which does all the work that the main program would normally be expected to do, this was because the main program has to be recompiled every time any other subroutine is compiled or else held in a different file from the remainder. This subroutine initialises all the arrays, variables and tables, begins two activations of the routine JOBGEN to create off-line and on-line jobs respectively and the activity ALG2 to do the scheduling and then dumps the program. After this it reads in the data giving the number of runs and their length and the serial number of the algorithms to be used which are selected by means of switches on the serial number at appropriate points. There is then an initial period of 15 minutes simulated time to allow the system to warm up, since the simulation starts with an empty system. After this all the tables and other data gathering variables are reset to zero, since it is difficult to switch off the data gathering, and the run is started.

The routine MAIN interferes six times in each run to reset the random number variables in accordance with the algorithm mentioned above, and at the end of each run it outputs all the statistics, adds on to the cumulative tables the details for this run, resets the other tables and the data gathering variables to zero and starts the next run if there is one. All running jobs remain in the system unaffected and the second run starts from the situation at the end of the first. At the end of the final run, if there is more than one, the cumulative results are printed out. Merging the tables at the end of each run avoids the inefficient process of trying to collect data in two tables simultaneously.

The routine JOBGEN has two activations which create off-line and on-line jobs respectively. These run simultaneously in simulated time, and are differentiated by a different value of the variable INDEX. They use the empirical data on the number of off-line and on-line jobs running given in Chapter 3 and synthesize this distribution in a way that is designed to be reasonably similar to the actual pattern of job arrivals and departures. This was not recorded in the statistical measurements on the real system, since it is so variable. There is an additional difficulty insofar as it is not possible to stop a job at any time, because of the facilities it may be using and the queues it may be in, though it is possible to start one at any time. When this activity wishes to stop a job it reduces the desirable number of jobs running by one. When a job reaches strategic points it checks to see that the actual number of jobs running is equal to the desirable number, and if there are too many running it stops gracefully. When this activity wishes to create a job it increases the desirable number of jobs running by one, and only creates a new job if this is greater than the actual number of jobs running. The inaccuracy produced by this lag is not significant.

After each creation or termination of a job the activity waits for a time that is exponentially distributed with mean proportional to the fraction of the total time with that number of jobs running in the empirical distribution. It then randomly decides with equal probability whether to increase or decrease the number of jobs. With zero or the maximum number of jobs running there is obviously only one alternative, and this is compensated for by multiplying the mean time spent in these states by two.

If we let the empirically determined distribution probabilities of the number of jobs running be a_i for $i = 0,1...n$, and the distribution probabilities determined by this algorithm be p_i for $i = 0,1...n$, we have in equilibrium that the rate of entering state i is the same as the rate of leaving state i, using the theory of Markov chains as described in Chapter 4. This gives the following equations:

$$p_o \left(1 - \frac{1}{2ka_o}\right) + \frac{p_1}{2ka_1} = p_o$$

$$\frac{p_{i-1}}{2ka_{i-1}} + p_i \left(1 - \frac{1}{ka_i}\right) + \frac{p_{i+1}}{2ka_{i+1}} = p_i \qquad \text{for} \quad i = 1,2...n-1$$

$$\frac{p_{n-1}}{2ka_{n-1}} + p_n \left(1 - \frac{1}{2ka_n}\right) = p_n$$

which are solved by $p_i = a_i$ for $i = 0,1...n$, i.e. our algorithm reproduces the empirical distribution. The constant k detemines the rate at which the number of jobs running changes, a value of 500 was used to give a reasonable compromise between a very rapidly changing number, which would suffer more from the lag mentioned earlier, and a slowly changing number, which would cause bigger fluctuations in the load as backlogs built up, the extreme being a system which gradually added jobs until the maximum number had been reached and then gradually terminated them again. The largest individual probability is 0.16 giving a mean time before a change of 80 seconds, though most of them are rather less than this. There was an alternative to generating jobs and terminating them explicitly of keeping them all going and halting and freeing them, but the former was chosen as being conceptually simpler and more faithful. This also keeps the list of activities on the agenda shorter and thus saves time when searching it.

The routine JOB is the heart of the program and one activation of it exists for each running job. It consists basically of a loop which is obeyed continuously as long as the job is running, the only difference between on-line and off-line jobs being different means for the various time and store parameters and the inclusion of a stage in the loop representing console input-output and thinking for the former. No attempt has been made to reproduce identically the various possible paths that real jobs can take, but rather to generate a typical mix of activities for a given number of jobs running. The first action is to join the end of the scheduling algorithm's queue for processor time, and then call the sampling subroutine.

The subroutine SAM samples the size of store to the nearest 4K from a histogram, and chooses an exponentially distributed processor time with the mean corresponding to the store size. The result of this is to maintain the correlation between store size and processor time noted earlier, and produce a hyperexponential distribution of processor time, which is more faithful than an exponential distribution, as can be seen from Figures 3.3 and 3.4. If the sampling procedure gives a store size of zero this signifies a pure procedure with 64 words of working space and the probabilities and mean processor time are set accordingly. These can be treated as using no store, since there is a special region of store reserved for them which has nearly always spare space, and a copy may well be in the store already. If the sampled store size is not zero, the job calls the subroutine GEST to get store, which halts if necessary until the store is allocated and returns. Tabulation of the waiting time for store and the use of it, and recording the throughput and

processor use is all done in routine JOB. The maximum waiting times for store are also recorded to give an idea of the range.

After acquiring store (except for a pure procedure) the job makes a disc request by calling the subroutine DIS, and if necessary joins a single queue for two notionally separate discs. The justification for this assumption has been discussed in Chapter 4. To give a more realistic picture half the disc time is used before the processing, corresponding mainly to program loading and file access, and half is used after, corresponding mainly to output. After this it calls the scheduling algorithm, which will halt where necessary like the storage allocation algorithm, and return control when it has had the total of processor time requested.

After the second disc request the store is released and the job calls the subroutine CHR which checks to see if there are too many jobs running as described earlier. Off-line jobs then go back to the beginning. On-line jobs delay for an exponentially distributed time corresponding to "thinking", as this is usually in a pure procedure no major resources are tied up. They then check once more to see if there are too many jobs running and go back to the beginning.

The scheduling of jobs is probably the most complicated feature to simulate and several different methods were tried before the one finally chosen. The basic difficulty lies in ensuring that the use of processor time is recorded both for scheduling purposes and in order to free a job when it has finished with the processor. The second of these could be solved by an instruction of the form "delay for x while free" and this was implemented with some difficulty at one stage. However it proved inefficient since when the activity is

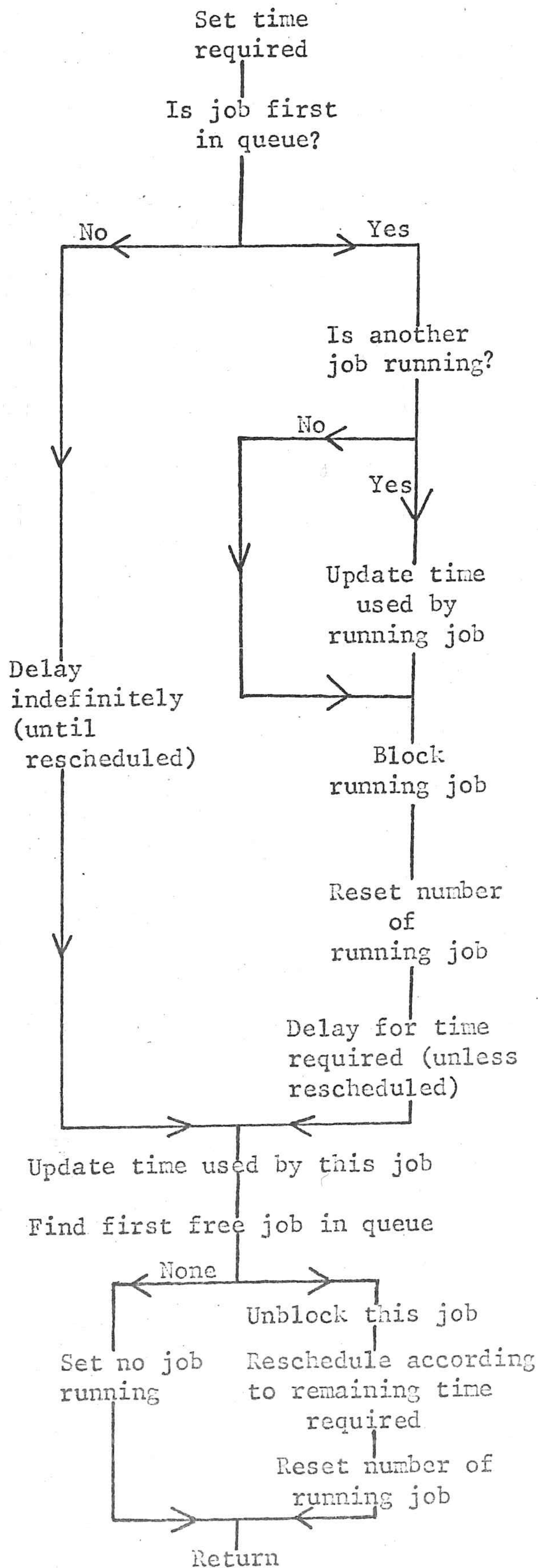
halted the entry corresponding to it in the agenda has to be reordered every time simulation time is advanced, since this also advances the completion time.

