AN ANALYSIS OF TIME-SHARED COMPUTER SYSTEMS

ALLAN LEE SCHERR

RESEARCH MONOGRAPH NO. 36
THE M.I.T. PRESS, CAMBRIDGE, MASSACHUSETTS

For Herb Teager

FOREWORD

This is the thirty-sixth volume in the M.I.T. Research Monograph Series published by the M.I.T. Press. The objective of this series is to contribute to the professional literature a number of significant pieces of research, larger in scope than journal articles but normally less ambitious than finished books. We believe that such studies deserve a wider circulation than can be accomplished by informal channels, and we hope that this form of publication will make them readily accessible to research organizations, libraries, and independent workers.

Howard W. Johnson

ACKNOWLEDGMENTS

This monograph is based on the author's doctoral thesis, written while a Research Assistant with MIT's Project MAC. It is an attempt to apply measurement and evaluation techniques to a relatively new development in the digital-computer systems area: the time-shared, interactive machine. Some of the aspects of the operation of such systems are analyzed with the emphasis on the reaction of the hardware systems to the demands placed upon them by its users. Both time-shared systems and their users are characterized in order that the performance of the two acting together can be predicted. User characteristics are defined and quantitatively described. Simulation models as well as mathematic ones are developed and their accuracy is compared to actual system measurements. All of the measurements were made with the Project MAC "Compatible Time-Sharing System" (CTSS).

First, simulation models are used to study the effects of changing small details in the operation of CTSS-like systems. Throughout, the actual measurements of CTSS are used as a basis for comparison.

The author would like to express his deep appreciation to Professor Herbert M. Teager, who served as thesis supervisor. Thanks are also due to the thesis readers, Professors Ronald A. Howard and Chung L. Liu.

Many friends and colleagues at Project MAC contributed to this research, and, in a very real sense, they were the ores being measured. The author would particularly like to thank Professors D. C. Carroll, R. M. Fano, M. Greenberger, and D. Wilde; Messrs. P. Clermont, P. Denning, G. Everest, G. Gorry, T. Johnson, M. Jones, L. Kanodia, R. Mills, D. Ness, F. Russo, A. Saltalamacchia, L. Selwyn, and M. Wantman. In addition, thanks are also due to G. A. Maley and J. Ever of the IBM Corporation.

Finally, the author would like to express his sincere appreciation to his parents, his aunt, and his wife, Marsha, who was a constant source of inspiration.

A.L.S.

Poughkeepsie, New York
October 7, 1966

PREFACE

Allan Scherr's monograph is an important contribution to the art of quantitative performance evaluation of computing systems. He has shown that a somewhat gross characterization at the level of processing and transmission rates between subsystem "black boxes" can realistically duplicate on a contracted time scale the actions of a real system. His simulation model, when applied to an existing time-shared system, has pinpointed its rate-limiting aspects and predicted its reactions under a wide range of conditions. Evaluation tools such as he has developed are crucial in improving the design of computer systems, yet taken alone they are not sufficient, since knowledge of the load and its probable future changes is a precondition for their use. Aspects subject to future change, such as applications, user population, programming languages, and the input-output (I/O) device interface can dramatically alter the loading and thus the behavior of computer systems. However, it was not part of Scherr's task to treat these influences explicitly.

The author of this preface is a systems engineer whose abiding professional interest is the development of better computer-information processing systems. His intentions are, first, to give his reasons for writing; second, to consider briefly facets that can affect system behavior relative to a set of performance criteria; and finally, in the light of these criteria, to discuss some of the commonly held notions with respect to time-shared computer systems.

My reasons for writing thus go beyond simply evaluating Scherr's efforts and placing them in a larger context. Like most other professionals in the field -- scientists, engineers, technicians, and promoters -- I have my own notions of how to achieve progress. Since I have worked at each of these levels as a designer, programmer, and user, my biases may hopefully be at least somewhat more wideranged than is the usual case. It is my observation and conviction that the following factors are materially slowing progress in realizing the potential of computers:

1. Computer information processing is a systems problem that goes beyond the range of separate disciplines and is more than the sum of their parts. Computer systems are highly complex, and their development requires the concentrated and co-ordinated skills of many professions. A system can be a success from the standpoint of its hardware and programming, yet still be a resounding failure when not meeting the needs of its user or doing so at too great a cost.

2. Computer information processing is not a science and cannot begin to become one without a prior foundation of evaluation and measurement. Neither, for that matter, can systems be improved without a careful study of their defects and strengths relative to a meaningful set of evaluation criteria. A physical system such as a bridge, a power network, or a satellite has some advantages in these regards as compared to information-processing systems. Their failures are glaring and evident to the senses. Their objectives are generally clear, their tangible costs can be ascertained, and corrective measures can be taken. This is not usually the case with computer information-processing systems, which generally involve large numbers of people with the usual distribution of intelligence, knowledge, thoughtlessness, self-interest, and cussedness, to which any and all "temporary" and intangible difficulties with a system can be ascribed.

3. While our understanding of the interaction between programming and hardware aspects of computer systems leaves much to be desired, it is in the area of the needs and capabilities of the user that our ignorance is greatest and most telling. Most professions -- management, science, engineering, medicine, and the military -- have a large and growing need for computers and "intellectual automation." If we are to believe the prognostications, it will not be long before ordinary citizens experience intimate contact with computing services. Yet today all is still far from ideal in the interface between men and machines, and the major impact on most professions is yet to come. A fruitless and frustrating interchange between a citizen and a "mindless" merchandising or government agency computer may rate a humorous squib in the Sunday supplement, but it is symptomatic of current and future difficulties. The interface between men and machines is not yet sufficiently close to human needs and capabilities for us to be comfortable about their

extension into the brave new world to come.

4. There is a communication and semantics problem within the subspecialties of the computer field as well as between the computer specialist and the larger community of professional users. To avoid confusion, the term "professional user" shall be reserved for the engineers, scientists, doctors, managers, and others, whose use of computers, irrespective of their computer training, is ancillary to their major professional occupation. The term "programmer" shall be used to denote the computer-software specialist (technician through scientist), regardless of training and work level, whose primary professional interest is in computation per se. The professional user, regardless of his computer competence, faces a difficult problem in making his influence and observations on computer usage felt. It is all too easy to dismiss his dissatisfactions by attributing them to a lack of knowledge or to a reactionary attitude. Further, within the computer industry there is no single group willing or prepared to accept responsibility for a system's usefulness to a user. There is thus no form of protest open to the user except the nonconstructive, passive one of avoiding computers.

5. There is a basic and deep conflict between the promotional and prestige needs of the computer field and the development of better computer systems. The attitude that any "unfavorable" criticism can stifle and impede progress in "computer" science discourages free inquiry and frank discussion. Multiple-access computing is currently enjoying an adulatory promotional buildup. When such systems were first considered in the late fifties, it was exceedingly difficult to convince people that such systems and their interaction with people were worthy of study. Today, the pendulum has swung to the state wherein "computer utilities," "time-sharing," "Teleprocessing," "multiprocessing," "multiple access," and "machine-aided cognition" are hailed as universal panaceas, although there has been all too little penetrating analysis and evaluation in the interim to support such conclusions.

The hope of bringing about some future change in these factors is my primary reason for attempting to assess the current state of the art in multiple-access systems.

Performance Issues

There is neither the knowledge nor the space to treat all issues bearing on computer usage in depth here. Yet even a superficial consideration of evaluation criteria and effects can aid in delineating the intrinsic limitations of a particular design or usage from those that are subject to influence and change, such as experience, marketing practice, technological advance, and programming emphasis.

Time-shared, multiple-access computing exhibits many of the virtues and defects of other modes of access, such as batch processing and single-user machines. With the exception of some unique applications, it is generally comparable with such access modes. The primary performance criterion of this preface and the accompanying monograph is feasibility, and the fundamental requirement for feasibility, once possibility has been ascertained, is that over-all cost, including time, resources, and effort relative to the benefits, must be advantageous compared with the ever-present alternatives. The performance of all computing systems depends upon at least four strongly interconnected factors: the application and user population; the language and device interface through which communication to the machine is achieved; the internal programming systems that translate user specifications into machine processes; and, finally, the central processing and accessible storage capability of the equipment per se. The observed behavior of a system is largely dictated by the weakest link in the user-language-software-hardware chain. Unfortunately the causal relationships in a system may be far from clear. Effects may be caused by such diverse and ephemeral factors as a manufacturer's marketing policy, production technology, or programming staff and inclinations, over and above any intrinsic limitations.

Let us first consider the key factor of usage and user population, restricting our attention to classes of uses encountered in various areas of scientific research and professional practice. We could make a long list of specific applications involving "intellectual automation" in areas such as weather forecasting, satellite data systems, experiment monitoring, machine-aided design, hospital and library automation, and so on. Each such area has three levels of interest: the varied objective of the system, a common core of information-processing technique, and a residue of psychological

research problems bearing on how to substitute for human ability and where to put the man-machine interface. Our understanding of human capabilities such as sensory perception, concept formation, abstract reasoning, insight, and human judgment is primitive and, while crucial to computer scientists, in the domain of the psychologist.

Let us direct our attention for the moment upon information processing techniques, such as data reduction, simulation or modeling, casual "desk-calculator" computation, on-line experimentation, programming language translation, and data retrieval.

Each of these processes puts its own specific requirements upon the hardware, software, and interface characteristics of a system in the sense of minima on cost, data and computation rates, storage sizes, and interface modes and languages. On-line experimentation, regardless of its specific nature, for example, implies sufficiently high data-rate sensors and internal processing speeds to keep pace with the experiment, with appropriate display and input facilities for the user both to monitor and direct the course of the experiment. Data reduction and program language translation imply the ability to handle the input, output, and internal processing. The rates and costs are determined by what the user is willing to accept. Simulation can cover a broad class of phenomena, some of which can place inordinate demands upon central processing speeds and costs. For example, in the case of atmospheric modeling, the computations required to achieve a single useful "run" may be on the order of 10^{12}, which in turn implies submicrosecond speeds, if results are to be achieved in hours or days. Data retrieval requires an appropriately sized storage whose contents are readily accessible and intelligibly displayed on an acceptable time scale, while a personalized "desk calculator" requires a minimum of an appropriate keyboard and printer text that will accept and produce a usable language.

For any application, these processes are preceded by a period of program development and checkout, which may well require sizable amounts of computation. Thus, applications require the availability of low-cost computation capacity, which is general-purpose enough to handle both the application itself and the debugging of the software.

The user and user population are also important in dictating systems requirements. In a trivial sense, the computation requirements of a group of users go up as their numbers and average demands increase. It is difficult to set limits on the optimum user demand when benefits are compared with costs. Clearly, a computational facility can be justified or expanded if the change increases the job effectiveness of the user population commensurate with the cost. It would be a platitude to point out that a scientist, engineer, or manager might be made more effective if he were relieved of the tedium of "cranking out" the answers to problems whose solution techniques were straightforward but tedious or time consuming. It is also obvious that such a worker can and will shun proffered assistance if it is more troublesome and time consuming to explain what was wanted and to interpret and validate the eventual result than to do it himself, some other way. If the cost in time and effort is acceptable, the process is, to all intents and purposes, subjectively one of "real time" for the user, regardless of the absolute elapsed time.

It is in such real-time applications that a potential user impinges upon the device-and-language interface of a computer system. A computer, in general, is designed to accept a step-by-step procedure rather than a statement of a problem. Further, the general standard of keyboard-symbol communication devices puts further constraints on the languages with which the user can express his requirements. Although programming systems, such as compilers and translators, are generally available to raise the level of language somewhat above a detailed sequence of machine operations, the current common standard of an "algebraic" scientific programming language is still at the level of a detailed sequence of arithmetic and trigonometric operations upon scalar variables.

This means, for example, that to evaluate an integral for which a formula can be expressed in a few inches of writing, a user would have to begin by developing an iterative procedure, then carefully typing out the ten or twenty lines of the process before submitting the procedure to a computer. The result would then have to be carefully checked and interpreted from a table of numbers, and the entire process repeated until the answers were satisfactory.

It should be pointed out that we are not talking here of what is feasible in the area of programming languages and input-output devices, but we wish to indicate only the general level of attainment. It is not an impossible task to generate "problem-oriented" languages, programming systems, and more versatile input-output devices for technical workers. In specific areas, such as highway, circuit, or lens design and analysis, as well as "desk-calculator"-type computing, many companies and laboratories have seen fit to undergo the process of developing languages that allow the user to express his problem succinctly in the language of the field. Some of these have, in addition, been made available on special-purpose, remote-access systems.

Such language facilities are, however, costly and require the intense and continuing attention of professional computer programmers and engineers who understand the user's problem rather than that of the user himself. It is thus no surprise that, faced by a language-and-device barrier and the time lapses of breaking down and checking out a detailed procedure in the available languages, most professionals outside of university computing laboratories choose to work through an intermediate programmer rather than directly with a machine. To our hypothetical user, regardless of his occupation and whether or not he works through an intermediary, the process of working with a machine can still be in "real time" if he is not significantly delayed relative to other alternatives.

Obviously, a shorter "real time" will allow the user to pose better and more pointed questions on subsequent tries and will thus increase his own effectiveness.

Until such time as more appropriate "problem-oriented" languages and devices are developed and come into widespread usage, we can only speak of "man-machine interaction" or "real-time computation" for the currently predominant user, that is, the professional or part-time programmer. It is well, however, to keep in mind that we have been removed one stage from the real consumer. Conclusions drawn from today's experience may well have to be substantially modified if newer communication interfaces shift the center of usage to the professional scientist or engineer.

While "real time" to the ultimate user may have a loose and flexible meaning, to a programmer it has a definable immediacy.

Programming in the current languages requires concentrated and single-minded attention, and a lengthy "turnaround time" between run submission and result availability, such as an hour or a day, can cause a loss of productivity. Further, if the result is in the nature of notification of a trivial clerical error, such as a misplaced comma or bracket, anything short of immediate turnaround can be highly frustrating.

The programmer's needs are thus not nearly so diverse and demanding as those of the ultimate user, and by and large the programmer is at home with the familiar programming devices and languages. Yet, perhaps even more than the user, he requires large amounts of inexpensive computation, available at reasonable turnarounds. Neither batch processing nor the available single-user machines have been able to cope for long with ever-growing usage, and thus ever-lengthening turnarounds. The periodic changes to higher performance equipment, generally accompanied by a learning transient and the reworking of old programs, have bought only short breathing spaces before the problems reoccurred.

It has therefore been the directly affected programmer who has been most strident in his demands for ever larger computation rates, shorter turnarounds, centralized storage, and remote-consoled, multiple-access systems. Even for him, however, we must look far more closeup at the feasible alternatives in order to be sure that the potential advantages are not illusory. In comparing feasibility, let us distinguish between the three modes of computer access: First, there is the computer system, usually "small" and relatively "inexpensive," is operated by a single "on-line" user; that is, one is connected directly to a computer via its manual input-output interface. Such a system may include interface facilities such as keyboards, printers, magnetic tapes, and graphical I/O. Second, there is the "large," centralized "computer center" which processes the user "runs" semi-automatically in the sequence in which they are submitted as separate "batches." Such a system usually includes scores of backup peripherals, such as high speed printers, card readers, magnetic tapes, and plotters, along with distributed card-preparation equipment such as keypunches on which a user prepares his run "deck" off-line. Finally, there is the "remote-access" system with a distributed set of input-output terminals that can be connected to a centralized computing

facility via public or private communication lines. To a user, such a system can be "on-line" in the sense that the processing equipment is capable of directly servicing his requests. It is this last category that is usually called "multiple-access computing," and the processing techniques for providing the necessary time multiplexing for a set of active user programs and requests are called "time-sharing."