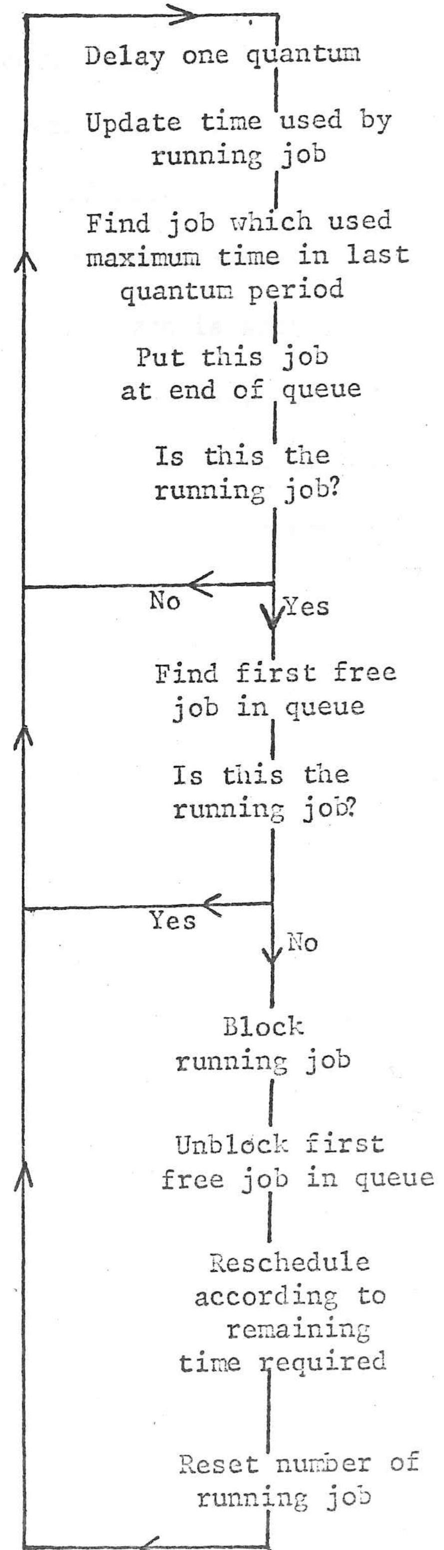
Another approach that was tried and discarded as too inefficient was to make the running job find out each time what the next event was, delay until then and record the amount of processor time used, and then halt until the event was over. When one considers the number of events which may be completely unrelated, such as the arrival of a job or another job finishing its disc or thinking time, the disadvantages of this method become apparent.

A flowchart of the method finally adopted is shown in Figure 7.1. When a job requires processing it calls the subroutine ALG1. This sets the amount of processing time required and looks to see if it is first in the queue. If the job is not first in the queue it delays for a very long period, which in practice means until it is later rescheduled. If the job is first in the queue it looks to see if there is another job running, and if so it calls the subroutine RECT to record the amount of processor time used by the running job and then blocks it. It then marks itself as the running job and delays for the amount of processing time required, though during this time it may be rescheduled by other jobs or the scheduling algorithm. After it has finished it again calls RECT to record the amount of processor time used and looks for the first job in the queue that is free to go. If there is such a job it marks the new one as the running job, unblocks it and reschedules it to continue after the length of time the new job still requires for processing.

Fig. 7.1

Scheduling routine flowchart





The supervisor is represented by the activity ALG2, which runs regularly throughout the simulation at intervals in simulated time equal to the quantum, which is 1.28 seconds since the Titan hardware effectively dictates a multiple of this. Each cycle it calls RECT which updates the amount of comp. time used in the last quantum period by the running job. It then looks to see which job has had the most processor time in the last quantum interval and puts it at the end of the queue. If this was the running job the scheduler looks for the first job in the queue that is free to go. If there is another job free to go the scheduler blocks the previous running job, marks the new one as the running job, unblocks it and reschedules it to continue after the length of time the new job still requires for processing.

There are various possible ways of representing the store of a computer, for example one can have a matrix with the elements of store (in this case units of 4K words) as one dimension and the jobs as the other dimension, or a one dimensional array by jobs or by store. List processing techniques can be used to indicate the free areas of store and their size. However when a job is bound to be in a contiguous area of store the effort of keeping the more subtle structures up to date is not adequately repaid when finding spaces to put jobs in. A simple approach was adopted of having one array element for each element of store and setting this to be -1 if unavailable, 0 if free, and the job INDEX number if in use. There were three basic subroutines, GEST which gets store, LOST which loses it and ISST which when called by either of the other two checks to see if there is sufficient store for a given job and allocates it if there is.

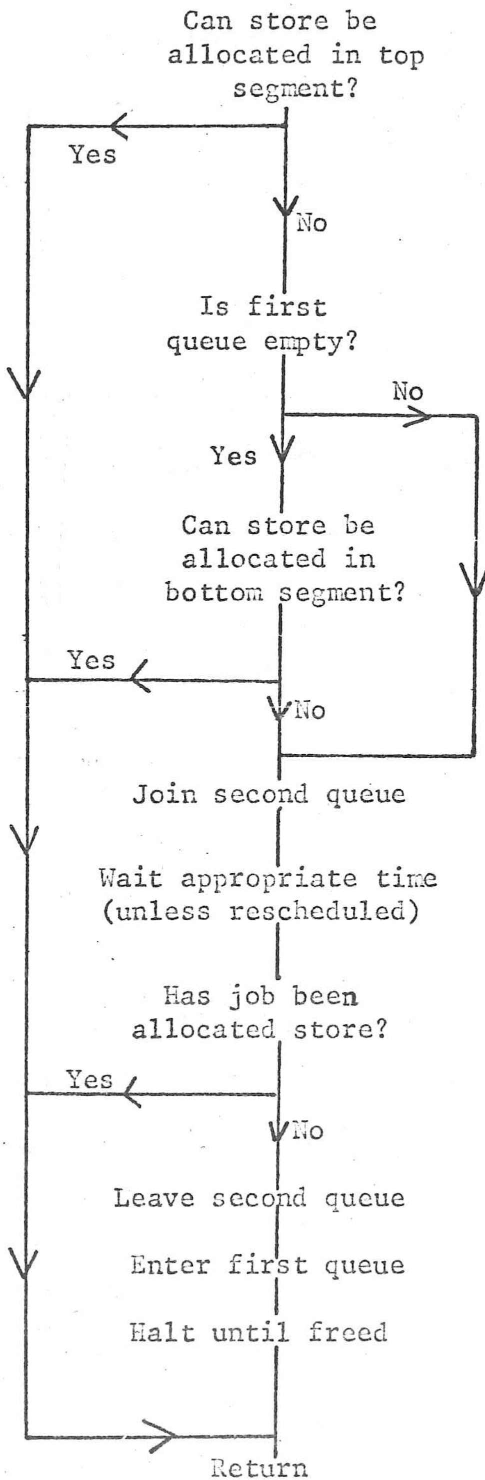
Because the supervisor was originally designed to operate in 64K words of store, the store available to object programs in the Titan with 128K words of store is divided into two regions, 0-40K (bottom segment) and 64K-112K (top segment). The remainder is used by the supervisor for program, working space, buffers and pure procedures. The store allocation algorithm is designed to take account of this fact by allocating the two regions in different ways. There are two queues and initially all jobs join the second queue if they cannot fit in the store. When space becomes available the first job in this queue that will fit is allocated space and so on. If space has still not been allocated after a certain time the job moves to the first queue, which is served preferentially on a first come first served basis, and allocation of the bottom segment stops until the first queue is empty.

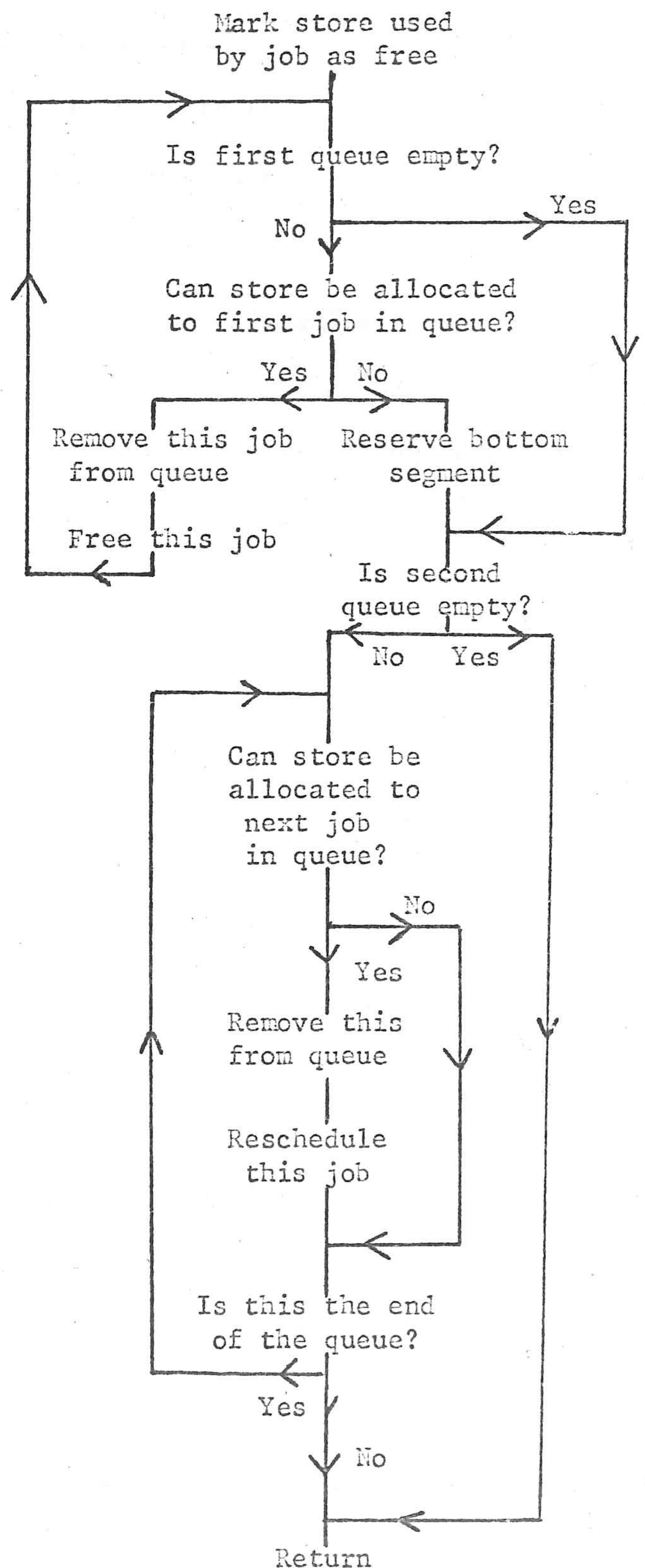
This system ensures that all jobs however large will get run eventually without affecting the response time of smaller jobs too adversely, since the latter can still use the top segment while the large ones are waiting for the bottom segment to empty. Its implementation in the simulation follows a similar pattern, and a flowchart is given in Figure 7.2.