Computing through remotely located terminals connected to a centralized single or distributed facility is not new. In high-priority military command and control areas such as air defense, similar systems have been in existence since the beginnings of the modern electronic technology that made reliable, high-performance computers feasible. Similarly, in the commercial world, accurate and timely inventory control, such as centralized airline reservation, was possible only via on-line, remote access. The unique feature of these systems has generally been the necessity of providing distributed access to a large, time-varying data base.

With the exception of such priority needs where viable alternatives were lacking, the acceptance of remote-access centralized computing systems in the scientific and business world has, until recently, been slow and fitful. In part at least, the slow acceptance can be ascribed to a lack of published experience and the tools to evaluate the performance of systems. Both these factors might point to definite advantages that can offset the added costs, unknown pitfalls, and the inertia of habit.

Among the advantages that have been or could be given for time-shared, multiple-access systems, we might consider the following:

1. Computing capacity becomes cheaper and more efficient as the speed, memory size, peripheral makeup, and thus the cost of a system are increased.

2. The only feasible method of reducing turnaround is to utilize multiple access and time sharing.

3. The user finds that multiple access greatly augments his effectiveness.

4. The existence of a large centralized store of programs and data is essential to the user.

5. A centralized, multiple-access system is more versatile, reliable, and flexible than either single-user or batch modes.

None of these statements is true in other than a highly qualified sense. At best they can be restated as a series of tradeoffs. Let us now consider each of them.

1. Relative Cost and Efficiency of Computing Capacity. Until the standardization of low-cost, high-speed transistor circuitry and the development of economical, moderate-speed (microsecond) core memories, it was generally true that the computation costs on commercially available equipment dropped with increasing system size, expense, and complexity. Thus, it was then economically advantageous to centralize capacity on the largest facility that could be afforded, rather than to parcel out the problem to a set of smaller units. Large systems were further equipped with adequate storage peripherals and large memories that made programming simpler and computing rates higher. Finally, the large investment on the part of both manufacturers and users assured that an adequate population of programmers was available to contribute to the common systems programming chores, such as the writing of compilers, etc.

The situation with respect to low-cost systems was far less advantageous. Only low-performance logic and memory was available at low cost, and thus performance fell off very rapidly as costs were reduced. Adequate peripherals and programming systems were generally lacking. Therefore, the over-all performance of such systems was comparably quite poor.

The situation has dramatically changed over the past three years. First, with advances in the technological art, the costs of integrated high-performance circuitry have dropped by an order of magnitude and do not begin to rise until ultra high-speed performance (in the nanosecond range) is required. At that point costs may rise faster than the added performance. Thus it is harder and more expensive to design and build an ultra high-performance machine than one of moderate (microsecond range) capability. Today, a moderate performance processor comparable to a high capacity system of the early '60s with about an even split between logic and memory cost are being produced and sold for on the order of $50,000. Such systems do not possess a huge memory, but with adequate peripheral backup, such as low-cost magnetic tapes, even a few thousand words are quite sufficient for many computing tasks, once the software is made available and accessible.

Many of the data-reduction, on-line experimentation, and programming language-translation tasks do not require a costly, massive, addressable memory and can be far more efficiently accomplished on a small but adequate system. Thus, for example a four-million dollar system need not be significantly faster for such tasks than one of a forty-thousand dollar investment. With the passage of time and the further development of mass-fabrication techniques such as integrated circuits and subsystems, the balance can well shift further to small machines, particularly if production rates are high enough to attract and justify the necessary program support and attention.

A small system puts much of the processing burden upon the user, requiring him to select his tapes, complete his commands, or merely idly await his next move, during which the capacity of the system is not utilized. A larger system, on the other hand, cannot afford the delay of slow manual operations and thus requires additional equipment, which in turn may end up running at a very low-duty cycle. As Scherr's analysis shows, a time-shared system, regardless of the number of processors, can achieve an acceptable level of central processor efficiency only by the addition of costly equipment in the form of high-speed memory to eliminate access bottlenecks. Whether a system is run at an efficiency of 25 per cent usable processing steps or its cost is doubled to run at 80 per cent, the over-all affect is still one of substantially increased computing cost relative to a batch-processing alternative. The selfsame system, when run in a batch-processing mode in off hours, will further have the added overhead disadvantage of unused equipment.

Clearly, there is and will be a population of users whose demands for computing are too small to support even the smallest machine and to whom the cost of a low capability terminal and supporting communication in a time-shared system will be the most attractive alternative.

2. Reduction in Turnaround Time. Turnaround manifests itself in different ways in batch, single-user, and multiple-access systems. A lengthened turnaround is a system's reaction to excess loading relative to a fixed and finite computation rate: that is, too many users with an excessive demand. With a batch system, a user may be kept waiting for long periods of time before his results become available. With a single-user system regulated by a signup, the signups will fill

far faster, and the inattentive user will face longer gaps between his intervals on the machine.

Multiple-access, time-shared systems, as Scherr's analysis shows, are not exempt from the effects of supply and demand. While any number of remote consoles can be potentially connected to a single system, there is an upper limit to the number that should be active at any one time. Beyond that limit, additional users will face the equivalent of a busy signal, while those on line will experience the effects of competing user demands in the form of a much slower machine and longer waits. While these effects can be minimized for a preferred class of user by an administrative policy such as priorities and time allocations, such alleviation is at the expense of the balance of the user population. So long as the system is kept underloaded, that is, the rate limiter of the user time demand is lengthy input and output requests (at 10-15 characters per second that currently can be handled by the available low-grade communication lines and teleprinters), a multiple-access system can significantly reduce the turnaround time to the user. Much of the multiple access experience has been in just such I/O limiting masks as file maintenance, editing, and listings. With larger computing demands, the delays can be comparable to those of any other overloaded facility.

3. Increased Effectiveness. To a computer user, the facts of current usage are a far cry from the popular concept of a "thinking machine" or even of the "industrious dog worker" that can augment the human intellect by rapidly carrying out meaningful tasks once these are propounded. We cannot yet talk to machines, and we are only beginning to write into them, and the machines are generally limited in how they can communicate back. Yet it no longer takes a dreamer to extrapolate from current experiments on small and large single-user machines with graphical I/O devices and languages to see that man-machine interaction can potentially provide technical workers, other than computer specialists, with a potent, real time tool. No one yet knows where to put the split between man and machine tasks or what is needed in the way of language-and-device interfaces to assist the designer or analyst in arriving at the insight he needs to solve his technical problems. The central problem is psychological and, subsequently, one of providing suitable resources to attack the language and device requirements. Until

more is known about such factors, it would be premature to ascribe any special benefits to the mode of computer access alone. Certainly such issues as the closeness of the language to the user's needs and experience and the degree of distraction associated with the mechanics of carrying out a task are as crucial as response time.

As we have already indicated, multiple-access systems may not have any intrinsic virtues for "machine aided cognition" relative to other on-line possibilities, and may well, in the case of high-data-rate graphical I/O facilities and languages suffer from a cost disadvantage due to the added costs of communication lines, buffering equipment, and terminal gear that can easily add up to a cost greater than a similarly equipped computer system of moderate capacity.

A multiple-access system may, however, serve as a catalyst for the development of interactive languages and devices if part of the necessarily large programming staffs are diverted from the more pressing and immediate problems of reproducing batch capabilities on multiple-access machines. For the present, such experience as has been gathered about effectiveness pertains to the usage of programmers or of professionals acting as part-time programmers. Such experience largely parallels that of equivalently loaded small machines and batch facilities when equipped with comparable language facilities. The elapsed time to check out a program can go down by a factor on the order of 5, but both machine time and the user's time are substantially increased relative to off-line processing. No psychological studies have yet been undertaken on the issue, but there must be some minimum limit to turnaround and maximum limit on a session length before fatigue and other factors cause a programmer to lose effectiveness. It might be argued that there is a point at which a user ceases directing the machine and instead begins to react to its insistent replies. Delays, either forced or voluntary, that allow time for thought and reflection might even be beneficial under these circumstances.

4. Centralized Storage of Data and Programs. For a programmer surrounded by files of punched cards and magnetic tapes, a centralized store can appear as an unmitigated blessing. Accordingly, it has appeared as an unavoidable biproduct on most remote-access systems, and the user has been provided with no comparative alternative for storage requirements at his remote

location.

A centralized, short-access time storage file can be an expensive facility (with costs per bit of one to two orders of magnitude greater than off line media) whose finite storage limits, regardless of magnitude, are generally reached far sooner than expected. In such circumstances, management policy must be invoked to limit the space allocated to individual users and to purge the files of relatively "inactive" records periodically. File access then becomes something far short of immediate or convenient for the user, whose definition of "inactive" or "little-used" disagrees with the management's. Further, since a user can no longer scan his programs and data manually without the intervention of the central facility, comparatively simple changes and deletions can invoke additional computer attention in the form of file maintenance and printing.

Access to a large, central program file and library can alternatively be gained on either a batch-processing or a single-user system. Access is after all a relative term and can be achieved by a variety of means, ranging from completely manual to semi-automatic. Certainly paramount issues are both the presence of the program library in some form and knowledge of its existance and details in the mind of the user. Where the development of programs is rapid, the sheer mechanics of dissemination of information and its assimilation by the user can far outweigh the time advantage of rapid program availability. Changes without the user's knowledge are generally worse than no changes at all.

The situation with respect to centralized data files is more advantageous for the remotely accessed system, if the data base is exceedingly large, the data turnover is rapid, and the changes are observed and contributed by a distributed set of users. A large complicated design effort, a management control system, or an information retrieval system can easily be of this nature.

5. Versatility, Flexibility, and Reliability. The major examples of large-scale, multiple-access systems have generally been developed with substantial government support. They are on the whole more versatile and flexible than their batch-processing counterparts with respect to their system commands and varieties of programming languages developed.

Versatility and flexibility in complicated hardware and software systems are, however, generally difficult to achieve and

to maintain in the face of the changing requirements of large numbers of people. Original concepts can become frozen, enshrined, hallowed, and perpetuated because of the complexity and human inertia associated with change, regardless of potential benefits. The volume of programming associated with such systems is far greater than that for other modes of access, and thus is even more costly and resistant to change. A "compatible" system designed around file maintenance, with multiple levels of command, programming, and debugging language tends to remain in its original form, and changes will occur by the slow accretion of additional structure rather than by quantum jumps based upon experimental findings.

This would appear to be the normal double-edged sword of development in general, where a large group can bring large resources to bear on moderate changes which then acquire a permanence and stability, whereas a small group can be far less restricted in scope but is much more limited in depth and permanence of its hardware and software developments.

To the extent that present-day multiple-access computer designers have correctly foreseen and allowed for the probable changes in need, a multiple-access system can be highly flexible. Where the needs are beyond the scope and experience of such designers, present systems may well have safeguards against future change which will inhibit their growth. Computer development is not immune to the psychological blinders which afflict any other in-group which is given the responsibility for both developing and then assessing their own brain child. A prior promotional statement, under these circumstances, can become an "established conclusion," and contradictory evidence can be ignored, or belittled.

With respect to reliability, a large system can be made more failure-proof than a smaller one, at the expense of added facilities. Such failures as inevitably will occur can be more difficult to foresee, diagnose, or correct. They also have more drastic effects on a user who has been left no other alternative.

Conclusions

Despite all the foregoing, it would be incorrect to assume that remote-access systems are destined for premature obsolescence. A good deal of attention is being given to their development by government, universities, and industry. Many industrial users are already contracting for such facilities, and

some service bureaus are offering remote facilities on a leasing arrangement.

In other instances, however, some organizations are rushing into such facilities on a basis of prestige, manufacturer's pressure, and fashionable catchwords, rather than on a considered analysis. One object of this introduction was to indicate that a completely general purpose regional "public utility" form of computing facility with longhaul communication lines attached to unlimited numbers of consoles is neither a completely obvious nor the best solution to all computation problems; there are alternatives that are potentially as good or better.

The next decade will probably see the expansion of medium-scale, limited-purpose, time-shared systems in schools, hospitals, merchandising and industrial establishments within which there are large numbers of users with moderate computing demands for which the cost of transmitting data over private or leased lines is not excessive, where centralized data and program files with rapid access are needed, and, most important, where the devices and languages that are available match the needs of the user population. Scherr has focused some of the attention that is needed to understand the equipment constraints of such systems. It is the user about whom we must learn far more in the coming years: The assumption that we understand his needs, working habits, and human constraints remains the greatest impediment to future progress in this field.

Herbert M. Teager

Cambridge, Mass.
December, 1965

CONTENTS

Chapter 1

INTRODUCTION

Computer systems that are able to serve a single user in a conversational or interactive manner have been in existence for relatively many years. Lately, for economic and other reasons, interactive computer systems have been implemented to serve many users simultaneously. At any given time in the operation of such a system, some portion of the interacting users may require particular programs to be executed. The function of the computer system is to execute these programs in such a way as to provide "reasonable" service to each user's requirement. A widely-used technique for providing this type of service is called "time-sharing" and consists of executing each user's program in some order, for some time, not necessarily to completion. Typically, a particular user's program will be allowed to use a processing unit for a period of time, will be stopped so that another user's program can run, and then at some later time will be continued from the point where it was stopped. In order to be able to continue a program, its status must be saved when it is stopped and restored when it is resumed. At the point in time when one user's program is stopped and another's resumed, the status of the former must be saved and that of the latter restored. This process is called "swapping." In general, the status of a program consists of the state of the processor as well as a copy of every word of the program.

The primary aim of the research reported here is to develop techniques and models for the analysis of a broad class of interactive, time-shared computer systems. The emphasis is on the reaction of a hardware system to the demands made upon it by its users. Segments of the over-all problem include the specification of a model for and the measurement of user behavior, the development and verification of both mathematical and simulation models of time-shared systems, and the specification and measurement of performance metrics for time-shared systems. The user measurements and some of the performance measurements were made on Project MAC's "Compatible

Time-Sharing System" (hereafter referred to as "CTSS;" see Corbato, 1962, and Saltzer, 1965).

The model development is divided into two phases. First, simulation models are used to study in detail the effects of small changes in the operation of CTSS-like systems. Then, continuous-time Markov processes are used to model more general classes of time-shared systems. Throughout, the CTSS measurements form a basis of comparison with the model predictions.

The primary result obtained is that it is possible to model accurately users of interactive computer systems and the systems themselves by means of relatively simple models. Moreover, many of the results obtained and most of the techniques developed are applicable to the analysis of a broad class of interactive systems. This fact is established and discussed in the Conclusions (Chapter 5).

Since human users are an integral part of any interactive system, no real progress can be made in the analysis of the performance of such systems until the behavior of its users is established. The CTSS user characteristics were measured and modeled, and these results are presented in Chapter 2. The user model is represented as being the composite of the models for users working on a "mix" of different tasks. Thus, to a certain extent, models for a type of user other than those studied can be generated by using a different task mix.

The models for the hardware-software time-shared systems are divided into two classes (Chapter 3). First, simulation models are used to study three systems: (1) CTSS, (2) CTSS modified by the replacement of its scheduling procedure by a simple round-robin, and (3) a time-shared system using the CTSS hardware but using some multiprogramming techniques to improve hardware efficiency. Using the operation of these simulation models, detailed measurements are made of several performance parameters and hardware usage (Chapter 4). Where possible, simulation results are also compared with actual CTSS measurements for verification.

The second class is a simple continuous-time Markov model for single- and multiple-processor time-shared systems, derived for the purpose of predicting the mean of one of the performance measures (Chapter 3). The accuracy of the values predicted by these models is established by comparing them with the CTSS

measurements. Then, using the Markov models, several examples of multiple-processor time-shared systems are investigated (Chapter 4).

The Conclusions (Chapter 5) discuss the generality of the techniques and models used as well as the specific results obtained. It is shown that a broad class of systems has been covered by the techniques presented, the extension of these techniques to other systems is discussed, and observations are made about the operation of CTSS-like systems.

It should be stated at this point that nowhere in what follows is discussed the question of whether or not time-sharing is the proper mode of operation for any type of system. Such a debate cannot go very far on a strictly technical level. It does not make sense, for example, to base the evaluation of time-sharing on hardware efficiency, because much of the usage of time-shared hardware is for functions not provided by a conventional system. Any such judgment can be made only on the basis of what is currently unmeasurable: the economic value of the increased function and responsiveness provided by time-shared systems.

Chapter 2

THE CTSS ENVIRONMENT

The CTSS environment consists of approximately 250 users whose individual load on the system varies from nearly zero to the equivalent of several hours of IBM 7094 time per month. The model of this environment consists of a description of what a single user does during an elementary operation at his console, the "interaction." Simply stated, an interaction consists of the user requesting and then receiving service from the system. The events usually forming an interaction are: the user's thinking, typing input at his remote console, waiting for a response from the system, and finally watching output. From the point of view of the time-shared system, the user may be thought of as being in one of two states: either the user is waiting for the system to respond, or the system is waiting for the user. Since the function of a time-shared system is to execute programs at the request of its users, these states may be rephrased as: either the system does or does not have a program to execute for a particular user. An interaction can now be precisely defined as the events occurring between two successive exits from the state in which the system has a program to execute for the user. Figure 2.1 shows a typical interaction with the corresponding activities at the console and internal to the system. Appendix A gives an explanation of the internal status of a user's program.

The primary purpose for the development of a user model is its incorporation into models of complete time-shared systems. The most important requirement for these models is that they reproduce the activities of a real system during several hours of operation. They will be required to reproduce faithfully distributions of service times, hardware usage, and so forth. Fidelity of the model's minute-to-minute operation is of secondary importance.

In this chapter, the user interaction model is completely described, next the inherent compromises with reality in the model are discussed, and finally the limitations on this type of model are outlined. It is assumed that the reader has some

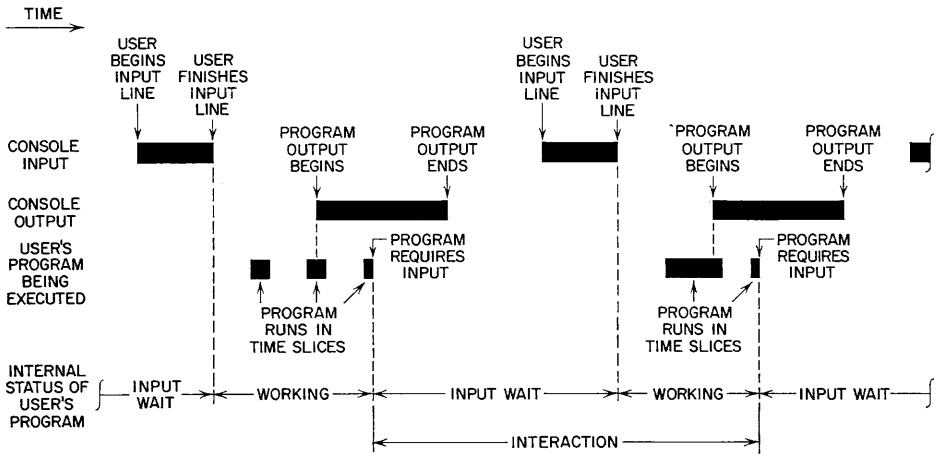

Figure 2.1. Example of a simple interaction.

knowledge of the operation and hardware configuration of CTSS. However, Appendix A describes the operation of this system in sufficient detail to enable understanding of the remainder of this monograph.

The data presented in this chapter were taken primarily between December 29, 1964, and February 4, 1965, during 112 hours of weekday 9:00 A.M. to 5:00 P.M. operation. Approximately 80,000 commands were monitored. The day-to-day changes in the data were slight, and there was stability in the system as well as in user behavior for the duration of the monitoring. A more detailed discussion of these considerations appears at the end of this chapter.

2.1 The Composite Interaction Model

The two states of the user during an interaction have a definite correspondence to the six user states internal to CTSS. The part of the interaction during which the system has a program to execute for the user corresponds exactly to the "Working" state and the "Command Wait" state. This portion of the interaction will be called "the working part of the interaction," and its duration is defined as the response time. The other part of the interaction corresponds to the "Dead," "Dormant," "Input Wait," and "Output Wait" states and will be called "the console

part of the interaction." This name comes from the fact that the time which the user spends in this part of the interaction is determined by console operations; however, since the primary activity during this portion is the user's thinking, this portion will also be referred to as the "think time." Exit from either the Dead or Dormant states is caused by the user's completion of a line of input that is interpreted as a command by the CTSS Supervisory Program. A line of input for the program serving the user terminates the Input Wait period. Output Wait lasts until the console output buffers empty.

The event marking the transition from the working part of one interaction to the console part of the next is the completion of the program serving the user. The nature of this completion determines what happens during the following console portion. The program may require input from the console, it may have attempted to add to an already full output buffer, or it may terminate. In addition, a program may enter the Dormant state for a program-specified period of time. The subsequent transition from the console part to the working part occurs at the end of this "sleeping" period. A program-generated command will be considered as a new interaction with a zero console portion.

Thus, one transition can be tied to the event of a program's finishing, the other usually to an event at the console (that is, end of an input line or output buffers at a certain level). The importance of the other console events, that is, the beginning and end of output, the beginning of input, etc., to the operation of the system is slight. Moreover, the volume of input-output is negligible. Even 50 consoles transmitting or receiving at 15 characters per second yield a total rate of only 750 characters per second. This figure is a maximum; the average rate on CTSS with 30 users is closer to 100 to 150 characters per second. Furthermore, there is no relation of these other console events to anything of importance within the system. For example, output started during the working part of the interaction usually overlaps into the console part. Naturally, higher data rate consoles, based on devices other than keyboards and typewriters, are possible but will not be considered. Appendix B contains statistics describing console input-output. No further separation of console I/O and the time the user spends thinking, etc., will be made; and, as was stated, all of these activities

will be lumped under the term "thinking."

The description of the user during the console part of an interaction consists of simply the elapsed time from start to finish of this portion and the specification of the cause of the last program completion. The time will be specified by a probability distribution. The nature of the program completion marking the beginning of the console portion indicates what the user is doing during this portion. It is necessary to know, for example, whether a new program will be started, that is, a new command, or the old one continued, etc. The distribution of the time a user requires for the console part of the interaction is shown in Figure 2.2. The mean value of this time was determined

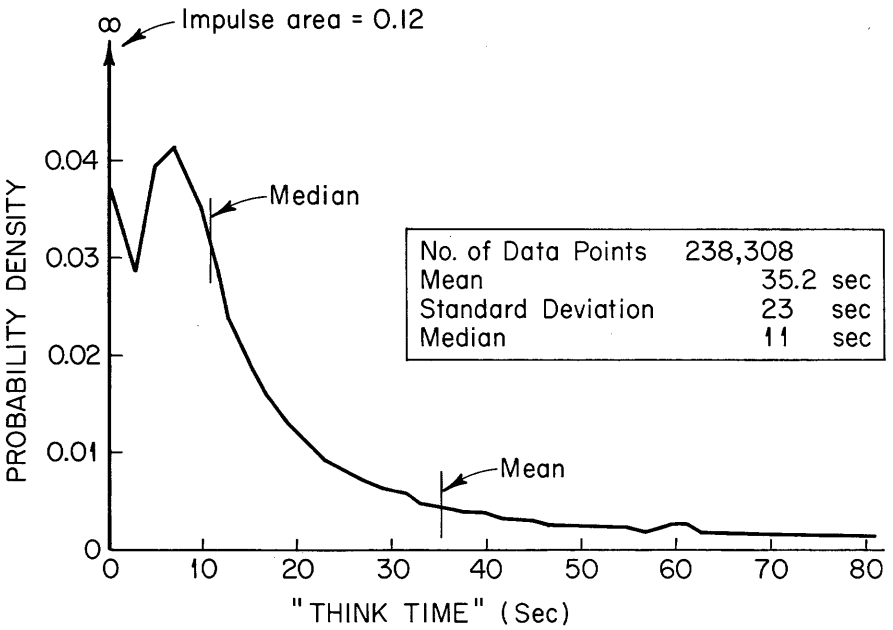

Figure 2.2. Probability density of time for part of interaction (Think Time).

to be 35.2 seconds. Table 2.1 shows the relative probabilities for the various reasons that a user is in the console part of an interaction. These probabilities are generally not independent from interaction to interaction, as will be seen.

TABLE 2.1

Probability of Activity During the Console Part of Interaction

User typing the next command (Dead or Dormant)	.23
User typing program input (Input Wait)	.58
Program waiting for output buffers to empty (Output Wait)	.05
Program "sleeping" (Dormant)	.01
Program-generated command	.12

The time distribution shown in Figure 2.2 can be divided into several different phenomena. The impulse of area .12 at time zero represents the zero console time required for program-generated commands. The high probability between zero and 2 seconds is caused primarily by the easily typed, trivial responses, such as a carriage return only, in Input Wait. A user is in Dead or Dormant (typing a command) for at least 3 seconds. This delay occurs because a user must wait until a "ready" message is typed (10 to 15 characters taking approximately 1 second), and then he must type the several characters constituting the command word. The line of input for Input Wait, however, may consist of only a carriage return character. Users nontrivially responding at their maximum rate cause the second high-probability area at around 7 seconds. Superimposed on these maxima is an extensive, uniformly distributed time caused by the responses that require the user to stop to think, the times that a user is in the Output Wait state, and the periods when the user is temporarily away from his console.

An important statistic describing the nature of the user's activity during the console part of the interaction is whether or not a new command is coming next. In addition, it is necessary to know whether or not the user is in the Dead or Dormant state (i.e., whether or not a core image is being saved on the drum). Figure 2.3 shows the distribution of the number of interactions per command. Note that the probability of having another interaction of a command is not independent of the number of interactions preceding it. The average number of interactions per command is 2.8. The ratio of entries to Dead versus Dormant is .8 to .2.

During the time a user is in the working portion of the interaction, he is loading the system. The amount of time that the user remains in this state is determined by how much of a load the system is already carrying (in the form of other users working) and the amount of work this user requires. A user's loading during the working part of an interaction will be

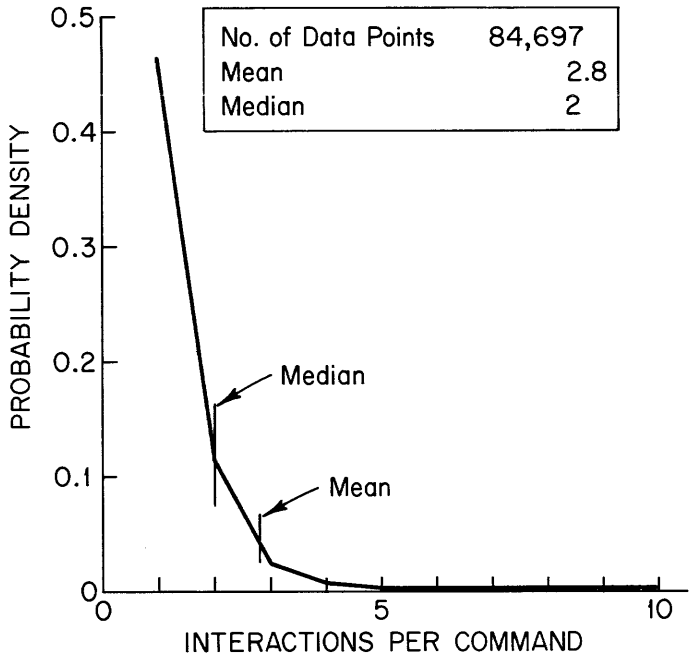

Figure 2.3. Probability density of interaction per command.

described by two parameters: the amount of processor time required by the user's program during the interaction and the size of the program. The latter determines how long it will take for the system to swap the user's program in and out of core memory and/or how many other users' programs are able to fit simultaneously into a core memory with it. Ordinarily, the size of a user's program remains constant for the duration of the command. The user's program may change size in CTSS, but this will be ignored in the model. The probability distribution of program size, shown in Figure 2.4, reflects measurements made at the end of a command, i.e., on the program's entry to either the Dead or Dormant state. The average program size was determined to be 6300 words. The parameter of processor time was chosen to reflect the user's processing requirements because of its simplicity. Since there is no significant overlapping of processor and channel operation in CTSS, the processor runs at a nearly constant rate. The time that a user requires the processor for an interaction includes any overhead computation

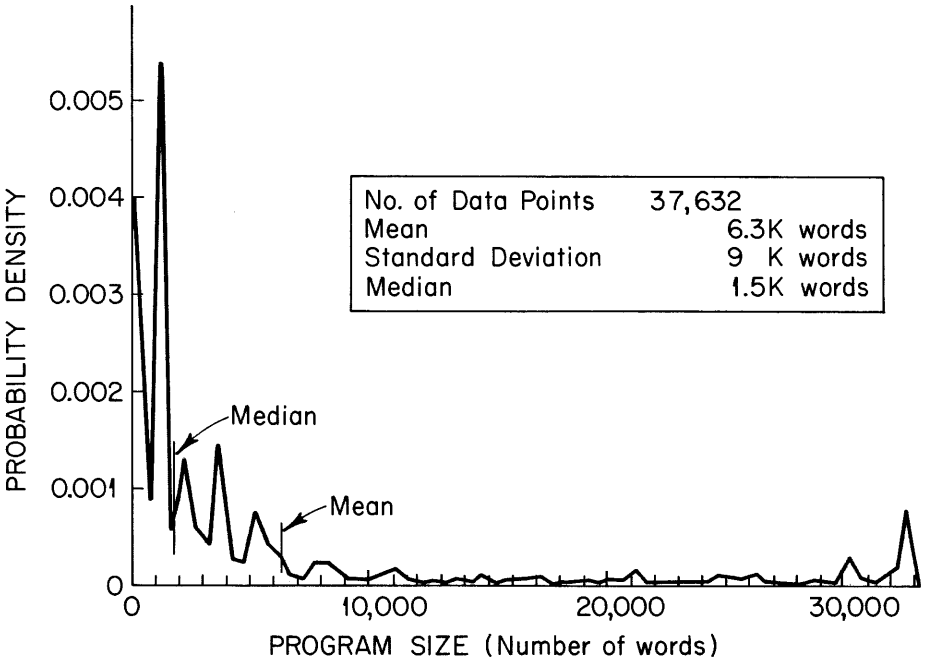

Figure 2.4. Probability density of program sizes.

needed by the system and unoverlapped access and transmission times for the user's program to access the disk storage. Swapping time is not included in the processor time, since it is a function of the time-shared system and not of the user. System overhead includes scheduling and the continuous processing of console input. These functions are almost uniformly distributed, degrading the processor's execution rate by almost a constant. Disk storage is used by the user's programs in much the same way as tapes on a conventional batch-processing system. The breakdown between overhead computation, disk usage, and "useful" computation will be discussed later. The distribution of processor time per interaction is shown in Figure 2.5 and has a mean of .88 second.