The subroutine GEST which gets store first calls subroutine ISST to check if the job will fit in the top segment, or either segment if the first queue is empty. If this fails it joins the second queue and waits for 200 or 600 seconds according to whether it is an on-line or off-line job respectively. This figure is doubled if it needs more than 16K words of store. If it has not been allocated space in this time it then joins the first queue and is eligible for both segments.

Fig.7.2

Storage allocation flowchart





The subroutine LOST which releases store first searches through and marks all the store used by the job as free. If there are any jobs in the first queue it calls ISST to attempt to find store for them in order using both segments. If all the jobs in the first queue are allocated store it serves the second queue from both segments. If a job in the first queue cannot be allocated store it serves the remaining jobs in order in the first and then the second queues from the top segment only.

The routine ISST to check if a job will fit and allocate store goes through all possible starting points. Jobs greater than 32K words of store must start at the bottom of store to avoid overfilling it and programs of size N must start at an absolute store address which is a multiple of 2^M where M is chosen as the minimum number such that $2^M \geq N$. This is because the base address register is OR-ed with rather than added to the program's address as described in Chapter 2. It selects the highest possible starting point when there is space, if any, and allocates from there. This is in order to keep the lower parts of the store free for larger jobs which must go there.

The object of the first simulation runs was to choose a suitable number of runs and length of run for future use. As already mentioned this is a difficult process and any choice is to some extent arbitrary as there are no good criteria. The amount of processor time used imposes an upper limit, since to avoid taking an unreasonable time and to ensure a better turnround and the possibility of testing on-line, this should be less than five minutes for any run. Since macrogeneration, compilation and assembly took just over a minute, and the ratio of simulated time to real time was about 100:1 depending

on the algorithm used, this restricted the maximum length of run to six hours of simulated time. Six different runs of two simulated hours each were done and the results compared to see how great the range was. The results are given in Table 7.1.

It can be seen that over six runs the range is quite high, up to half the mean value in several cases, though this is often due to an isolated unusual value. The sixth run shows a remarkable correspondance with the mean, however we cannot take it by itself since we would have to do the previous runs anyway to get the initial conditions the same. Taking the first two runs we get a maximum error of 26% and a mean error of 14% with consistently high values of the measurements. Adding the third run, which has values near or below the mean, would improve this to a maximum error of 14% and a mean of 7% but would involve 50% more computing.

It was decided to adopt two runs of two hours simulated time each as the standard for subsequent simulations, this representing a reasonable compromise between the amount of processor time used and accuracy. The comparison given above holds good for the other simulations since the same sets of random numbers were used in each case as far as possible. Having this enables comparisons between runs as well as comparisons of the means for different algorithms, and hence two runs of two hours are preferable to one of four hours. There is little point in splitting it up any further or else one gets overwhelmed with paper and the individual runs become too short, with a resulting increase in variability.

For the simulations the second set of statistics mentioned in Chapter 3 was used. Comparing the results with these statistics we

Table 7.1

Simulation of Cambridge system with 6 runs of 2 hours

Measurement	Run						2 runs		3 runs		6 runs	
	1	2	3	4	5	6	Mean	% error	Mean	% error	Mean	Range
Off-line throughput	2.06	1.40	1.28	1.51	1.25	1.50	1.73	15	1.58	5	1.50	0.78
On-line throughput	4.39	5.54	4.01	3.40	3.77	4.43	4.97	17	4.65	9	4.26	2.14
Off-line processor queue	3.94	3.99	3.44	3.65	3.05	3.73	3.96	9	3.79	4	3.63	0.94
On-line processor queue	0.28	0.45	0.31	0.26	0.29	0.36	0.37	12	0.35	6	0.33	0.19
Off-line processor use*	0.91	0.89	0.92	0.93	0.91	0.90	0.90	11	0.91	0	0.91	0.04
On-line processor use	0.07	0.11	0.08	0.07	0.09	0.09	0.09	0	0.09	0	0.09	0.04
Off-line disc queue	0.23	0.16	0.13	0.14	0.15	0.17	0.20	25	0.17	6	0.16	0.10
On-line disc queue	0.49	0.67	0.41	0.33	0.41	0.46	0.58	26	0.52	13	0.46	0.34
Off-line disc use	0.10	0.07	0.06	0.07	0.06	0.07	0.08	14	0.08	14	0.07	0.04
On-line disc use	0.20	0.26	0.19	0.16	0.19	0.21	0.23	6	0.22	10	0.20	0.10

*Including overhead

see that only 9% of the time (including overhead) was devoted to on-line work instead of 11% in fact, and thus the on-line throughput has fallen from 5.33 phases/min to 4.97 and the off-line throughput has fallen from 1.33 to 1.79. This is probably because in the simulation we mix in large jobs which tend to last a long time fairly randomly, whereas in practice they tend to be run at night when there is little on-line load and thus do not reduce the on-line throughput so much. A facility was included to allow for this by biasing the random numbers to represent night and day running, but this was not pursued because it was not a sufficiently accurate representation.

For this algorithm Table 7.2 shows that the store waiting times increase with program size, as do the maxima. Usually the randomness of the simulation does not produce such a neat result. 32K jobs wait an average of 6 minutes and 40K jobs wait 13 minutes with considerably higher maxima, which is a long time to keep a job waiting around in the machine and suggests that we should try only to have one or two large jobs running at a time. The waiting times of on-line jobs are quite acceptable, the maxima probably being caused by a long off-line job. These maxima should be reduced by swapping or shuffling. The overall disc usage seems low, but it has to cope with the peaks and the figures are comparable with those obtained from the analytical model, when one allows for the higher off-line load and mean comp. time in the second set of statistics on which these results are based.

The first alterations to the basic model that were tried were the effects of 20% variations either way in the means of the off-line and on-line comp. times and the thinking time. This was done partly to see how sensitive the simulation was to these parameters in case a

Table 7.2

Simulation of Cambridge system

	Off-line	On-line
Throughput (phases/min)	1.73	4.97
Processor queue	3.96	0.37
Processor use (inc. overhead)	0.90	0.09
(exc. overhead)	0.75	0.07
Disc queue	0.20	0.58
Disc use	0.08	0.23

Store	Off-line					On-line				
	Queue Mean	Queue Max.	Use	Wait(secs) Mean	Wait(secs) Max.	Queue Mean	Queue Max.	Use	Wait(secs) Mean	Wait(secs) Max.
Pure procedure	-		0.10	-		-		0.56	-	
0-4K	0.04	3	1.17	2	79	0.01	2	0.21	0	21
5-8K	0.06	2	0.77	5	167	0.05	1	0.10	12	399
9-16K	0.50	5	0.89	83	445	0.19	2	0.08	62	389
17-32K	1.04	4	0.99	352	1145					
33-40K	0.41	2	0.28	793	1666					

bad estimate had been made from the data, and partly to see how things varied with changes in the load of this type. The comparison was done using the same stream of random numbers for each of the variables to be sampled as was used for it previously, but multiplying the result by the appropriate factor. The same period was used for each run as before, which meant that the runs were not exactly comparable since the ends of the various streams were at different points and so some phases are not included in all the runs. Unfortunately there is no satisfactory method of getting round this problem. Because of this the waiting times for store have not been included in the results in Table 7.3, since the random variation in these seems to outweigh any other effects.

The first comparison was with a low off-line processor time. Here we notice that the off-line throughput increases as expected from 1.73 to 2.07 phases/min. For a high off-line processor time it falls to 1.22, and the processor use increases slightly.

Changing the on-line processor time makes very little difference to the on-line throughput, since this is not really the bottleneck for on-line jobs. However it affects the distribution of processor time as expected and when it is raised the off-line throughput drops to 1.38, a fall of 20%.

Changing the on-line thinking time affects both the on-line throughput and the processor use in the opposite way as would be expected, since this has a big effect on the amount of on-line work that is done. The reduced off-line throughput for a high on-line thinking time seems surprising, however the detailed results show that this is the average of two very dissimilar runs. These have off-line

Table 7.3

Simulation results with 20% variation in certain parameters

Parameter changed	Off-line Throughput	Off-line Proc. use*	On-line Throughput	On-line Proc. use*
Standard parameter values	1.73	0.90	4.97	0.09
Low off-line comp. time	2.07	0.90	4.75	0.09
High off-line comp.time	1.22	0.91	4.43	0.08
Low on-line comp. time	1.77	0.93	4.83	0.07
High on-line comp. time	1.38	0.90	4.70	0.10
Low on-line think time	1.55	0.89	5.37	0.10
High on-line think time	1.47	0.92	4.41	0.08

*including overhead

throughputs of 2.03 and 0.92 respectively, and for the second run the mean store usage for off-line jobs more than 32K was 0.92, which is exceptionally high and helps to account for the low throughput. Reducing the number of on-line jobs helps the large jobs more than the small ones, since the large jobs tend to get held up the most by on-line jobs.

The first alteration to the basic storage algorithm that was tried was to take the jobs in the first (priority) store queue which had been waiting a long time in a different order by allocating store to the first one in the queue that would fit into the space, and then the next, if any, that would fit into the remainder and so on instead of allocating store strictly first come first served as before. This algorithm will be called First Fitting First Served (FFFS). The danger of this policy is that jobs may arrive in the first queue and get served so quickly that some jobs never get served. However the results in Table 7.4 indicate virtually no difference, a slightly higher throughput for both off-line and on-line jobs because of more efficient use of space, and a longer wait for larger jobs as would be expected.

The next alteration was to combine this with putting on-line jobs directly in the first queue, thus making them eligible for all the store even if there are some other jobs waiting in the first queue. Here the danger outlined before is magnified, and sure enough we see from Table 7.5 that the off-line throughput went right down to 1.47 and the mean waiting times for off-line jobs increased significantly. The on-line throughput is much the same and the

Table 7.4

Simulation with first queue FFFS*

	Off-line	On-line
Throughput	1.79	5.08
Processor queue	3.75	0.40
Processor use (inc. overhead)	0.89	0.10
(exc. overhead)	0.74	0.08
Disc queue	0.20	0.57
Disc use	0.08	0.24

Store	Off-line					On-line				
	Queue Mean	Queue Max.	Use	Wait Mean	Wait Max.	Queue Mean	Queue Max.	Use	Wait Mean	Wait Max.
Pure procedure	-		0.13	-		-		0.54	-	
0-4K	0.07	3	1.27	3	112	0.47	3	0.22	1	109
5-8K	0.10	3	0.65	12	292	0.06	2	0.11	18	310
9-16K	0.55	5	0.72	95	603	0.15	3	0.10	48	280
17-32K	1.08	4	0.67	415	2648					
33-40K	0.52	2	0.45	1058	2845					

*First Fitting First Served

Table 7.5

Simulation with on-line jobs immediately joining first store queue which is FFFS*

	Off-line	On-line
Throughput	1.47	5.10
Processor queue	3.57	0.36
Processor use (inc. overhead)	0.90	0.10
(exc. overhead)	0.74	0.08
Disc queue	0.16	0.56
Disc use	0.07	0.24

Store	Off-line					On-line				
	Queue Mean	Queue Max.	Use	Wait Mean	Wait Max.	Queue Mean	Queue Max.	Use	Wait Mean	Wait Max.
Pure procedure	-		0.15	-		-		0.53	-	
0-4K	0.05	3	1.05	3	130	0.08	2	0.20	0	27
5-8K	0.07	3	0.65	1[illegible]	171	0.06	3	0.10	13	188
9-16K	0.48	5	0.86	97	603	0.08	2	0.08	24	140
17-32K	1.58	5	0.47	668	2468					
33-40K	0.88	2	0.72	1902	4782					

*First Fitting First Served

reduction in their waiting times, though significant, does not seem sufficient to warrant this policy, since these were fairly low anyway.

Another way of serving the queues is Largest Fitting First Served (LFFS). Table 7.6 shows as might be expected that the waiting times for small jobs were increased slightly, the waiting time for 17-32K jobs fell, and the waiting time for 33-40K jobs lay between those for the basic algorithm and FFFS. This algorithm is probably preferable to FFFS since it has a more balanced distribution of waiting times.

A simulation model for swapping was developed, using an image of the store and making alterations to it to decide whether swapping was worthwhile or not, but results with the basic algorithm indicated that if we swapped when an on-line job entered the first queue and could not obtain store this resulted in very little swapping. There is no real point in swapping jobs to make room for off-line jobs, since there is not the same pressure for quick response. Accordingly it was decided not to pursue this line of inquiry further. If the discs were faster or there were magnetic drums it would probably change the balance in favour of more swapping.

Table 7.6

Simulation with both store queues LFFS*

	Off-line	On-line
Throughput	1.78	4.96
Processor queue	4.16	0.40
Processor use (inc. overhead)	0.90	0.09
(exc. overhead)	0.74	0.08
Disc queue	0.20	0.56
Disc use	0.08	0.23

Store	Off-line					On-line				
	Queue Mean	Queue Max.	Use	Wait Mean	Wait Max.	Queue Mean	Queue Max.	Use	Wait Mean	Wait Max.
Pure procedure	-		0.12	-		-		0.54	-	
0-4K	0.04	3	1.26	3	123	0.04	2	0.23	1	136
5-8K	0.21	4	0.91	25	616	0.14	4	0.11	32	280
9-16K	0.68	5	0.91	113	764	0.14	2	0.08	47	379
17-32K	0.81	3	0.67	291	1591					
33-40K	0.39	2	0.50	893	2535					

*Largest Fitting First Served

Chapter 8

CONCLUSIONS

There are two main reasons for doing the type of work described here, to compare and develop techniques and to obtain useful results. Although the initial intention of the research was to obtain useful results, it ended up with the emphasis mainly on techniques, as can be seen from the preceding description. When the work first started the Cambridge system was in an embryonic state and the main need was seen to be for more knowledge about the effects of various factors on system performance, and how it could be improved. However to find this out it was necessary first to learn and then to develop techniques.

There seemed to be more scope for comparing and developing these than for producing results relevant to the Cambridge system, since as time went on the need for results became less and less as experience was gained with the system and it became operational, thus reducing the possibilities for change and experiment. It also seemed better to model the system as it was and see how it worked rather than produce models of hypothetical systems which would never be implemented, or evaluate software changes, which had been tried or could be seen to be unsatisfactory.

By the time results were available there were very few obvious problems outstanding except core store allocation, which is one reason why that has been examined in some detail, though it is also a useful vehicle to demonstrate the simulation and its use. Some other problems which might have been considered were rejected because they would involve major changes to the hardware or software. It is pointless to invent

problems where no real problem is felt, since it is at the system design phase that the results are really needed, not when the system is operational.

We should now consider the value of the various techniques, in the three main categories of Experiment, Analysis and Simulation, and what has been learnt about examining multi-access systems. The most satisfactory source of data on system and user behaviour is an existing system. In this case the system being analysed was sufficiently stable and had enough users to make it a good source of data, but in general an analysis should come during the design or development of a system and this would not be so. For developing systems one must analyse a system with a similar user population, using a batch processing one if no multi-access one is available. By making assumptions about the management policies and charging system one can estimate the potential on-line load.

The system performance is also better obtained from analysing a system with the same or similar hardware than from manufacturer's data. This will give some idea of the overhead and time required to run typical jobs with existing software, and whether the manufacturer's claims are to be trusted.

It is more efficient and convenient if the system contains built-in facilities for collecting various types of statistics, and produces these in a usable form rather than as sheets of paper with detailed figures which require a considerable amount of arithmetic to be of any use. It should be possible to turn this on and off, and preferably to set it to sample at a certain rate to reduce the amount

of data and increase efficiency. It could be made to produce only wanted information, but then there might be a request for some historical information which had been lost. Probably the best solution is to produce the most important information regularly on one stream and have other streams of specialized information on request. The Titan accounting system, which produces the system name (if it is a public system), store sizes and cumulative comp. and exec. times at the end of each phase or where the store size changes, provides a lot of useful information. It would be extremely useful if the disc usage details were also given for each job, at the moment only the totals each minute are given. A software switch has recently been included to cut out the detail when it is not required. It would also be useful if the more important overall statistics such as the means and standard deviations of the comp. times and the throughput in terms of jobs and phases were printed as a matter of course on a daily basis which would give some indication of trends.

As a means of determining the effect of changes to the system experiment is not very practical. It cannot be properly controlled, since it is impossible to repeat an on-line load except on a small scale. Greenbaum[21] used a computer to simulate a set of users and their responses, but this as a very expensive method. The results are only valuable with a relatively stable system and a large load, and then the commitment to the system is too great to start messing around with it. The cost in terms of computer and human time may be great, and it is only practicable for small changes, or ones whose effects can be predicted by intuition or one of the other methods.

Before making any model of a multi-access system it is important to know exactly what information is being sought and what the problem is, particularly for analytical as opposed to simulation models where a wide variety is possible. The process should ideally be treated as an experiment with the models becoming more and more refined as more data becomes available and more insight into the nature of the system being modelled and what the critical factors are.

The emphasis should be on comparison, both between alternative models of the same system and the system itself where possible, and between results with different parameter values and possible hardware or software modifications. The absolute values of the results are useful in giving an insight into the operation of the system, but it is the relative values that will tell us how sensitive the system is to various factors, particularly with simulation where the absolute values are rather unreliable.

It is because of this that there are a wide variety of models for both analysis and simulation depending on the emphasis. As far as analytical models are concerned the main differences are between those that concentrate on an individual job as it passes through a system and determine the distribution of waiting time, and those that concentrate on the system and treat jobs merely as things to be counted. A lot of work has been done on the first type of model, and some examples are given in the bibliography, but even simple scheduling systems require approximations, and more complicated ones cannot be analysed in this way. The only results that can be obtained from these models are about the time spent waiting for processing, and if there are other delaying factors such as disc or core store

queues which are too difficult to analyse this information is not of much use.

The second type of model, which gives results about mean waiting times, queue sizes, resource usage and throughput is much more useful. Here the types of model can be further categorized according to whether we assume that the service times are constant, negative exponential, or a combined distribution. This can be regarded as several negative exponential stages of service, probably with different means, either in series (Erlangian) or in parallel (Hyperexponential) each with certain probabilities, or a mixture of the two.

Deterministic models with constant service time have received little attention in the past, presumably because they were considered too unrealistic. However the results are remarkably similar to those with exponential service time where they are comparable, suggesting they might be of use in more complicated systems which cannot be analysed using Markov models. They are useful for providing a clear indication of when a device becomes saturated and giving simple equations connecting the various quantities which are quite good approximations to more realistic cases. They can also show how sensitive various results are to the distributions by comparing a deterministic model using constant service times with a Markov model using negative exponential distributions.

Markov models with negative exponential distributions which consider the whole system have not received much attention in the past, because they could only deal with a restricted class of systems or else required numerical solution. However extending this technique to include on-line and off-line work, several stages of service and

requests for specific servers and producing analytical solutions has enabled us to deal with a much wider class of multi-access systems, and the assumption of negative exponential instead of constant service times is much more realistic. A lot of information can be gleaned from these models as to the effects of changing the quantity of on-line or off-line work, increasing the processor speed or adding another processor, and changing the mean times. This is much cheaper than simulation and produces more accurate answers if the model is good. It can provide a useful background to simulation by giving values of the most important results and an idea of what the critical factors are. It also provides a good check on the accuracy of a simulation and vice versa. The ideas developed here could be extended to other systems, even possibly to paged systems by regarding the store as another resource for which one has to queue. However they cannot give a picture of store allocation in a system without swapping or paging, and for this reason we must use simulation.

Models with more complicated distributions have been solved numerically, since the number of states increases rapidly as new notional stages of service are added, but it is arguable whether the increase of accuracy justifies the extra labour and time when only an approximate solution to the model is produced and the solution often appears to be insensitive to the distributions.