There is really no necessity to know anything more about a user's processing needs than the amount of time required. Typically, a program receives control of the processor for about one to four seconds at a time. During this period, several

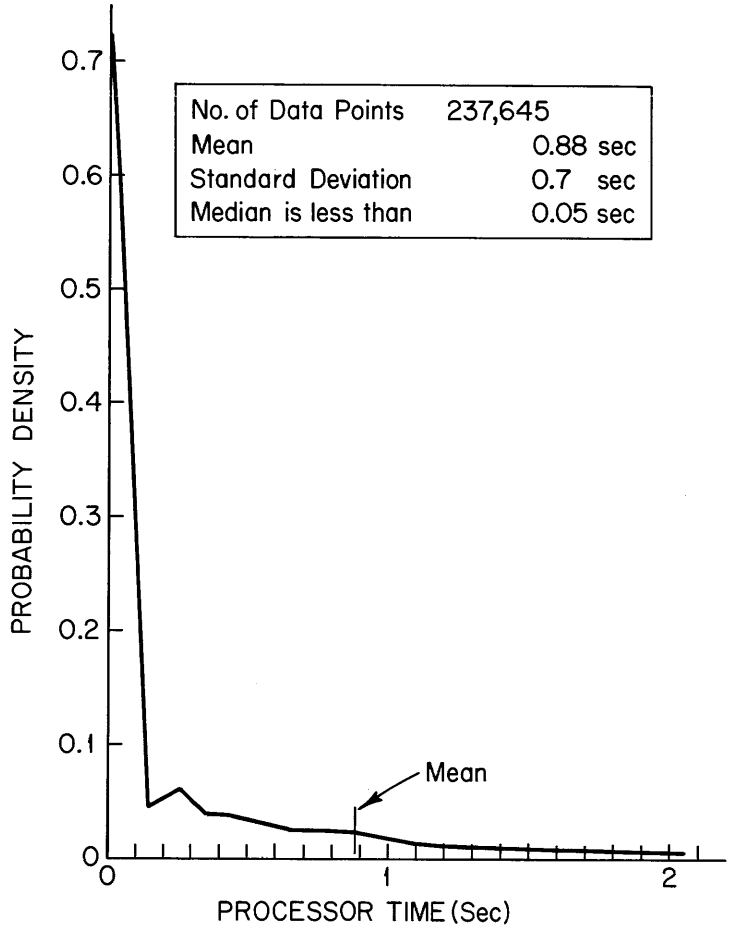

Figure 2.5. Probability density of processor time per interaction.

hundreds of thousands of instructions may be executed. Clearly if the structure of second-to-second operation is not important, no instruction mixes, disk I/O statistics, etc., are required. However, if channel operation overlaps processor operations to the extent that the rate of instruction execution is affected, or if multiple special-purpose processors are used in a time-shared system, information concerning instruction mixes, command types, and so forth, may be desirable. Such considerations will be discussed next.

2.2 Limitations and Possible Extensions of the User Model

In order to develop the interaction model, many simplifications of reality have had to be made. The following paragraphs discuss these and attempt to justify them from CTSS data and other reasoning. Some possible expansions of the interaction model are discussed. In addition, the effects of such simplifications on the behavior of the model are outlined.

The first simplification is to consider all users as equal in the sense that all are representable by the same model. The interaction model tends to average out the differences between individual users. However, since there may be as many as thirty or forty consoles being used simultaneously and since there is a considerable turnover in this user population (approximately 30 per cent per hour), no significant long-term errors should result. The model will also tend to smooth out the effects of users working in bursts. That is, users sometimes have flurries of activity, working at a high rate, followed by periods of comparative inactivity.

The next simplification is that there is no dependence of the model on the total number of users sharing the system. Data show that user loading, as reflected by the time in the console part of the interaction and the processor time per interaction, is relatively insensitive to the number of users sharing the system. Moreover, there has been very little drift in the means of the user parameters during the two months when the data were taken. However, changes in the system tend to affect these user parameters. Such changes will be discussed later.

The choice of the interaction to be the basis for the model can be defended in several ways. First of all, the interaction is an operation of great interest to the user. The response time of the system to a line of input from the user is an important parameter of a time-shared system. In fact, it is one of the few well-defined, measurable performance parameters available. This response time determines the basic rate at which the user can operate. Moreover, data show that the count of the number of interactions per hour is a stable, reliable measure of how much the users or the system are accomplishing. Counts of commands per hour are much less dependable and have greater deviations from the mean. The fact remains, however, that users working on different tasks behave differently. A "task" is defined as the interactions comprising a sequence of commands of the same type

from start to finish. The various purposes of the user's operation can be reflected by a modification of the parameter values of the basic interaction model. Thus, different interaction models can be used for different types of command sequences. In addition, information describing the length of command sequences in interactions per task and/or commands per task is required to complete the specification. In this way, users can be modeled by selecting a task type for each, using the corresponding interaction model for the number of commands or interactions specified by the appropriate distribution, and then switching tasks. The concept of a task is useful for returning some of the fine structure to the operation of the interaction model and for deriving a new composite interaction model with a different "mix" of tasks.

The expansion of the interaction model, as just outlined, will first involve the definition of the task types. The approximately 75 CTSS commands will be divided into five types, and then the parameters for the interaction model for each type will be presented and discussed. The first type of task is (disk) File Manipulation. In terms of conventional batch-processing systems, File Manipulation is usually carried out by hand or with tabulating equipment. The CTSS commands that correspond to this type of task print the contents of a file, combine files, split files, list the names of the files in a user's directory, delete files, copy files, and so on. A complete listing of these commands is given in Appendix B. File Manipulation commands are characterized by usually consisting of a single interaction and requiring little processor time.

The second type of task is that of Program Input and Editing. Commands of this type are used for the generation and modification of disk files that contain the MAD, FAP, FORTRAN, etc., source programs written by the user. These commands are characterized by many interactions and very little processor time per interaction. Commands that generate disk files from console input for purposes such as memoranda, etc., are not included in this category. The third type of task is that of Program Running and Debugging. These commands cause files corresponding to binary decks to be loaded and linked by a conventional BSS loader. Also included are commands to start and stop these programs, to initiate debugging traces of their operation, to examine the registers of a program or the state of the processor,

etc. Commands of this type are characterized by a moderate number of interactions and more processor time per interaction than either of the previously mentioned types. The fourth type is that of Program Compilation and Assembly. These commands generate binary card-image files from source language files, and are characterized by usually having just one interaction and requiring the greatest amount of processor time per interaction.

The fifth and final type is "Miscellaneous" and consists of the unclassifiable commands. Included in this type are the commands that save and resume core images of programs in the process of running. Also included are commands to generate other commands, to cause the commands listed in a disk file to be executed, to generate memoranda, and so forth.

Table 2.2 presents the parameters of the various interaction models for each task and information describing the relative frequency of each type of task. These data were gathered concurrently with the composite interaction model statistics.

The shapes of the individual distributions for the parameters of the different tasks are very similar to those of the composite model.

No data on program sizes were taken to go with Table 2.2, but it is fairly easy to estimate the program sizes of most of the command types. A breakdown of the usage of individual commands is also given in Appendix B.

Since there are differences between the time a user takes to type a command, the time he takes to type a line of input to a program, etc., the question might be asked if the time in the console part of the interaction can be predicted from the mix of typed commands, Input Waits, Output Waits, and so forth. The answer is generally that such a prediction can not be made, although the mix furnishes some indication. For example, it turns out that there are significant differences in the distributions of time in Dead or Dormant for File Manipulation commands and Program Running and Debugging commands. Perhaps the best explanation of these differences is that the user requires different thinking times for different command input lines: some commands are easy to remember and require no arguments; others require a complicated argument string.

The loss of fine structure in the operation of the model amounts to a smoothing of the peaks in user activity. Of course, use of the task model will return some of this structure, but the

TABLE 2.2

Parameters of Various Interaction Models

	File Manipulation	Program Input and Editing	Program Running and Debug.	Assembly and Compilation	Misc.
Command Probability	.36	.15	.12	.09	.29
Interaction Probability	.16	.32	.14	.04	.34
Interactions per Command (Avg.)	1.3	6.3	3.0	1.4	3.4
Avg. Duration of Console Portion of Interaction (Sec.)	52.	33.	38.	25.	29.
Avg. Processor Time per Interaction (Sec.)	1.1	0.4	1.5	3.4	0.6
Prob. of Activity During Console Portion of Interaction:					
Typing Command	.61	.10	.29	.54	.12
Program-generated Command	.18	.06	.04	.16	.18
Input Wait	.02	.84	.61	.16	.65
Output Wait	.18	.00	.05	.04	.03
"Sleeping"	.00	.00	.02	.10	.02
Avg. Interactions per Task	2.8	10.7	5.8	1.7	6.3
Proportion of Processor use	.21	.15	.25	.16	.24
Proportion of user's total time	.22	.27	.18	.06	.28

model is unlikely to reproduce every detail. The effects of users getting "into phase" with each other is still possible with the interaction models but not as likely. This phenomenon, which occurs when many users require service at the same time, is self-perpetuating. That is, as more and more users begin waiting, service for the individual user is reduced; and there is time for more users to reach the point where they require service. Thus, users can fall into step with each other; many working at the same time, many thinking at the same time. With the composite interaction model, this phenomenon will occur with one basic frequency. If the task model is used, a frequency

corresponding to each task type will appear. In reality, there are many subfrequencies to this rise and fall of the number of users in the working part of the interaction.

2.3 The Reliability of the Data

During the period when these data were taken, the hardware configuration of CTSS remained nearly constant. The software system was virtually the same throughout. Several commands were added to the system, but their usage amounted to less than 1 per cent of the interactions. The day-to-day user characteristics did not change very much. During the 47 different time periods when the data were taken, there were only one or two instances of really unusual behavior (e.g., unusually short times for the console portion of the interaction and high processor times per interaction). The average of the means obtained during each period for the console part of the interaction time was 34.2, the median was 34, and the standard deviation was 5 seconds. This compares with the mean time for the console portion for all of the data of 35.0. These figures include data taken during the evening and on weekends! In fact, no significant differences were noted between the user parameters for weekend and evening operation and prime-time operation. The mean time for the console portion for prime time only was 35.2, for evening and weekend operation, 32 seconds. What makes this so surprising is that on nonprime time the average number of users sharing the system was approximately 10, whereas for prime time it was nearly 28. Moreover, there was no measurable correlation between the variation of the times for the console portion of an interaction with time, time of day, or the number of users interacting with the system. For processor time per interaction, the average of the 47 means was .92, the median .93, and the standard deviation .15 second. Here again, these data include both prime time and weekend, and evening operation. This compares with the over-all mean of prime-time operation of .88 second of processor time per interaction and the nonprime-time mean of .94. The biggest difference between prime- and nonprime-time operation turned out to be in the standard deviation of the means from separate periods. The evening and weekend user characteristics tended to be much more erratic. With regard to the mix of interaction types, there was no significant correlation of the probability of a particular type of interaction with time, time of day, number

of users interacting with the system, level of service, and so forth. The standard deviation about the mean probabilities for interactions of types one through five were .16, .16, .28, .31, and .19 of the means, respectively.

These data were taken by a program written to run as part of the CTSS Supervisory Program. The data-taking program was entered each time the Scheduling Algorithm was entered and thus was able to determine the exact time of user state changes. The data gathering added negligibly to the overhead computations of the system and in no other way affected the operations of the users being monitored. State changes took approximately 100 microseconds to process and occurred about 2 to 3 times per second. Typically, the periods monitored amounted to several hours, but several periods were less than 1 hour and some as long as 8 hours. Due to the fact that there was a strict limit to the size of the data-gathering program (roughly 2000 words), some important parameters could not be measured. This is the primary reason why the program-size distribution was gathered during a separate interval.

Overhead computation can be thought of as degrading the effective operating rate of the processor as seen by the user. Overhead computation primarily involves scheduling and the processing of input characters from remote consoles. Every 200 milliseconds, and whenever a character is typed at a console the processor is stopped and control is transferred to the CTSS Supervisory Program for these purposes. Therefore, this overhead can be considered as being uniformly distributed because users are generally interrupted for these functions many times. Rough measurements show that CTSS, operating with 30 users, has an approximate overhead of from 3 to 5 per cent. That is, a program requiring 1 second of IBM 7094 processor time under nontime-shared operation requires approximately 1.05 seconds on CTSS. As previously stated, all processor-time measurements include this overhead. Also included is user programs' use of the disk (not to be confused with loading commands from the disk) as a tapelike input-output device. Data taken over the summer of 1964 by T. Hastings, formerly of the M.I.T. Computation Center Programming Staff, indicate that the average program accesses approximately 3500 disk words per interaction: 2000 words read, 750 words written, and 750 words deleted by reading. The number of words deleted per interaction was estimated by equating it

with the number of words written because the net number of words on the disk remains nearly constant.

2.4 Changes in the System Reflected in User Characteristics

Changes in the time-shared system itself have a definite effect on the behavior of the users. For example, if a heavy scheduling penalty is placed on the users of large programs, the average program size will decrease. In fact, over the summer of 1964 data taken by Hastings indicated that the average program size dropped from 9000 to 6000 words in the space of three months because of just such a scheduling policy. The addition of new commands, which may supplant the functions of the older ones but operate in a different manner, will obviously affect the users' characteristics.

Chapter 3

MODELING TIME-SHARED SYSTEM HARDWARE AND SOFTWARE

This chapter will describe models that represent time-shared hardware-software systems. They will range in generality from CTSS-like systems to idealized, multiple-processor, time-shared systems. Simulation models that represent systems close to CTSS will be used to predict distributions for response times, processor usage, disk usage, saturation, and so on, whereas more general, mathematical models will be used for the prediction of the mean values for some of these parameters. The level of detail incorporated into these models will match that of the interaction model. That is, individual instructions, data words, and disk tracks will not be considered. The models will be based on processor time for an interaction, transmission time for a block of words from disk to core-memory, and so forth.

The first model developed matches CTSS. The very same Scheduling Algorithm and storage-allocation scheme will be used. Next, a simple, first-come, first-served, round-robin* scheduling procedure will be substituted. Then, a model that incorporates multi-programming techniques with the CTSS hardware configuration will be developed. Finally, a simple continuous-time Markov model will be used to represent both single-processor and multiple-processor time-shared systems.

As stated in the Introduction, the first three models to be developed are of the simulation type. Simulation models are required because of the level of detail necessary to handle some of the features studied is beyond the scope of mathematically tractable models. Markov models cannot, in general, be used to represent processes where other than random queueing is used. Queueing Theory models are not usable for processes where the arrival rates of service requests are a function of the service rate. Furthermore, the addition of pre-emptive scheduling

---

* That is, users are entered into a queue in the order in which they enter the working portion of an interaction. The queue is served in a round-robin manner, each queue member getting a slice of processor time in turn.

complicates the mathematics beyond the point where models can even be formulated.

In order that efficient simulations can be written using a convenient notation, a special simulation programming language and operating system were designed. The language itself is fairly straightforward and about as readable as any of the algebraic programming languages. A discussion of the problem will be found later in this chapter, and a complete description of this simulation language and operating system will be found in Appendix C along with listings of the simulation programs for the models presented here.