The swapping model is included primarily to show that even quite complicated systems can often be analysed in a relatively simple way, provided that the appropriate assumptions are made. With a specific system in mind one would be able to derive some useful results from a model of this type, since swapping systems are difficult to simulate.

When we come to simulation the first question is the choice of a language. Often this will be dictated by practical considerations such as what is available on the computers that can be used, or what the analyst is familiar with. However choice of language can be very important from the point of view of efficiency and simplicity and the range of models that are possible. If the time and effort are available and much simulation is to be done, serious consideration should be given to implementing another language, whether new or old if a suitable existing language is not available.

If compatibility is not important the language can be tailored to the model rather than vice versa, and good diagnostics can be built in, which can save hours trying to find obscure errors. If it is based on another language such as ALGOL or FORTRAN, as many existing simulation languages are, a lot of time is saved on development. Useful facilities in the base language are available, including possibly debugging facilities, and it is much easier to learn. However this may well prejudice the design of the language and its efficiency. The most important efficiency features are to have a language that is compiled rather than interpreted, since simulations involve many loops, and to have an event-based rather than an activity-based language, since this avoids all the leading tests that are a feature of the latter. Some of the more recent languages, for example Simula and OPS-4, can be used in either way, which is the ideal solution.

The experience gained in the design and implementation of FOSSIL was useful in understanding the nature and variety of simulation languages and models. The structure and syntax are very simple, some

of the limitations being imposed for efficiency reasons. It was not designed for general use, and hence lacks data structure manipulation facilities, which were not really necessary for this application. The implementation exploits non-standard features that might prove difficult elsewhere, though a language which was implemented entirely by Fortran subroutines would probably be intolerably inefficient. The features most worthy of incorporation in subsequent languages are the ability to begin and terminate activities at will, contained in several existing languages, the interlock and queueing system, the tabulation facilities, and having local variables identified by an index for different activations of the same routine.

Simulation must always be done with care, both to save time and to ensure more accurate results insofar as these can be reconciled. One of the main difficulties is to decide on the right level of detail, and it is here that the results of preliminary analyses can help. There is always more that can be included, for example a user behaviour model or details of the internal workings of the discs or the supervisor, and it is difficult to know when to stop. Randell and Zurcher[51] proposed a modular simulation, with "black boxes" whose workings are not included gradually being replaced by more detailed models. However modularity is very difficult and inefficient to achieve in practice, since different algorithms may imply differences all over the program.

Because of the fact that simulation programs are constantly changing they must be easy to modify and it should not be necessary to modify 10 different programs to test 10 different algorithms. The simulation described in Chapter 7 was one program with switches all

over it to select the different branches for the different algorithms, this is slightly inefficient and impractical for large numbers of alternatives, unless they can be reduced to smaller independent sets of alternatives, but is much easier to modify. Debugging facilities should also be built into the program with switches so that modifications can easily be tested, they can be removed completely for production runs if necessary.

A little effort put into techniques to improve accuracy can bring great savings in time and improved results, and it is important to realise that what is wanted is not a random sample but a typical sample. Tests should also be done to show the sensitivity of the simulation to certain parameters and assumptions.

Having a convenient operational system and output format can make a big difference in the rate of developing simulations and the ease with which the relevant information can be extracted. As mentioned earlier it is comparisons between similar runs that are important, and the eye cannot take in more than about two pages at once, so whatever detail is provided a summary of the important results is vital.

The main scope for future work lies in the study of paged systems probably with several processors. To do this properly would require a considerable amount of effort. A combination of all three of the techniques described above should be used, since each has its own distinctive advantages. The combination is more powerful than relying solely on one technique, and also avoids putting all one's eggs in the same basket.

It is most important that this study should occur at the system design stage, and preferably when the hardware is being designed, so that the total system can be analysed with a wide range of alternatives and hypotheses tested. A study of a fully developed system is of value for developing techniques and furthering understanding of the system. However it is not likely to come up with startling proposals for improvements, because by then there are too many constraints. The person to whom the results could be most useful is the one who decides the basic philosophy of the system, often completely in the dark, rather than the software expert tinkering about with the scheduling algorithm, who by the time the system is operational usually has a fairly good idea of what is going to happen anyway.

UNIVERSITY LIBRARY CAMBRIDGE

Bibliography

Note The references are given in alphabetical order of author and the authors of an article or book are also in alphabetical order.

The following abbreviations are used:

ACM	Association of Computing Manufacturers
AFIPS	American Federation of Information Processing Societies
FJCC	Fall Joint Computer Conference
IBM	International Business Machines
IEE	Institute of Electrical Engineers
IFIP	International Federation for Information Processing
MIT	Massachussets Institute of Technology
SJCC	Spring Joint Computer Conference

1 New developments in simulation
A.P. Amiry and K.D. Tocher
Proceedings of the third conference on Operational Research
1963 p.832

2 File handling at Cambridge University
D.W. Barron, A.G. Fraser, D.F. Hartley, B. Landy and R.M. Needham
Proceedings of the AFIPS SJCC 1967 p.163

3 ML/I User's manual
P.J. Brown
Cambridge University Mathematical Laboratory Nov. 1966

4 The output of a queueing system
P.J. Burke
Operations Research 4, Dec. 1956 p.699

5 Writing simulations in CSL
J.N. Buxton
Computer Journal Vol.9 No.2 Aug. 1966 p.137

6 Control and Simulation Language
J.N. Buxton and J.G. Laski
Computer Journal Vol.5 No.3 Nov. 1962 p.194

7 Systems performance evaluation, survey and appraisal
P. Calingaert
Communications of the ACM Vol.10 No.1 Jan. 1967 p.12

8 Extended Control and Simulation Language
A.T. Clementson
Computer Journal Vol.9 No.3 Nov. 1966 p.215

9 Stochastic models of multiple and time-shared computer operations
E.G. Coffman
University of California Engineering Department No.66-38 June 1966

10 Distribution of attained service in time shared systems
E.G. Coffman and L. Kleinrock
Journal of Computer and Systems Science Vol.1 No.3 Oct. 1967 p.287

11 The Multics system (series of articles)
F.J. Corbato et al.
Proceedings of the AFIPS FJCC 1965 p.185

12 CTSS - A programmer's guide
Ed. P.A. Crisman
MIT Project MAC 1965

13 Simula - An Algol based simulation language
O.-J. Dahl and K. Nygaard
Communications of the ACM Vol.9 No.9 Sept. 1966 p.671

14 A description of the Simscript language
B. Dimsdale and H.M. Markowitz
IBM Systems Journal Vol.3 No.1 1964 p.57

15 An optimization model for time-sharing
D.W. Fife
Proceedings of the AFIPS SJCC 1966 p.87

16 Preliminary investigations in time-sharing simulation
G.H. Fine
System Development Corporation SDC-TM-2203 Jan. 1965

17 Simulation of a time-sharing system
G.H. Fine and P.V. McIsaac
System Development Corporation SDC-TM-2203 Dec. 1964

18 A Markovian model of the University of Michigan executive system
J.D. Foley
Communications of the ACM Vol.10 No.9 Sept. 1967 p.584

19 Probability models for multiprogramming computer systems
D.P. Gaver
Journal of the ACM Vol.14 No.3 July 1967 p.423

20 Evaluating computer systems through simulation
R.P. Goldberg and L.R. Huesmann
Computer Journal Vol.10 No.2 Aug. 1967 p.150

21 A simulation of multiple interactive users to drive a time-shared computer system
H.J. Greenbaum
MIT Project MAC MAC-TR-58 Jan. 1969

22 On-line computation and simulation OPS-3
M. Greenberger et al.
MIT Project MAC 1965

23 A new Monte Carlo technique: Antithetic variates
J.M. Hammersley
Proceedings of the Cambridge Philosophical Society 52 July 1956 p.449

24 Simulation techniques in operational research
J. Harling
Operational Research Quarterly Vol.9 Mar. 1958 p.9

25 Cambridge multiple-access system user's reference manual
Ed. D.F. Hartley
Cambridge University Mathematical Laboratory Nov. 1968

26 The structure of a multiprogramming supervisor
D.F. Hartley, B. Landy and R.M. Needham
Computer Journal Vol.11 No.3 Nov. 1968 p.247

27 GPSS III: An expanded general purpose simulator
H. Herscovitch and T. Schneider
IBM Systems Journal Vol.4 No.3 1966

28 A computer center simulation project
G.K. Hutchinson
Communications of the ACM Vol.8 No.9 Sept. 1965 p.559

29 Computer systems design and analysis through simulation
G.K. Hutchinson and J.N. Maguire
Proceedings of the AFIPS FJCC 1965 p.161

30 Networks of waiting lines
J.R. Jackson
Operations Research Vol.5 No.4 Aug. 1957 p.518

31 On-line simulation
M.M. Jones
Proceedings of the ACM National Conference 1967 p.591

32 Incremental simulation on a time-shared computer
M.M. Jones
MIT Project MAC MAC-TR-48 Jan. 1968

33 Simulation of a multiprocessor computer system
J.H. Katz
Proceedings of the AFIPS SJCC 1966 p.127

34 An experimental model of System/360
J.H. Katz
Communications of the ACM Vol.10 No.11 Nov. 1967 p.694

35 Time-shared systems: A theoretical treatment
L. Kleinrock
Journal of the ACM Vol.13 No.12 April 1967 p.242

36 Certain analytical results for time-shared processors
L. Kleinrock
Proceedings of the IFIP congress Aug. 1968 p.D119

37 SOL - A symbolic language for general purpose systems simulation
D.E. Knuth and J.L. McNeley
IEE Transactions on electronic computers Aug. 1964 p.401

38 Cyclic Queues
E. Koenisberg
Operational Research Quarterly Vol.9 No.1 Mar. 1958 p.22

39 The past, present and future of general simulation languages
J.S. Krasnow and R.A. Merikallio
Management Science Vol.11 No.2 Nov. 1964

40 T3 Fortran reference manual
J. Larmouth
Cambridge University Mathematical Laboratory July 1967

41 MLS - the Titan mixed language system
J. Larmouth and C. Whitby-Stevens
Computer Journal Vol.11 No.3 Nov. 1968 p.256

42 On time structure in Monte Carlo simulations
J.G. Laski
Operational Research Quarterly Vol.16 No.3 Sept. 1965 p.329

43 A proof of the queueing formula $L = \lambda W$
J.D.C. Little
Operations Research Vol.9 No.3 May 1961 p.383

44 Computer simulation: Discussion of the techniques and comparison of languages
J.F. Lubin
Communications of the ACM Vol.9 No.10 Oct. 1966 p.273

45 The design of multiple-access computer systems: Part 2
R.M. Needham and M.V. Wilkes
Computer Journal Vol.10 No.3 Nov. 1967 p.315

46 Computer simulation of computer systems performance
N.R. Nielsen
Proceedings of the ACM National conference 1967 p.581

47 The simulation of time-sharing systems
N.R. Nielsen
Communications of the ACM Vol.10 No.7 July 1967 p.397

48 A status report on Simula
K. Nygaard
Proceedings of the third conference on Operational Research 1963

49 An analysis both theoretical and by simulation of a time-shared computer system
J.P. Penny
Computer Journal Vol.9 No.1 May 1966 p.53

50 Operating systems for digital computers
P.R. Radford
Cambridge University Ph.D. Thesis 1968

51 Iterative multi-level modelling - A methodology for computer system design
B. Randell and F.W. Zurcher
Proceedings of the IFIP conference Aug. 1968 p.D138

52 Markovian models and numerical analysis of computer systems behaviour
R.S. Rosenberg and V.L. Wallace
Proceedings of the AFIPS SJCC 1966 p.141

53 CTSS Technical notes
J.H. Saltzer
MIT Project MAC MAC-TR-16 Mar. 1965

54 An analysis of time-shared computer systems
A.L. Scherr
MIT Project MAC MAC-TR-18 June 1965

55 Time-sharing measurement
A.L. Scherr
Datamation Vol.12 No.4 Apr. 1966 p.22

56 The SDC Time-sharing system: Parts 1 and 2
J.I. Schwartz
Datamation Vol.10 No.11 Nov. 1964 p.28 and No.12 Dec. 1964 p.51

57 The role of digital simulation
P.H. Seaman
IBM Systems Journal Vol.5 No.3, 1966 p.175

58 An analysis of time-shared computer systems using Markov models
J.L. Smith
Proceedings of the AFIPS SJCC 1966 p.87

59 Assessing computer performance
J.M. Smith
Radio and Electronic Engineer Vol.36 No.5 Nov. 1968 p.288

60 The art of simulation
K.D. Tocher
English Universities Press

61 Review of simulation languages
K.D. Tocher
Operational Research Quarterly Vol.16 No.2 June 1965 p.189

62 Further analysis of a computer centre environment
V.L. Wallace and E.S. Walter
Communications of the ACM Vol.10 No.5 May 1967 p.266