3.1 The CTSS Model

The CTSS hardware-software simulation model consists of six interacting subprograms:

1. The actual CTSS Scheduling Algorithm (see Appendix A)
2. The Console element (simulating the users' finishing the console portion of the interaction, waiting for the completion of the working part, and so on). This element is based on the data taken in Chapter 2 for the composite interaction model.
3. The Main Control element (informing the Scheduling Algorithm of changes in user status, starting and stopping the processor, and initiating swapping). This does the first function at the direction of the console element, the second and the third at the direction of the Scheduling Algorithm.
4. The Storage Allocation element (directing the detailed operation of swapping). This controls the disk and drum storage elements, causing programs to be loaded and dumped in accordance with the "onion-skin" algorithm used in CTSS.
5. The Processor element. This is started and stopped by the Main Control element.
6. The Bulk Storage element. This simulates both drum and disk storage, controlled by the Storage Allocation element.

The console element is based on the composite interaction model developed in Chapter 2. An array of values indicating the

status of each user is kept. When a user first enters the console part of an interaction, the amount of time he will "think" is drawn from a distribution fitting the one in Figure 2.2. After this time elapses, a signal is sent to the Main Control element placing the user into the Working status. When a "finish" signal is received from the Main Control element, the process is repeated.

The Main Control element has several functions: (1) to enter the Scheduling Algorithm every 200 milliseconds for the basic timing event, (2) to inform the Scheduling Algorithm of user status changes using the six CTSS user states, (3) to control the starting and stopping of the processor and the swapping process under the direction of the Scheduling Algorithm. An informal flow chart of this element is shown in Figure 3.1. Using the distributions presented in Chapter 2, the Main Control element selects for the user an internal state in the console portion of an interaction. Likewise, when a user enters the Working portion of an interaction, a program size and processor time are selected.

The Storage Allocation element is activated whenever a swap is to occur. Given the size and identification of the program to be loaded and a "map" of the current contents of the core memory, this element caused the proper disk and drum activity to load this program. With the use of the CTSS "onion skin" algorithm, only as much of the contents of the core memory is dumped as is required to make room for the incoming program. All dumping is done to the drum. Since all programs are loaded from location zero, there can be only one complete program in core at a time. However, there also may be many partial sections of other programs in core. In CTSS, the number of these partially-dumped programs is limited to four. Naturally, whenever a program is loaded which is already partially in core, only the part that is missing is transmitted from the drum. In addition to the transmission of the core images, each swap is accompanied by the dumping and loading (with the drum) of the processor status and disk file status of the outgoing and incoming programs. This transmission consists of 714 words in each direction.

The Processor element is started by being given the number of instructions to be executed. If it is stopped, it supplies the number of instructions actually executed. If it finishes, it informs the Main Control element. The Bulk Storage element

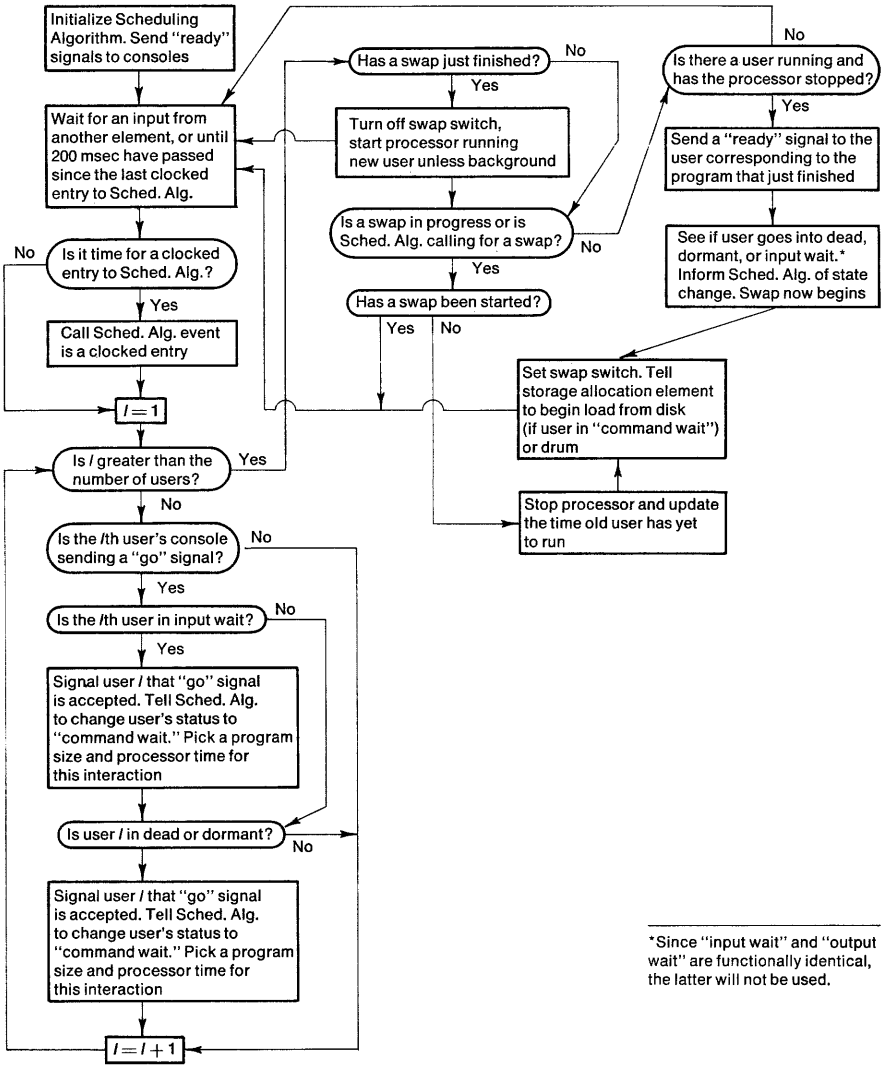

Figure 3.1. Operation of the main control element for the CTSS simulation.

simulates the disk and drum systems. The number of words to be transmitted is supplied along with the type of unit (disk or drum), and the element signals when the transmission is completed. For the drum, a rotational delay uniformly distributed between zero and 17.2 milliseconds is used with a transmission time of 8.4 microseconds per word (IBM 7320 drum). The disk is simulated as having a head positioning delay of either 50, 120, or 180 milliseconds with probabilities of .033, .134, and .833 respectively. The rotational delay is distributed uniformly between zero and 34 milliseconds, and the transmission time is 66.6 microseconds per word. Since up to 9320 words (20 "tracks") may be read without repositioning the read/write heads of the disk, some assumption must be made about the organization of the programs on the disk. Measurements showed that approximately 80 per cent of the programs loaded from the disk come from files that are arranged optimally on the disk, i.e., on adjacent tracks. The remainder of these files come from the user's file area (i.e., they are RESUME class programs, see Appendix B) and were determined to be arranged less optimally, approximately 650 words being obtained per seek.

The interconnection of these elements is shown in Figure

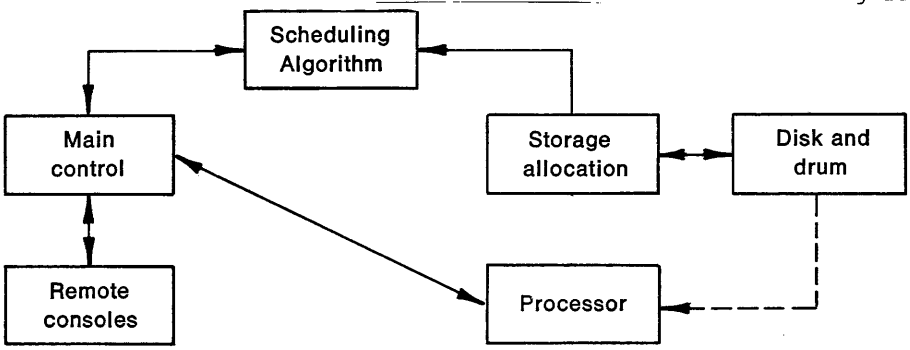

Figure 3.2. Interconnection of elements in CTSS simulation. (Dashed lines show connections for overlapped use of processor and disk or drum.)

3.2. Since CTSS does not overlap the operation of the drum, disk, and processor, there is no interference at the core memory for accesses. Thus there is no need to represent the core memory in the CTSS simulation.

3.2 Variations of the CTSS Model

The first variation to be considered is a change in Scheduling Algorithm. Instead of the multiple-queue arrangement

used by CTSS, a simple, first-come, first-served, round-robin procedure will be used. The time interval during which users' programs are given "bursts" of processor time will be a constant and not variable as in CTSS. The length of the burst is also referred to as the "quantum time" or "time slice." In operation, a list of all of the users in the working state will be kept, and the corresponding programs will be given bursts of processor time in the order in which they are listed. As users' programs finish, they are removed from the list; users whose programs have just received a burst of processor time are removed from the front and placed at the end of this list. In both cases, the remaining programs are moved toward the front. Programs newly entering the working status are added to the end of the queue.

The final model to be simulated will represent a system in which swapping and processor operation are overlapped. While a program is being run by the processor, the program that was running previously is dumped and the next program to run is loaded. Since loading and dumping cannot occur simultaneously, there must be room in the core memory for at least two complete user programs -- the program being executed and the program being dumped or loaded. Should two programs intended to run in sequence not fit together in the core memory, the processor must be stopped to complete the swapping. This procedure is shown in detail in the informal flow chart of Figure 3.3. While a program is running, all or as much as will fit of the next program to run is loaded. After the quantum time is up, the swap is completed if the next program could not be completely loaded; and then that program is started. A dump of the stopped program is then begun, and the process is repeated. The quantum may be extended in order to overlap swapping completely. Scheduling is done strictly on the basis of program size, with a simple algorithm used to sort programs in an attempt to maximize the number of adjacently scheduled programs fitting together in memory.

3.3 The Need for a New Simulation System

There exist dozens of different types of simulation programming systems. Among these are many special-purpose as well as several general-purpose systems. At this time, there exists no special-purpose simulation programming language specifically for use with models of digital computer systems. The general-purpose languages, such as SIMSCRIPT, GPSS, etc., all

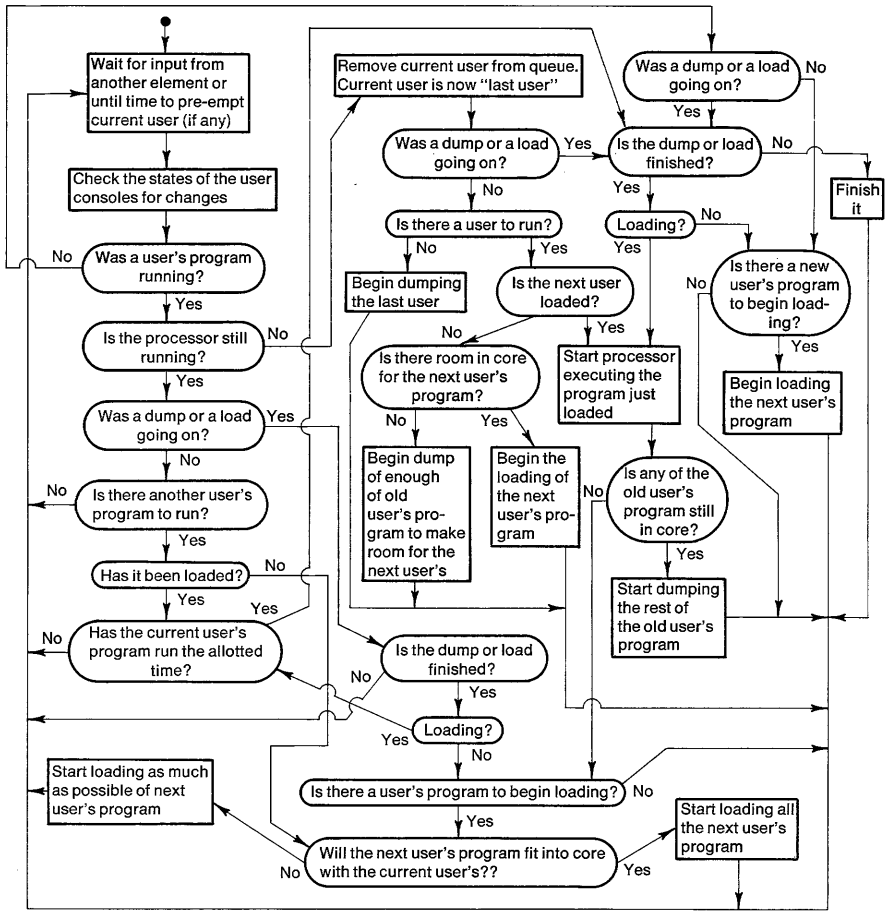

Figure 3.3. Operation of the main control element for a time-shared system with overlapped swapping and processing.

have faults which render them unsuitable for this type of work. Since SIMSCRIPT (MARKOWITZ, 1963) and GPSS (IBM, 1963) are perhaps the most widely used of the general-purpose languages, they will be discussed in more detail. The basic objections to these languages are that: (1) their notation is not convenient for the description of digital systems, (2) they are inefficient in their use of computer time, and (3) their basic timing structure is not well matched to the structure of the digital systems being simulated.

SIMSCRIPT is an event-based language. That is, the simulation is described, event by event, with small programs, one per event. Each event program (or subprogram) must specify the times for the events following it. Conditional scheduling of an event is extremely difficult. The notation is an augmented version of FORTRAN, which is acceptable; but this organization does not take advantage of the modularity of digital systems. SIMSCRIPT's list of coming events may grow to an indeterminately large size, but in a computer system there are only as many coming events as there are independent elements. In SIMSCRIPT it is difficult to distinguish the events associated with a particular physical element. Thus, the modification of such an element is frequently not easily accomplished. Finally, SIMSCRIPT is very difficult to learn, there are no real provisions for automatic tracing and other debugging aids, it is inefficient, and it is not available for on-line use with CTSS.

On the other hand, GPSS is organized around the flow of information through a system. A fairly inconvenient (for the author's purposes) block-diagram notation is used. Again, it is difficult to express and take advantage of modularity in GPSS. It is also extremely inefficient from the standpoint of computer time because it is executed interpretively, i.e., not compiled into machine language. The use of "canned" machine-language routines is therefore difficult. It is available as a command with CTSS but is rarely used, despite the fact that quite a lot of simulation is being done at Project MAC. Furthermore, GPSS is difficult to learn.

As opposed to the above disadvantages, the simulation programming system used here and described in detail in Appendix C is easy to learn, has great modularity, has automatic debugging features, is moderately efficient, and, it is felt, is very well suited to the simulation of digital computer systems. This new

language is now being used by several people at project MAC who learned it in approximately two or three hours of instruction. The language is based on MAD, and each physical element in a system to be simulated corresponds to a conventional MAD subroutine. Built-in traces of the activities of a running simulation can be started or stopped at any time by the programmer, special post-mortems are available to print the state of the simulation at any point, and real-time interaction with the simulation from a CTSS remote console is possible.