63 The design of multiple access computer systems
M.V. Wilkes
Computer Journal Vol.10 No.1 May 1967 p.1

64 Time sharing computer systems
M.V. Wilkes
Macdonald & Co. 1968

65 A model for core space allocation in a time-sharing system
M.V. Wilkes
Proceedings of the AFIPS SJCC 1969 p.265

Appendix

SIMULATION PROGRAM

```
GLOBAL COMMON IP,T2,T3,IV,FA,ISW,OFN,OFS
GLOBAL COMMON THINKTIME,DISCTIME,QUANTUM,OVERHEAD
GLOBAL COMMON STOREQUEUE,STOREWAIT,SQM,SWM
GLOBAL COMMON PROCQUEUE,DISCQUEUE,RN,CU,DU,LB
GLOBAL COMMON DISC,SQ,SN,ST,NB,RU,JA,NQ,US,UP,ND,T0
GLOBAL COMMON EC,CQ,CN,CC,JR,CS,OL,TH,STOREUSE
GLOBAL DIMENSION CQ(40),EC(40),CN(40),CS(40),RU(2),JA(2)
GLOBAL DIMENSION SN(40),ST(28,2),SQ(2),OL(40),TH(4),CU(4),DU(4)
GLOBAL DIMENSION STOREQUEUE(4),STOREWAIT(4)
GLOBAL DIMENSION PROCQUEUE(8),DISCQUEUE(8),SQM(5,4)
GLOBAL DIMENSION STOREUSE(4),UP(2),US(5,2),RN(11),SWM(5,4)
GLOBAL INTEGER CQ,SQ,ST,OL,SQM,OFN,OFS
GLOBAL LONG INTEGER RU,SN
CALL MAIN
END

SUBROUTINE MAIN
DIMENSION ONOF(4),NIDA(3),HILO(2),VARI(4)
EQUIVALENCE (KR,E1), (NR,E2), (LR,E3), (LC,E4), (MR,E5)
PM
CL
SQ = LO(2) - 1
SQ(1) = LO(41)
SQ(2) = LO(41)
CQ = LO(41)
EC = LO(40) - 1
CN = LO(40) - 1
CS = LO(40) - 1
SN = LO(40) - 1
OL = LO(40) - 1
ST = LO(56) - 1
RN = LO(11) - 1
US = LO(10) - 1
SQM = LO(20) - 1
SWM = LO(20) - 1
UP = LO(2) - 1
RU = LO(2) - 1
JA = LO(2) - 1
TH = LO(4) - 1
CU = LO(4) - 1
DU = LO(4) - 1
PROCQUEUE = LO(8) - 1
DISCQUEUE = LO(8) - 1
STOREQUEUE = LO(4) - 1
STOREUSE = LO(4) - 1
STOREWAIT = LO(4) - 1
THINKTIME = 60
DISCTIME = 5.6
QUANTUM = 1.28
OVERHEAD = 0.83
CALL SET (ONOF(1),CHARS(47,38,38,30,44,41,46,37))
CALL SET (ONOF(2),CHARS(47,46,30,44,41,46,37,5))
CALL SET (ONOF(3),ONOF(1))
CALL SET (ONOF(4),ONOF(2))
CALL SET (NIDA(1),CHARS(46,41,39,40,52,5,5,5))
CALL SET (NIDA(2),CHARS(45,37,33,46,5,5,5,5))
CALL SET (NIDA(3),CHARS(36,33,57,5,5,5,5,5))
CALL SET (VARI(1),CHARS(47,38,38,1,35,47,45,48))
CALL SET (VARI(2),CHARS(47,46,1,35,47,45,48,5))
CALL SET (VARI(3),CHARS(52,40,41,46,43,5,5,5))
```

```
      CALL SET (VARI(4),CHARS(36,41,51,35,5,5,5,5))
      CALL MS(1)
      DISC = -1
      DO 8 I = 11,16
8     ST(I,1) = -1
      NB = 28
      BEGIN (JOBGEN,1)
      BEGIN (JOBGEN,2)
      BEGIN (ALG2,0)
      CALL SETDUMP
      CALL FORCEDUMP
      CL0 = CLOCK(0)
      IN = 253 + NOCTAL(1)
      CALL II(IN)
      IF (IN - 254) 17,17,27
17    CALL SI(IN)
      ASSIGN 27 TO I
      CALL TRAP (I,11)
      MR = INTREAD (-1)
      NR = XSIGNF (MR,1)
      LR = INTREAD (-1)
      IP = INTREAD(-1)
      FA = 1
      IF (IP) 70,27,7
70    IP = -IP
      IV = INTREAD (-1)
      FA = REALREAD (-1)
7     CALL TD(T,D)
      DO 59 I = 1,4
      TABLE (STOREWAIT(I),1.,1.,5.,2)
      TABLE (STOREQUEUE(I),1.,1.,5.,-2)
59    TABLE (STOREUSE(I),0.,1.,5.,-2)
      DO 60 KR = 0,NR
      IF (KR) 57,57,58
57    DO 45 I = 1,11
45    RN(I) = 0.083 * I
      DELAY (900.)
      GOTO 60
58    T0 = TIME
      DO 9 I = 1,2
      DO 24 J = 1,5
      SQM(J,I) = 0
24    SWM(J,I) = 0
      TABLE (STOREWAIT(I),1.,1.,5.,2)
      TABLE (STOREQUEUE(I),1.,1.,5.,-2)
      TABLE (STOREUSE(I),0.,1.,5.,-2)
      PROCQUEUE(I) = 0
      PROCQUEUE(I + 4) = 0
      DISCQUEUE(I) = 0
      DISCQUEUE(I + 4) = 0
      CU(I) = 0
      DU(I) = 0
9     TH(I) = 0
      DO 31 LC = -3,2
      CALL RS(LC/3.)
      DO 20 I = 1,11
20    RN(I) = (I + KR)/(KR + 12.)
31    DELAY (10. * LR)
      DO 50 I = 1,2
      DO 51 J = 1,5
```

```
          SQM(J,I + 2) = XMAX0F(SQM(J,I),SQM(J,I + 2))
51        SWM(J,I + 2) = MAX1F(SWM(J,I),SWM(J,I + 2))
          CALL AD(CU,I)
          CALL AD(DU,I)
          CALL AD(TH,I)
          CALL AD(PROCQUEUE,I)
          CALL AD(PROCQUEUE,I+4)
          CALL AD(DISCQUEUE,I)
          CALL AD(DISCQUEUE,I+4)
          CALL MT(STOREWAIT,I)
          CALL MT(STOREQUEUE,I)
50        CALL MT(STOREUSE,I)
          M = 0
          IF (NR) 60,27,71
71        IF (IAND(KR,1)) 41,41,61
61        WRITEOUT 1,16,IP + ISW,T,D
          IF (FA - 1) 40,41,42
40        WRITEOUT 1,43,VARI(IV)
          GOTO 41
42        WRITEOUT 1,44,VARI(IV)
41        DO 10 I = 2 * M + 1,2 * M + 2
          GOTO (52,54,53,54),I
52        WRITEOUT 1,55,KR,LR
          GOTO 54
53        WRITEOUT 1,56,NR,LR
          T0 = 000
54        WRITEOUT 1,5,ONOF(I),60 * TH(I)/(TIME - T0)
          WRITEOUT 1,3,ONOF(I),AQ(PROCQUEUE,I)
          PU = CU(I)/(TIME - T0)
          WRITEOUT 1,18,PU,PU * OVERHEAD
          WRITEOUT 1,4,ONOF(I),AQ(DISCQUEUE,I)
          WRITEOUT 1,19,DU(I)/(2 * (TIME - T0))
          K = RC(STOREQUEUE(I),4)
          L = RC(STOREQUEUE(I),5)
          IF (K - L) 32,32,33
32        IF (IP - 4) 62,33,62
62        WRITEOUT 1,11,ONOF(I)
          MINTAB (STOREQUEUE(I),-4)
          WRITEOUT 1,25,(SQM(J,I),J = K,L)
33        WRITEOUT 1,6,ONOF(I)
          MINTAB (STOREUSE(I),-4)
          IF (IP - 4) 63,10,63
63        K = RC(STOREWAIT(I),4)
          L = RC(STOREWAIT(I),5)
          IF (K - L) 34,34,10
34        WRITEOUT 1,13,ONOF(I)
          MINTAB (STOREWAIT(I),-4)
          WRITEOUT 1,26,(SWM(J,I),J = K,L)
10        CONTINUE
          IF (M - 1/NR) 72,60,72
60        T0 = TIME
          M = 1
          IF (NR) 72,72,61
72        PRINT 15,XINTF(CLO - CLOCK(0)),LO(0)
          CALL UN
27        STOP
3         FORMAT (1X,A8,16H PROCESSOR <QUEUE>,F6.2)
18        FORMAT (15H+ PROCESSOR USE,2F6.2)
4         FORMAT (1X,A8,11H DISC <QUEUE>,F6.2)
19        FORMAT (10H+ DISC USE,F8.2)
```

```
5         FORMAT (1HA,A8,11H THROUGHPUT,F6.2)
6         FORMAT (1X,A8,10H STORE USE)
11        FORMAT (1X,A8,12H STORE <QUEUE>)
13        FORMAT (1H+,A8,14H <WAIT> VS STORE)
15        FORMAT (6H CLOCK,I5,6H STORE,I6,/)
16        FORMAT (10H1/D/6 IP =,I2,2(1X,A8))
25        FORMAT (4H+MAX,I16,4I12)
26        FORMAT (4H+MAX,F20.3,4F12.3)
43        FORMAT (6H+ LOW ,A8,5H TIME)
44        FORMAT (7H+ HIGH ,A8,5H TIME)
55        FORMAT (4HARUN,I2,4H FOR,I4,5H MINS)
56        FORMAT (/,I2,9H RUNS FOR,I4,5H MINS)
          END

          FUNCTION AQ(X,I)
          DIMENSION X(1)
          AQ = X(I)/(X(I + 4))
          END

          SUBROUTINE MT(T,I)
          DIMENSION T(1)
          CALL MERGETAB (T(I),T(I + 2),T(I + 2))
          END

          SUBROUTINE AD(X,I)
          DIMENSION X(1)
          X(I + 2) = X(I + 2) + X(I)
          END

          SUBROUTINE JOBGEN
          DIMENSION A(40)
          IF (A(0)) 7,7,8

7         A(0) = 0.001 * 2
          A(1) = 0.100
          A(2) = 0.097
          A(3) = 0.070
          A(4) = 0.071
          A(5) = 0.149
          A(6) = 0.167
          A(7) = 0.127
          A(8) = 0.167
          A(9) = 0.007
          A(10) = 0.007
          A(11) = 0.008
          A(12) = 0.012
          A(13) = 0.009
          A(14) = 0.002
          A(15) = 0.003
          A(16) = 0.002
          A(17) = 0.001
          A(18) = 0.001 * 2
          A(20) = 0.109 * 2
          A(21) = 0.112
          A(22) = 0.100
          A(23) = 0.096
          A(24) = 0.092
          A(25) = 0.095
          A(26) = 0.095
          A(27) = 0.092
```

```
      A(28) = 0.085
      A(29) = 0.056
      A(30) = 0.031
      A(31) = 0.018
      A(32) = 0.013
      A(33) = 0.004
      A(34) = 0.002
      A(35) = 0.001
      A(36) = 0.001 * 2

8     I = 20 * INDEX - 20
      IF (JA(INDEX)) 1,9,10
10    IF (JA(INDEX) - 20 + 2 * INDEX) 11,12,1
11    K = IR(RN(INDEX),0,1)
      IF (K) 1,12,9
9     IF (RU(INDEX) - JA(INDEX)) 1,2,3
2     DO 6 J = I + 1,18 * INDEX
      IF (OL(J)) 5,5,6
6     CONTINUE
      GOTO 1
5     BEGIN (JOB,J)
      OL(J) = INDEX
3     JA(INDEX) = JA(INDEX) + 1
      GOTO 13
12    JA(INDEX) = JA(INDEX) - 1
13    DELAY (EX(RN(INDEX + 2),A(I + JA(INDEX)) * 500))
      GOTO 8
1     PRINT 14,RU(INDEX),JA(INDEX)
      STOP
14    FORMAT (16H ERROR IN JOBGEN,2I3)
      END

      SUBROUTINE JOB
      DIMENSION BT(40),QC(2)
      EQUIVALENCE (I,OL(INDEX))
      RU(I) = RU(I) + 1
      IF (T2) 14,14,15
15    CALL WT(I)
14    CALL JQ(CQ)
2     CALL SAM
      IF (SN(INDEX)) 3,3,10
10    IF (IP - 4) 21,22,21
21    BT(INDEX) = TIME
      CALL GEST
      S = SM(SN(INDEX))
      W = TIME - BT(INDEX)
      SWM(S,I) = MAX1F(SWM(S,I),W)
      TAB (STOREWAIT(I),S,W)
22    S = SM(SN(INDEX))
      TAB (STOREUSE(I),S,US(S,I))
      US(S,I) = US(S,I) + 1
      GOTO 4
3     TAB (STOREUSE(I),0.,UP(I))
      UP(I) = UP(I) + 1
4     CALL DIS
      CALL UD(PROCQUEUE,QC,I)
      QC(I) = QC(I) + 1
      CALL ALGI
      CU(I) = CU(I) + CM(I)
      CALL UD(PROCQUEUE,QC,I)
```

```
      QC(I) = QC(I) - 1
      TH(I) = TH(I) + 1
      CALL DIS
      IF (SN(INDEX)) 5,5,6
6     S = SM(SN(INDEX))
      TAB (STOREUSE(I),S,US(S,I))
      US(S,I) = US(S,I) - 1
      IF (IP - 4) 20,23,20
20    CALL LOST
23    CALL CHR
      GOTO 2
5     TAB (STOREUSE(I),0.,UP(I))
      UP(I) = UP(I) - 1
      IF (I - 1) 2,2,1
1     CALL CHR
      TT = EX(RN(7),THINKTIME)
      IF (IV - 3) 18,19,18
19    TT = TT * FA
18    DELAY (TT)
      CALL CHR
      GOTO 2
9     FORMAT (2I3,F8.2)
13    FORMAT (I3,I4,F8.2)
      END

      SUBROUTINE ALG1
      EC(INDEX) = CN(INDEX)
      CALL INQF(I)
      IF (I - INDEX) 4,2,4
4     DELAY (99999.)
      GOTO 3
2     CALL RECT
      BLOCK (JOB,JR)
      JR = INDEX
      DELAY (EC(INDEX))
3     CALL RECT
      JR = 0
      CALL INQF(I)
      IF (I) 5,5,6
6     UNBLOCK (JOB,I)
      RESCHEDULE (JOB,I,TIME + EC(I))
      JR = I
5     RETURN
      END

      SUBROUTINE DIS
      DIMENSION QD(2)
      EQUIVALENCE (I,OL(INDEX))
      CALL UD(DISCQUEUE,QD,I)
      QD(I) = QD(I) + 1
      QUEUE (DISC)
      D = EX(RN(I + 4),DISCTIME/2)
      IF (IV - 4) 16,17,16
17    D = D * FA
16    DU(I) = DU(I) + D
      DELAY (D)
      CALL UD(DISCQUEUE,QD,I)
      QD(I) = QD(I) - 1
      RELEASE (DISC)
      END
```

```
      SUBROUTINE UD(A,Q,I)
      DIMENSION A(1), Q(1)
      T = TIME - T0
      A(I) = A(I) + Q(I) * (T - A(I + 4))
      A(I + 4) = T
      END

      SUBROUTINE SM(I)
      GOTO (1,2,3,3,4,4,4,4,5,5),I
1     SM = 1
      RETURN
2     SM = 2
      RETURN
3     SM = 3
      RETURN
4     SM = 4
      RETURN
5     SM = 5
      END

      SUBROUTINE CHR
      I = OL(INDEX)
      IF (RU(I) - JA(I)) 2,2,3
3     RU(I) = RU(I) - 1
      IF (T2) 6,6,7
7     CALL WT(-I)
6     OL(INDEX) = 0
      CALL LQ(CQ)
      FINISH
2     IF (T2) 4,4,5
5     CALL WT(0)
4     RETURN
      END

      SUBROUTINE LQ(Q)
      DIMENSION Q(1)
      INTEGER Q
      J = 0
2     I = J
      J = Q(I)
      IF (J - INDEX) 2,3,2
3     Q(I) = Q(J)
      END

      SUBROUTINE WT(I)
      DIMENSION OT(2)
      IF (I) 2,3,4
2     I = - I
      WRITEOUT 2,6,TIME,- INDEX,XINTF(RU(I)),TIME - OT(I)
      GOTO 5
4     WRITEOUT 2,6,TIME,INDEX,XINTF(RU(I)),TIME - OT(I)
5     OT(I) = TIME
      RETURN
3     WRITEOUT 2,6,TIME,INDEX
      RETURN
6     FORMAT (F8.2,2I4,F8.2)
      END
```

```
      SUBROUTINE SAM
      DIMENSION P(2), PT(11,2), CT(11,2)
      INTEGER PT,P
      IF (FS) 8,8,7
8     FS = 1
      P(1) = 16269
      P(2) = 64856

      PT(1,1) = 2908
      PT(2,1) = 5668
      PT(3,1) = 3592
      PT(4,1) = 1497
      PT(5,1) = 656
      PT(6,1) = 441
      PT(7,1) = 403
      PT(8,1) = 259
      PT(9,1) = 427
      PT(10,1) = 178
      PT(11,1) = 238

      PT(1,2) = 41870
      PT(2,2) = 18042
      PT(3,2) = 2721
      PT(4,2) = 1814
      PT(5,2) = 292
      PT(6,2) = 88
      PT(7,2) = 24
      PT(8,2) = 0
      PT(9,2) = 5

      CT(1,1) = 2.3
      CT(2,1) = 17.5
      CT(3,1) = 20.9
      CT(4,1) = 42.9
      CT(5,1) = 63.4
      CT(6,1) = 56.7
      CT(7,1) = 77.2
      CT(8,1) = 37.3
      CT(9,1) = 74.6
      CT(10,1) = 196.4
      CT(11,1) = 550.0

      CT(1,2) = 0.7
      CT(2,2) = 0.6
      CT(3,2) = 5.4
      CT(4,2) = 4.6
      CT(5,2) = 17.4
      CT(6,2) = 10.9
      CT(7,2) = 28.9
      CT(8,2) = 0
      CT(9,2) = 2.8

7     I = OL(INDEX)
      GOTO (4,5),I
5     J = IR(RN(8),1,P(I))
      GOTO 6
4     J = 1 + P(I) * RND(RN(9),ND)
6     L = 0
      DO 2 K = 1,13 - 2 * I
```

```
      L = L + PT(K,I)
      IF (L - J) 2,3,3
2     CONTINUE
      PRINT 9,I,J,L,P(I)
      STOP
3     SN(INDEX) = K - 1
      CN(INDEX) = - CT(K,I) * LOGF(RND(RN(I + 9),-ND))
      CN(INDEX) = CN(INDEX)/OVERHEAD
      IF (IV - I) 10,11,10
11    CN(INDEX) = CN(INDEX) * FA
10    RETURN
9     FORMAT (4I6)
      END

      FUNCTION RND(X,I)
      J = I + 3
      GOTO (1,2,3,4,5),J
1     RND = SQRTF (RR(X,1))
      RETURN
2     RND = SQRTF(RR(X,0.251,2.249)) - 0.5
      RETURN
3     RND = RR(X,1.)
      RETURN
4     RND = 1.5 - SQRTF(RR(X,0.251,2.249))
      RETURN
5     RND = 1 - SQRTF (RR(X,1))
      END

      SUBROUTINE ALG2
8     TRACE
      DELAY (QUANTUM)
      CALL RECT
      CM = 0
      J = 0
7     I = J
      J = CQ(I)
      IF (J) 3,3,4
4     IF (CM - CS(J)) 6,5,5
6     CM = CS(J)
      K = I
      L = J
5     CS(J) = 0
      GOTO 7
3     IF (CM) 8,8,10
10    CQ(I) = L
      CQ(K) = CQ(L)
      CQ(L) = 0
      IF (T3) 11,11,12
12    CALL WQ(CQ,3)
11    IF (L - JR) 8,14,8
14    CALL INQF(I)
      IF (I - JR) 13,8,13
13    BLOCK (JOB,JR)
      UNBLOCK (JOB,I)
      RESCHEDULE (JOB,I,TIME + EC(I))
      JR = I
      GOTO 8
      END
```

```
      SUBROUTINE INQF(I)
      I = 0
4     I = CQ(I)
      IF (I) 2,2,3
3     IF (EC(I)) 4,4,2
2     RETURN
      END

      SUBROUTINE RECT
      IF (JR) 2,2,3
3     CS(JR) = CS(JR) + TIME - TI
      EC(JR) = EC(JR) - TIME + TI
      TI = TIME
2     RETURN
      END

      SUBROUTINE JQ(Q)
      DIMENSION Q(1)
      INTEGER Q
      J = 0
2     I = J
      J = Q(I)
      IF (J) 3,3,2
3     Q(I) = INDEX
      Q(INDEX) = 0
      END

      SUBROUTINE WQ(Q,N)
      DIMENSION Q(1)
      INTEGER Q
      NL
      I = 0
1     I = Q(I)
      IF (I) 2,2,3
3     WRITEOUT N,4,I
      GOTO 1
2     RETURN
4     FORMAT (1H+,I3)
      END

      SUBROUTINE GEST
      DIMENSION QS(5,2)
      EQUIVALENCE (J,OL(INDEX))
      GOTO (18,18,20,1,18,20,18,18,18,18),IP
20    IF (J - 1) 1,18,7
18    IF (IC(SQ(1))) 1,7,11
7     LB = 1
      GOTO 3
11    LB = 17
3     IF (ISST(INDEX,1)) 2,1,4
2     S = SM(SN(INDEX))
      TAB (STOREQUEUE(J),S,QS(S,J))
      QS(S,J) = QS(S,J) + 1
      SQM(S,J) = XMAX0F(SQM(S,J),XINTF(QS(S,J)))
      GOTO (15,15,19,1,15,19,15,15,15,15),IP
19    IF (J - 1) 1,15,16
15    CALL EQ(SQ(2))
      T = 1000 - 400 * J
      IF (S - 4) 12,8,8
8     T = 2 * T
```

```
12        DELAY (T)
          IF (IQ(SQ(2))) 9,1,10
10        CALL RFQ (SQ(2),INDEX)
          GOTO (16,17,17,1,17,17,16,17,16,17),IP
17        LB = 1
          IF (ISST(INDEX,1)) 16,1,9
16        CALL EQ(SQ(1))
          HALT (INDEX)
9         S = SN(SN(INDEX))
          TAB (STOREQUEUE(J),S,QS(S,J))
          QS(S,J) = QS(S,J) - 1
4         RETURN
1         PRINT 14
          STOP
14        FORMAT (14H ERROR IN GEST)
          END

          SUBROUTINE LOST
          DO 8 L = 1,NB
          IF (ST(L,1) - INDEX) 8,11,8
11        ST(L,1) = 0
8         CONTINUE
          IF (INDEX - 20) 15,18,18
15        OFN = OFN - 1
          OFS = OFS - SN(INDEX)
18        LB = 1
          I = 0
10        I = IC(SQ(1),I)
          IF (I) 1,5,6
6         J = ISST (I,1)
          IF (J) 17,1,9
17        GOTO (7,10,10,1,10,10,7,10,7,10),IP
9         CALL RFQ (SQ(1),I)
          FREE (I)
          GOTO 10
5         I = 0
          IF (IC(SQ(1))) 1,4,13
13        LB = 17
4         I = IC(SQ(2),I)
          IF (I) 1,7,2
2         J = ISST(I,1)
          IF (J) 4,3,3
3         CALL RFQ(SQ(2),I)
          RESCHEDULE (JOB,I,TIME)
          GOTO 4
7         RETURN
1         PRINT 12
          STOP
12        FORMAT (14H ERROR IN LOST)
          END

          SUBROUTINE RFQ(Q,K)
          DIMENSION Q(1)
          INTEGER Q
          J = 0
4         I = J
          J = Q(I)
          IF (J - K) 3,2,4
3         IF (J) 1,1,4
2         Q(I) = Q(J)
```

```
          RETURN
1         CALL PQ(Q,0)
          PRINT 5,K
          STOP
5         FORMAT (I3)
          END

          SUBROUTINE EQ(Q)
          DIMENSION Q(1)
          INTEGER Q
          J = 0
2         I = J
          J = Q(I)
          IF (J) 1,3,4
4         GOTO (2,2,2,1,5,5,2,2,2,2),IP
5         IF (SN(J) - SN(INDEX)) 3,2,2
3         Q(I) = INDEX
          Q(INDEX) = J
          RETURN
1         PRINT 6
          STOP
6         FORMAT (12H ERROR IN EQ)
          END

          FUNCTION IQ(Q)
          DIMENSION Q(1)
          INTEGER Q
          I = 0
4         I = Q(I)
          IF (I - INDEX) 3,2,4
3         IF (I) 1,1,4
1         IQ = -1
          RETURN
2         IQ = 1
          END

          FUNCTION ISST(I,ME)
          ISST = -1
          J = SN(I)
          GOTO (11,11,11,11,11,11,11,11,17,17),IP
17        IF (I - 20) 19,11,11
19        GOTO (10,11),ME
10        IF (OFN - 3) 12,15,15
12        IF (OFS + J - 18) 11,11,15
11        K = MB(J)
          IF (K - 16) 7,8,8
7         N = NB - K + 1
          GOTO 9
8         N = 1
9         DO 2 L = LB,N,K
          DO 3 M = L,L + J - 1
          IF (ST(M,ME)) 2,3,2
3         CONTINUE
          ISST = L
          GOTO (18,18,18,18,18,18,2,2,18,18),IP
2         CONTINUE
          IF (ISST) 15,18,18
18        DO 4 M = ISST,ISST + J - 1
4         ST(M,ME) = I
          GOTO (16,15),ME
```

```
16        IF (I - 20) 14,15,15
14        OFN = OFN + 1
          OFS = OFS + J
15        RETURN
          END

          SUBROUTINE PRST (N)
          WRITEOUT N,4,(ST(I,1),I = 1,10),(ST(I,1),I = 17,28)
3         RETURN
4         FORMAT (4I3,I4,3I3,I4,I3,3(I4,3I3))
          END

          FUNCTION MB (I)
          MB = 1
2         IF (MB - I) 3,4,4
3         MB = MB + MB
          GOTO 2
4         RETURN
          END

          SUBROUTINE PQ(Q,J,N)
          DIMENSION Q(1)
          INTEGER Q
          IF (Q(0)) 2,2,6
6         WRITEOUT N,5,J
          I = 0
1         I = Q(I)
          IF (I) 2,2,3
3         WRITEOUT N,4,I
          GOTO 1
2         RETURN
4         FORMAT (1H+,I3)
5         FORMAT (I2,4X)
          END
```

UNIVERSITY
LIBRARY
CAMBRIDGE