3.4 Continuous-Time Markov Model for CTSS-like Systems

A simple continuous-time Markov process will be used to represent the operation of a single-processor, time-shared system. The primary reasons for developing such a model are to compare its predictions with those of the simulation models and with the actual CTSS data and to extend its usage to more complex systems. The basis for this model is the interaction model for the CTSS user, described in Chapter 2. The states of the Markov process representing a system with n users will correspond to the number of users in the working part of the interaction. That is, state j being occupied will indicate that j users are in the working portion of an interaction. Thus, the Markov process will have (n+1) states. In order to use a continuous-time Markov process, the distributions of processor time per interaction and the duration of the console portion of the interaction must be exponential. The mean time for the console portion will be T ; the mean processor time per interaction, P . In this development, P will include the necessary swap time per interaction since in CTSS swapping is not overlapped with computation, and the processor stands idle while swapping occurs. The time-shared aspect of the operation of CTSS will be idealized in that no overhead will be associated with additional users waiting for service. The processor will be considered as switching from program to program at an infinite rate with no loss of efficiency. Thus if there are currently j users waiting for service (i.e., in the working part of an interaction), the rate of exit to the state where (j+1) users are waiting for service is (n-j)/T; in other words, (n-j) users are in the console portion of the interaction. The rate of exit to the state where there are (j-1) users is(1/p). This implies that each of the j users is receiving (1/j) of the processor's

capacity. Thus, each user is finishing at a rate of (1/jp) and any one of the j users is finishing at a rate of (1/p). Equation 3.1 shows the rate matrix (HOWARD, 1960, pp. 92 ff). An expression for the steady-state probabilities is found; and expressions for the average number of users waiting for service and the mean ratio of waiting time to processor time are derived.

$$A = \begin{bmatrix}
-\frac{n}{T} & \frac{n}{T} & 0 & 0 & 0 & \cdots & & & & \\
\frac{1}{P} & -\left(\frac{1}{P}+\frac{n-1}{T}\right) & \frac{n-1}{T} & 0 & 0 & \cdots & & & & \\
0 & \frac{1}{P} & -\left(\frac{1}{P}+\frac{n-2}{T}\right) & \frac{n-2}{T} & 0 & \cdots & & & & \\
0 & 0 & & & & \cdots & & & & \\
0 & & & & & \cdots & & & & \\
 & \cdots & & & & & & & & \\
 & & \cdots & & \frac{1}{P} & -\left(\frac{1}{P}+\frac{3}{T}\right) & \frac{3}{T} & 0 & 0 \\
 & & \cdots & & 0 & \frac{1}{P} & -\left(\frac{1}{P}+\frac{2}{T}\right) & \frac{2}{T} & 0 \\
 & & \cdots & & 0 & 0 & \frac{1}{P} & -\left(\frac{1}{P}+\frac{1}{T}\right) & \frac{1}{T} \\
 & & \cdots & & 0 & 0 & 0 & \frac{1}{P} & -\frac{1}{P}
\end{bmatrix} \tag{3.1}$$

Let

$$\Pi = \{\pi_0, \pi_1, \pi_2, \cdots, \pi_n\} \tag{3.2}$$

be the vector of the steady-state probabilities of occupancy. Then,

$$\Pi A = 0 \tag{3.3}$$

The equations for the π's are

$$-\frac{n}{T}\pi_0 + \frac{1}{P}\pi_1 = 0 \tag{3.4}$$

$$\frac{n}{T}\pi_0 - \frac{1}{P}\pi_1 - \frac{n-1}{T}\pi_1 + \frac{1}{P}\pi_2 = 0 \tag{3.5}$$

yielding

$$\pi_1 = n\frac{P}{T}\pi_0 \tag{3.6}$$

$$\pi_2 = n(n-1)\left(\frac{P}{T}\right)^2\pi_0 \tag{3.7}$$

In general,

$$\pi_i = \frac{n!}{(n-i)!}\left(\frac{P}{T}\right)^i\pi_0 \tag{3.8}$$

Making use of the fact that

$$\sum_{i=0}^{n}\pi_i = 1 \tag{3.9}$$

the following equation results:

$$1 = \pi_0\left[1 + n\frac{P}{T} + n(n-1)\left(\frac{P}{T}\right)^2 + \cdots + \frac{n!}{(n-j)!}\left(\frac{P}{T}\right)^j + \cdots + n!\left(\frac{P}{T}\right)^n\right] \tag{3.10}$$

Letting $P/T = r$ and solving for π_0 yields

$$\pi_0 = \frac{1}{\sum_{j=0}^{n}\frac{n!}{(n-j)!}r^j} \tag{3.11}$$

Thus

$$\pi_i = \frac{\frac{n!}{(n-i)!}r^i}{\sum_{j=0}^{n}\frac{n!}{(n-j)!}r^j} \tag{3.12}$$

π_0 is the steady-state probability that the processor is idle.

Now, let W be equal to the mean length of time a user spends in the working part of the interaction (i.e., the mean response time). Let $\overline{Q}$ be the average number of users waiting for service. Then

$$\overline{Q} = n \frac{W}{W + T} = \sum_{i=0}^{n} i\pi_i \tag{3.13}$$

and

$$\overline{Q} = \frac{\sum_{i=0}^{n} \frac{in!}{(n - i)!} r^i}{\sum_{i=0}^{n} \frac{n!}{(n - i)!} r^i} = n \frac{W}{W + T} \tag{3.14}$$

Solving for W

$$W = \frac{\sum_{i=0}^{n} \frac{ir^i}{(n - i)!}}{\sum_{i=0}^{n} \frac{r^i}{(n - i - 1)!}} T \tag{3.15}$$

Dividing both sides by P and using the definition of r:

$$\frac{W}{P} = \frac{\sum_{i=0}^{n} \frac{ir^i}{(n - i)!}}{r \sum_{i=0}^{n} \frac{r^i}{(n - i - 1)!}} \tag{3.16}$$

Expressed in terms of π_0 ,

$$\frac{W}{P} = \frac{n}{(1 - \pi_0)} - \frac{1}{r} \tag{3.17}$$

$$W = \frac{nP}{(1 - \pi_0)} - T \tag{3.18}$$

It is interesting to note that the rate matrix and the resulting calculations would be the same if it were assumed that each program were run a finite quantum of time or if all programs were run to completion, I.e., batch processed! This is due to the fact that there is no swapping loss and that the time distributions are exponential. Different distribution functions would not, in general, yield the same results for any quantum size.

3.5 Markov Model for Multiple-Processor, Time-Shared Systems

Using the same definitions and assumptions as for the single-processor system, a model for the operation of an m processor, n user time-shared system can be derived. From a state where j users are waiting for service, the exit rate to the state where there are (j+1) users waiting for service is (n-j)/T, the same as for the single processor model. If j is less than m, then each user's program is assigned its own processor and the rate of exit to the state where (j-1) users are waiting for service is (j/P). If j is greater than or equal to m, the j users share the m processors just as in the case of a single processor, and the rate of exit to the state where (j-1) users are waiting is (m/P). The rate matrix for this process is not shown since it is similar to the one for the single processor case. Solving the equation

$$\Pi A = 0 \tag{3.19}$$

the steady-state probabilities are obtained:

$$\pi_0 = \frac{1}{\sum_{i=0}^{m-1} \frac{n!}{i!(n-i)!} r^i + \sum_{i=m}^{n} \frac{n!}{m!(n-i)!} \frac{r^i}{m^{i-m}}} \tag{3.20}$$

for i less than or equal to m,

$$\pi_i = \frac{\frac{n!}{i!(n-i)!} r^i}{\sum_{j=0}^{m-1} \frac{n!}{j!(n-j)!} r^j + \sum_{j=m}^{n} \frac{n!}{m!(n-j)!} \frac{r^j}{m^{j-m}}} \tag{3.21}$$

and for i greater than or equal to m,

$$\pi_i = \frac{\frac{n!}{m!(n-i)!}\frac{r^i}{m^{i-m}}}{\sum_{j=0}^{m-1}\frac{n!}{j!(n-j)!}r^j + \sum_{j=m}^{n}\frac{n!}{m!(n-j)!}\frac{r^j}{m^{j-m}}} \tag{3.22}$$

Finding

$$\overline{Q} = \sum_{i=0}^{n} i\pi_i \tag{3.23}$$

then

$$\overline{Q} = \frac{\sum_{i=0}^{m-1}\frac{ir^i}{i!(n-i)!} + \sum_{i=m}^{n}\frac{ir^i}{m!(n-i)!m^{i-m}}}{\sum_{i=0}^{m-1}\frac{r^i}{i!(n-i)!} + \sum_{i=m}^{n}\frac{r^i}{m!(n-i)!m^{i-m}}} \tag{3.24}$$

The average number of processors is similar to Q and is equal to

$$\sum_{i=0}^{m}(m-i)\pi_i$$

The expression for the ratio of the average time in the working part of the interaction to the average processor time per interaction is:

$$\frac{W}{P} = \frac{1}{r}\frac{\sum_{i=0}^{m-1}\frac{ir^i}{i!(n-i)!} + \sum_{i=m}^{n}\frac{ir^i}{m!(n-i)!m^{i-m}}}{\sum_{i=0}^{m-1}\frac{r^i}{i!(n-i-1)!} + \sum_{i=m}^{n}\frac{r^i}{m!(n-i-1)!m^{i-m}}} \tag{3.25}$$

The results obtainable from the models presented in this chapter will be compared to the measurements made on CTSS in the next chapter. No closed form expression for W/P in terms of

π_0 was found. However, as n gets large, π_0 approaches zero, and

$$\frac{W}{P} = \frac{n}{m} - \frac{1}{r} \tag{3.26}$$

Chapter 4

ANALYSIS OF MODEL PREDICTIONS

This chapter compares the results obtained from the various simulation models, the Markov models, and the CTSS data. In addition, extensions into multiple-processor systems are analyzed. Each system studied is compared on the basis of several metrics. Measures of a system's response time to requests for service, hardware efficiency, scheduling procedure, and the effects of loading will be used to illustrate the differences between time-shared systems. Since the Markov model for the single-processor, time-shared system yields surprisingly accurate results, this model and its multiple-processor extension will be used in the discussion of both parallel- and series-connected multiple systems composed of either equal general-purpose processors or special-purpose processors. This chapter can be divided into two portions: a detailed study of CTSS and related systems, and a general discussion of multiple-processor, time-shared systems.

Four measures are used to compare the results from the CTSS-like systems. The first of these is the dependence of the mean time for the working portion of the interaction (i.e., response time) on the average number of users interacting with the system, all other parameters being equal. Since during the measurements of CTSS in operation these other parameters could not be controlled, the variation of processor time per interaction will be removed by including it in the measure of response time. For this reason, the ratio of response time to processor time per interaction will be substituted for response time. It turns out that this ratio is a more stable measure of CTSS performance. The variation of this ratio as a function of the average number of interacting users is a measure of how well the system responds to a change in its load.

Secondly, the relationship between processor time per interaction and response time will be investigated. This will be done with no variation in the number of interacting users, in the distribution of processor time per interaction, program size, and

so on. This relationship is a measure of the scheduling policy of a system and clearly shows how a system achieves a low mean response time by scheduling short running jobs more favorably. Thirdly, the probability densities of the response time for the various systems will be discussed in conjunction with the scheduling policies.

Finally, the percentage usage of the processor, disk storage, and drum storage will be plotted as a function of the number of users interacting with the system. This function is used in a discussion of the relationships between hardware usage and scheduling policy, response time, etc.

Throughout the use of the above measures, the results from the simulation of CTSS will be compared with the actual measured data. Since the primary objective in the CTSS measurements was to characterize the user, and since time and space for the measuring programs was limited, some CTSS parameters were not measured.

The results from approximately twenty simulations are used here. Each simulation represents the operation of a time-shared system under a load of between twenty and forty-five users for a period of eight hours. Each such simulation cost between ten and twenty minutes of IBM 7094 computer time. While such a cost is not exorbitant, it put a limit on the total number of simulations that could be run.

Multiple systems are divided into two types: series and parallel. The parallel systems are those in which interactions may be processed by any one of the several processors in the system. The selection of a processor to service a particular user's interaction depends on the decision process employed. Serial systems are those in which interactions must be processed by more than one processor in some sequence. Both types of systems will be considered for the two-processor case. The results can be easily generalized to cases with more than two processors and having any arrangement of interconnections.

Two types of parallel two-processor systems are discussed. First, a system is analyzed in which either processor may serve any portion of any user's interaction requirement. This system corresponds exactly to the case for m equal to two of the multiple-processor Markov model derived in Chapter 3. This system can be thought of as a single queue feeding two processors. The second type of system analyzed is one in which

the two processors are able to service mutually exclusive types of interactions, i.e., each processor has its own queue. In both cases the two processors have the same capacity and service the same number of users, on the average. This is an optimal arrangement; the effect of the nonoptimal situation is discussed in another example. A double-speed, single-processor system will be used as a basis of comparison with these parallel two-processor systems. A two-processor serial system is then discussed and analyzed.

4.1 Response Time versus the Number of Users

Figure 4.1 shows the ratio of mean response time (time for the working portion of an interaction) to mean processor time per interaction for various systems. The normalization of response time with respect to average processor time is unnecessary for the simulated systems because the mean processor time per interaction was identical for all. However, the CTSS data were taken under normal operating conditions, and thus the data points represent many different means for processor time per interaction. Even though these means were all close to the overall mean of .88 seconds, use of the ratio of response time to processor time gives less variance to the results than if the unnormalized response time were used. Moreover, this ratio is an important performance parameter by itself being the reciprocal of the proportion of the system's computing power seen by an interacting user. The CTSS data points shown in Figure 4.1 were measured during all periods of operation: prime time, weekends, and evenings. Through these points is passed a cubic, least-squared-error fit. The RMS deviation from this fit was a surprisingly low 1.2.

The data points resulting from the simulations of CTSS are also shown in Figure 4.1. All of the simulation data points are well within the envelope of the CTSS points and are close to the cubic fit.

In addition, Figure 4.1 shows the prediction of the single-processor Markov model. In order to use this model, the mean processor time per interaction must be increased to take into account the mean swapping time per interaction, because the processor is idle during swapping in CTSS. The mean swapping time per interaction was measured and found to be .56 second. Thus, the mean net processor time per interaction is 1.44

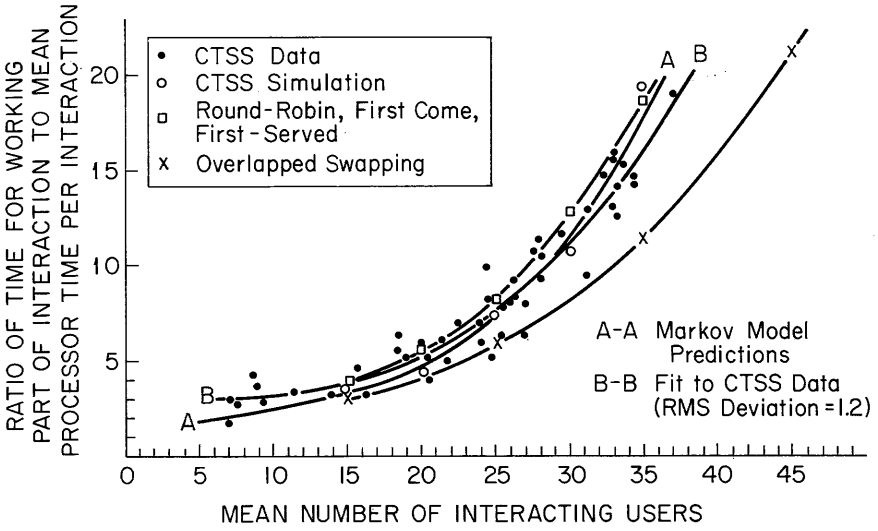

Figure 4.1. Ratio of mean time for working portion of interaction (response time) to mean processor time per interaction vs. mean number of interacting users.

seconds. A mean time for the console portion of an interaction of 35.2 seconds was used. These values were substituted into the expression for (W/P) for the single-processor Markov model in the preceding chapter. The results obtained from this expression must be increased by a factor equal to the mean processor time per interaction, including swapping, divided by the mean processor time per interaction without including swapping time. This is done to keep the ratio in terms of the .88 second processor time per interaction. The implications of the good fit of the results obtained from the Markov model to the CTSS data will be discussed in Chapter 5. No significance should be attached to the way the Markov curve crosses the fit to the CTSS data.

Figure 4.1 also shows the data obtained from the simulation of CTSS with its Scheduling Algorithm replaced by a first-come, first-served, round-robin scheduler. Each user's program was allowed to run for a time slice of two seconds. If it had not finished by that time, it was pre-empted and placed at the end of

the queue, and the next user swapped in and run. The slightly worse performance of this system compared with CTSS can be attributed entirely to the difference in scheduling. The difference between this system and that of the Markov model can be attributed to the fact that they have slightly different swapping overheads.

The results of the simulation of the system with overlapped swapping and processing are also shown in Figure 4.1. The reason for the improvement in performance is that the time during which the processor is idle because of swapping is down to .33 seconds per interaction, a decrease of approximately 41 per cent from the CTSS value.

The ratio of response time to processor time can be thought of as indicating what part of the system's capacity a user is receiving. For example, a value of ten for this ratio indicates that, on the average, a user receives one tenth the capacity of the system. As expected, this ratio is just over one for only a few interacting users and nearly equal to n, the number of users, as n gets large. The increase over a unity value for n equal to one or two is because of swapping time. As n increases, this ratio increases slowly due to the fact that not all interacting users require service at once. For large n, nearly all of the users are in the working portion of the interaction and the ratio approaches n.

4.2 Scheduling Policy and Response-Time Distributions

The average response time to a particular processor time requirement can be a useful parameter. It is used here to show how the CTSS Scheduling Policy differs from the round-robin, first-come, first-served system. These measurements, made from simulations and shown for a load of twenty-five interacting users in Figure 4.2, is highly dependent on the distribution of processor time per interaction. The one used to obtain these measurements was shown in Figure 2.5. It is clear that CTSS obtains its slightly better mean response time by favoring the short-running interactions at the expense of the longer ones. The primary effect is that the swapping overhead is reduced. The reason for this effect is that the longer-running interactions, with low priority, are run for a longer time when they are finally run. This is one of the advantages of a dynamically variable quantum time. The "shortest-operation-first" aspect of

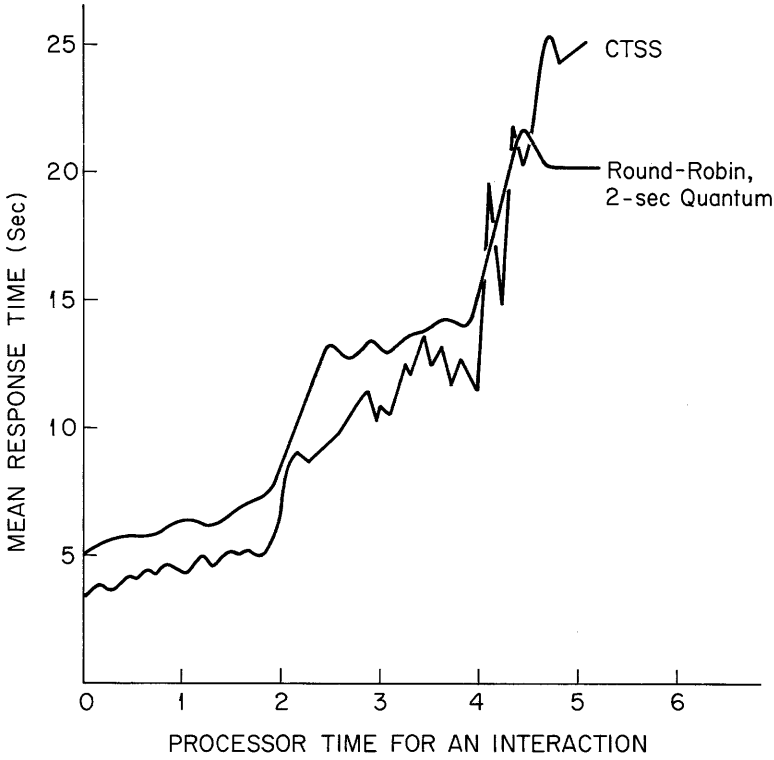

Figure 4.2. Mean response time vs. processor time per interaction for CTSS and round-robin (quantum = 2 sec.)

CTSS scheduling seems to be slight, and is discussed in the Conclusions (Chapter 5).

Figure 4.3 shows the response time versus processor time per interaction for CTSS under varying loads. The shift from an effective quantum time of two seconds to one second can be clearly seen at loads of approximately 28 users. Figure 4.4 shows the same curves for the round-robin, first-come, first-served scheduler for 25 users and values of quantum times from .5 to ten seconds. Using a quantum time of ten seconds is almost equivalent to a policy of running all programs to completion. The variability of the response times about the means plotted can be estimated by the jaggedness of the plots. From this it could be deduced that the CTSS Scheduling Algorithm yields somewhat less predictable response times than does a round-robin, first-come, first-served scheduler. This is in agreement with the result of job-shop scheduling showing that first-come, first-served scheduling yields a higher mean

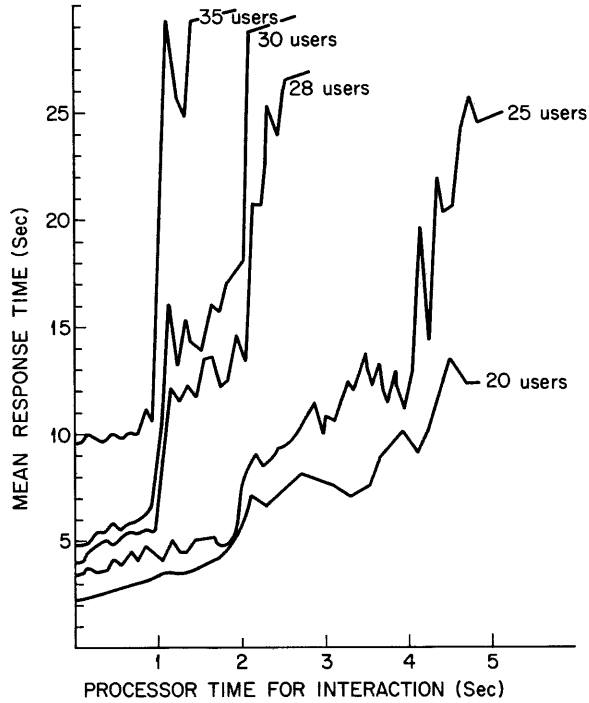

Figure 4.3. Mean response time vs. processor time per interaction for CTSS.

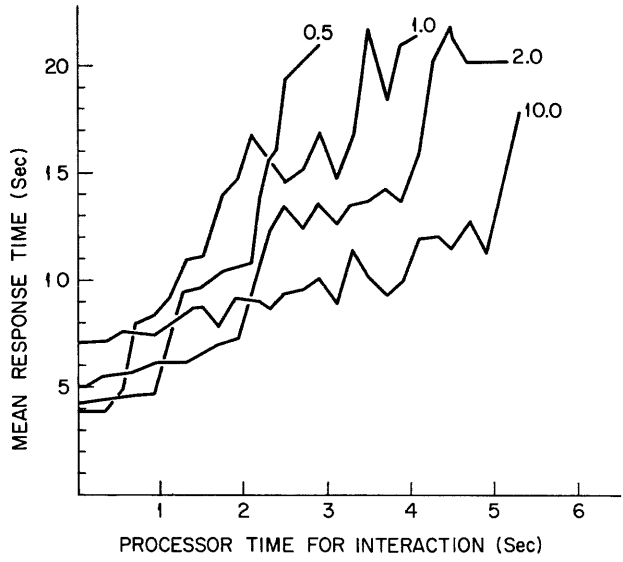

Figure 4.4. Mean response time vs. processor time per interaction for various quanta in round-robin, first-come, first-served scheduling (25 users).

completion time with a lower variance than shortest-operation-first scheduling (see CONWAY, 1964, p. 1020). This "trade-off" will be discussed in Chapter 5.

There are several restrictions on the response time as a function of processor time. If the scheduling procedure does not schedule on the basis of job types or attempt to predict the processor time for an interaction, the slope of the response time versus processor time curve must be greater than or equal to unity at all points. Furthermore, the response time as a function of the processor time, w(p), must satisfy the following relation:

$$n \int_0^{\infty} \frac{p\, q(p)}{t(p) + w(p)}\, dp \leq 1 \qquad (4.1)$$

where n is the number of users interacting with the system, p is the processor time per interaction, q(p) is the probability density of p, t(p) is the think time as a function of p, and w(p) is the scheduling policy function -- the response time as a function of p. All that this relationship expresses is that the scheduling policy cannot attempt to provide to the users more than one second of processor time per second.

Figure 4.5 shows the response-time distributions for CTSS (simulation) and round-robin scheduling with quanta of .5 and 2 seconds.

The differences between CTSS and the round-robin schedulers for large response times is not shown. The CTSS response time distribution has by far the longest tail. The effects of CTSS favoring the shorter interactions is again evident in the higher probability for shorter response times. The round-robin with a quantum time of 2 seconds yields almost exactly the same distribution as for longer quanta, except for differences in means. An interesting effect can be seen when comparing the distributions for the two different round-robin schedulers. With a quantum time of .5 second, it would be expected that the probability of extremely short response times should increase over the case with a quantum of two seconds. However, the effect is that a .5-second quantum increases swapping overhead sufficiently to overcome most of the advantage of a small quantum time to the short-running users. The data points from the CTSS measurements are not shown in Figure 4.5 because they coincide

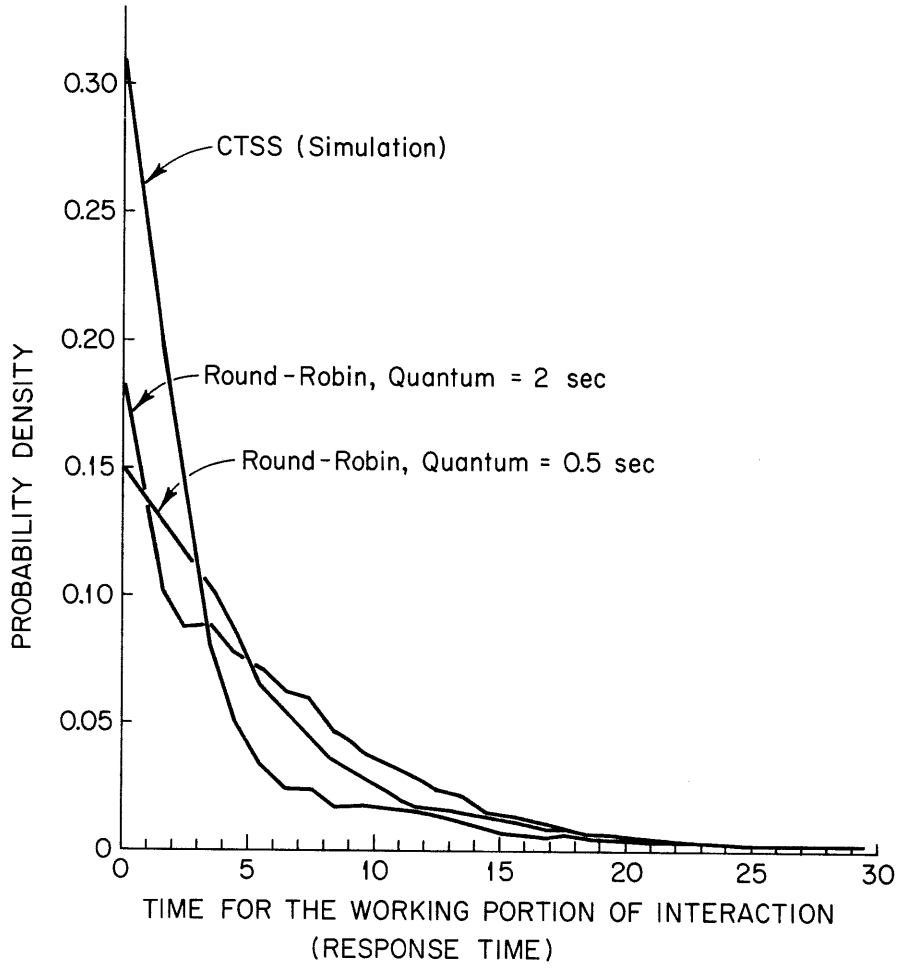

Figure 4.5. Probability density of response time per interaction.

almost exactly with the results of the CTSS simulation.

As a further means of comparison, the mean response times of the various systems simulated are listed below (all figures are for 25 users):

System	Mean Response Time (seconds)
CTSS	7.0
Round-Robin	
Quantum= .5	10.7
Quantum= 1.0	7.7
Quantum= 2.0	7.3
Quantum= 10.0	8.1
Overlapped Swapping version of CTSS	5.1

The differences in the mean response times for the round-robin schedulers are due to the differences in swapping overhead. Other effects of changing the quantum size will be discussed in Chapter 5.

4.3 Hardware Usage

Figure 4.6 shows the percentage usage of the processor for

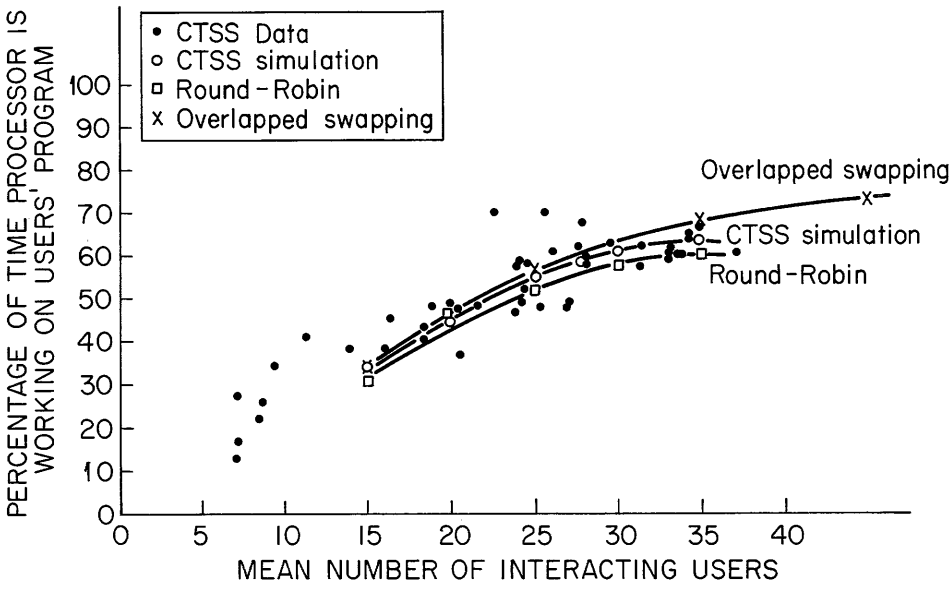

Figure 4.6. Percentage usage of processor for users' programs vs. mean number of interacting users.

users' program execution as a function of the number of interacting users. Shown are the CTSS data points as well as the points from the simulations of CTSS, the round-robin system with a quantum time of two seconds, and the system with overlapped swapping. The phenomenon of saturation can be clearly seen. As the number of users increases, the usage of the processor increases until some limit is reached. With completely overlapped swapping, this limit would be 100 per cent. But since there is some swapping overhead, the limit is approximately 60 per cent for CTSS and the round-robin system and 75 per cent for

the (imperfectly) overlapped system. It is interesting to note that these curves are equal to the quantity $(1-\pi_0)$ of the Markov models within a multiplicative constant, where π_0 is the steady-state probability of zero users waiting for service. This constant is equal to the asymptote of the usage curves. (The Markov model predictions are not plotted because they are so close to the measured results.)

The efficiency of the processor usage in CTSS is not particularly bad. For a scientific installation (IBM 7090 or similar) using batch processing techniques, a 60% processor usage is about average, and 75% usage is excellent.

Since saturation can be identified with the probability of zero users waiting for service, it might be defined as follows: Saturation occurs when the probability of zero users waiting for service is lower than some small number, ε. The specification of ε is arbitrary, but typically might be .01. Another definition of saturation will be found in Chapter 5.

Figure 4.7 shows the usage of the drum and disk for swapping as percentage of total time. As is expected, the usage of the disk and drum increases with the number of interacting users. The usage of the disk for CTSS is slightly larger than that of the round robin. This effect is a result of the preference shown by the CTSS Scheduling Algorithm for short-running interaction. This preference extends to interactions that have received no service; since these interactions very often require loading from the disk, the amount of disk usage is up slightly. The drop in disk usage for CTSS at approximately 30 users is due to the fact that the CTSS scheduler switches to an effective quantum time of one second at that point (see Figure 4.3). With a smaller quantum time, there is more swapping and thus an increase in the use of the drum. Comparing the CTSS disk and drum usage with that of the round-robin scheduling (quantum time of 2 seconds), it is apparent that CTSS has a slightly lower swapping overhead. This is because the CTSS Scheduling Algorithm uses different quanta, depending on the number of users waiting for service at a particular time -- the fewer the users waiting, the longer the effective quantum.

The approximately 10 per cent usage of the drum and 30 per cent usage of the disk imply that if interprocessor interference could be eliminated a single disk or drum could service more than one processor. The only condition on this statement is that

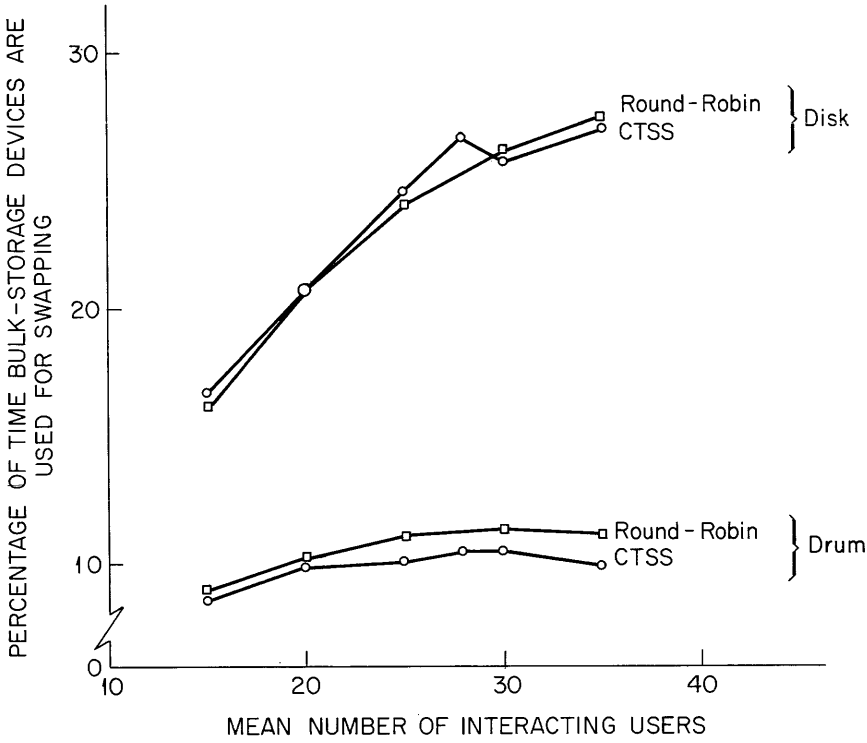

Figure 4.7. Usage of bulk storage devices for swapping vs. mean number of interacting users.

the storage capacity of the disk or drum would have to be increased to take care of the additional users' disk files and/or core images.

In the system with overlapped swapping and processor use, the total use of the bulk storage devices increases with the usage of the processor, as is the case with the nonoverlapped systems. At a load of 45 interacting users, the usage of the drum was 12.7 per cent, and that of the disk 33.8 per cent. Nonoverlapped usage of the drum amounted to 8.5 per cent, and of the disk 9.1 per cent. Unfortunately, runs were not made with more users, and saturation operation with the overlapped system was not achieved. However, an extrapolation of the measurements taken indicates a saturation level of usage for the processor at approximately 75 per cent.

4.4 Multiple Systems

Parallel multiple-processor systems can be divided into two basic classes, depending on the ability of a single processor in the system to serve requests. In a multiple system with general-purpose processors, any processor can service any request. In effect, requests to be serviced (i.e., programs to be executed) are drawn from a single queue. The alternative multiple-system arrangement uses special-purpose processors to the extent that a single processor can serve only certain types of requests. In this case a processor draws requests to be serviced from the queue containing the proper type of request. In practice, a multiple system may contain both types of operation: a group of processors fed from a single queue, and many queues differentiated by the type of request being serviced by the attached processor group. A third kind of multiple system is the serial type, in which jobs must pass through two or more time-shared processors in sequence. An example of such a system would be one with input and output processors which pre- and post-process all jobs for the main processor.

In general, any multiple system can be analyzed by considering each group of processors individually, as will be discussed. Two parallel multiple systems, each with two processors, will be analyzed. Their performance will be compared with a single-processor system of the same capacity in instruction execution rate. Next, the effects of augmenting a CTSS-like system with a processor capable of simple character manipulation and channel operations to take care of some of the job types normally executed at the central processor will be studied. The augmented system will be compared to CTSS on the basis of performance and additional cost. Finally, serial multiple systems will be analyzed.

4.4.1. Equal Capacity, Parallel Multiple Systems. The first multiple system to be considered is one with two general-purpose processors. Assume that each processor has the capacity of the single processor of the CTSS system (IBM 7094-I) and that the same swapping overhead is present. Thus, the average net processor time per interaction is 1.44 seconds, .88 second of useful processor time and .56 second of processor idle time due to swapping per interaction. With an average "think" time of 35.2 seconds, the parameter r is .041. The results from

the Markov model of a two-processor, time-shared system are plotted in Figure 4.8. The average response time (time in the working portion of an interaction) is shown as a function of the

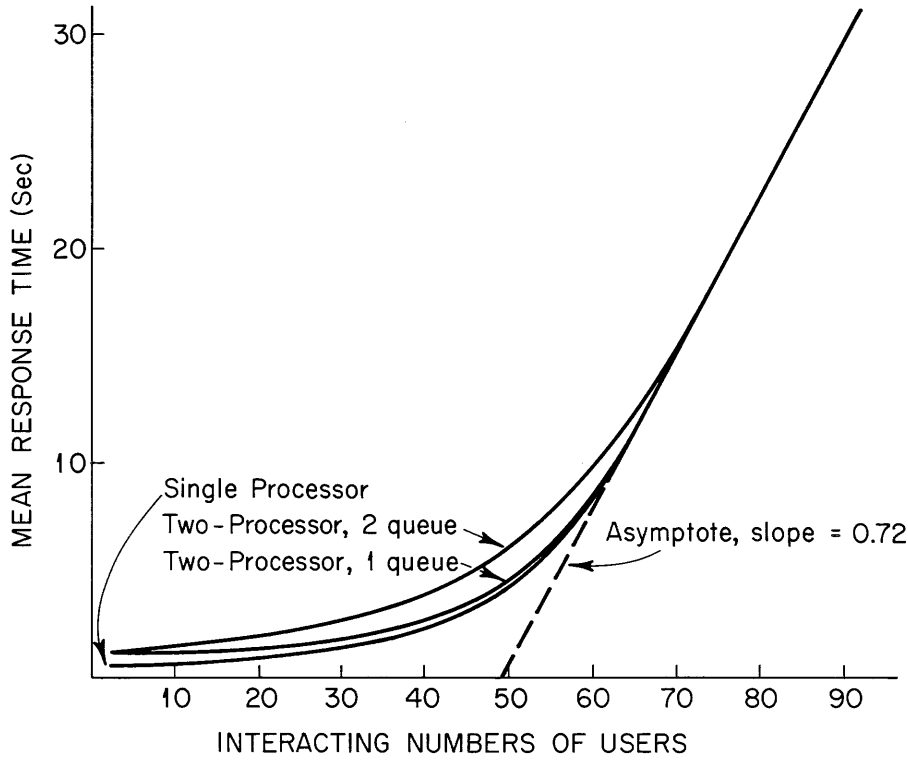

Figure 4.8. Mean response time vs. number of users for three multiple systems with equal capacity.

number of users interacting with the system.

A system with two special-purpose processors can be equivalent to two separate time-shared systems if each subsystem is used, on the average, by half of the user population and each is equally loaded (i.e., equal P's and T's). As will be seen, this is an optimal situation. Any deviation from the ideal results in an increased over-all mean response time. For such a system, the mean response time can be calculated as a function of n, the number of interacting users, by using n/2 in the expression for the response time for the single-processor Markov model. The results of this calculation are also shown in Figure 4.8.

As a basis for comparison, a single-processor system of exactly the same capacity as the two-processor systems will be used. If the single-processor system is exactly twice as fast for every operation, the r for this system is .02, exactly half that of the duplex systems. The mean response time for this system is plotted in Figure 4.8 as a function of the number of interacting users.

Looking at the three systems from the standpoint of over-all mean response time, the double-speed, single-processor system is superior. The two-processor, single-queue system is better than the two-processor, two-queue system. The reasons for the differences in the systems can be explained in terms of degrees of freedom. The single-processor system can turn the entire capacity of its processor to the execution of a single job. Thus, the single-processor system is superior by almost a factor of two to the two-processor systems when the number of users is small. The only difference between the two-processor, single-queue system and the single-processor system is their performance when only one user is waiting for service. This difference is due to the fact that there are times when the queue of one processor of the double-queue system is idle and its capacity cannot be used to relieve the other processor. The fact that the plots all join into the same straight line is indicative of the fact that all three systems have identical capacity. The full capacity of all three systems is used because the number of interacting users is sufficient to keep all of the queues non-empty nearly all of the time. Thus, running in complete saturation, the performance of each of these systems is identical.

4.4.2. Systems with Special Purpose Processors. The two-processor, two-queue system is of special interest because it can be used to represent a system with two special-purpose processors. In such a system the users interact with one processor at a time, and the processor that is used is determined by the type of task the interaction performs. The assumption will be made that the mix of interaction types remains constant regardless of the service a user receives. Thus, the relative probabilities of different types of interactions remains fixed. Reference should be made to the end of Chapter 2 for the measurements made of the probabilities of different types of interactions. In order to analyze the two-queue, two-processor

situation quantitatively, the following variables are defined:

α_i = the probability that a user's next interaction will require the use of processor i. The sum of the α's is unity.

P_i = the mean processor time per interaction for processor i. This time includes any non-overlapped swapping time.

T_i = the mean time for the console portion of interactions using processor i (i.e., "think" time).

n = the total number of users interacting with all processors.

All of these parameters are determined by user behavior and the speeds of the various processors. The following variables are functions of the above:

n_i = the mean number of users interacting with processor i.

W_i = the mean time for the working portion of interactions using processor i (i.e., response time).

W = the overall mean response time.

The analysis of multiple, special-purpose processor systems will be carried out for only two processors, but the extension to any number is straightforward. In order to determine the W_i and W, the n_i must be found. For a two-processor system, $n_2 = n - n_1$ and thus only n_1 need be found.

$$n_1 = \frac{\alpha_1(W_1 + T_1)}{\alpha_1(W_1 + T_1) + \alpha_2(W_2 + T_2)}\, n \qquad (4.2)$$

The expression for n_1 was found by equalizing the rate of users starting interactions of type one and the rate of users finishing interactions of type 1. This equation can also be interpreted as the total number of users multiplied by the mean proportion of the time that a single user is interacting with processor number 1. Since the W_i are non-linear functions of the n_i, and since both are unknown, the n_i are perhaps best found by a relaxation technique (i.e., a value is assumed for n_1 and n_2,

then W_1 and W_2 are calculated; a new value for n_1 is calculated, etc., etc.). This being done, the overall mean response time per interaction can be found:

$$W = \alpha_1 W_1 + \alpha_2 W_2 \tag{4.3}$$

In the two multiple systems previously discussed, the two-processor, two-queue system was "balanced." That is, each processor served the same average number of users, each had the same associated mean think and processor time per interaction. In general, values for the parameters of a system with special-purpose processors will not be so favorable. The parameters under the control of the systems designer are the P_i and the a_i. Selectively scheduling on the basis of command type by giving certain commands a higher scheduling priority, the probabilities for the occurrence of interactions associated with "out-of-favor" commands will decrease as users abandon their use as being too time consuming. The P_i can be changed by shifting "capacity" around the system; that is, by manipulating processor speeds. Comparisons between two systems without involving economic issues makes sense only if both have the same over-all capacity to execute instructions or to process interactions. If capacity is defined in terms of instructions per second or interactions per second, then manipulation of the P_i that leaves the total capacity unchanged is subject to the following constraint:

$$\sum_{i=1}^{m} \frac{1}{P_i} = C \tag{4.4}$$

Here C is the capacity of the entire system in interactions per second, and m is the total number of processors. A solution to the problem of minimizing W as a function of the a_i and P_i is possible, but is best taken up after a consideration of the two-processor, two-queue system in an unbalanced mode of operation. If the processors are unbalanced, the operation of the total system exhibits an interesting phenomenon as one of the processors saturates before the other.

The following describes a specific example of a two-processor, two-queue system drawn from a real world situation. Nature being what it is, this system is unbalanced. The effects of imbalance, possibly remedies for it, and

differences in performance will be discussed. The example described has another merit in that an interesting actual situation is analyzed.

Using the data concerning the mix of task types for the CTSS user (see Chapter 2), some predictions will be made on the cost and performance of a CTSS system augmented by an auxiliary special-purpose processor. Of the five types of tasks that CTSS performs, two can clearly be accomplished by a processor that is simpler than that of the IBM 7094 with little increase in the mean processor times per interaction for these tasks. Specifically, the Program Input and Editing commands involve very little data processing. A simple, character-oriented processor could probably do the same manipulations that are required of the 7094 processor in much the same time. Fewer instructions would have to be executed since no unpacking and repacking of characters is necessary in the character-oriented processor. Furthermore, such a processor would be considerably cheaper than an equivalent speed (for Input and Editing), general-purpose, parallel arithmetic, floating-point processor. The other type of task which a very simple processor could perform is that of File Manipulation. The operations involved are merely that of controlling a disk channel so as to copy, concatenate, split, merge, and so on, disk files. If such a processor were added to the present configuration, it would relieve the load on the central processor of the 7094 to the extent of handling nearly half of the interacting users, nearly half of the interactions, and approximately 35 per cent of all users' processing requirements. First, the performance improvement brought about by the addition of this auxiliary, special-purpose processor will be analyzed; then a few general statements about the additional cost of the system will be made. Essentially, the auxiliary processor will transform CTSS into a two-processor, two-queue system. For the sake of simplicity, the assumption will be made that the auxiliary machine is able to process interactions of the File Manipulation and Program Input and Editing types at the same speed as the IBM 7094 processor. Furthermore, assume that the swapping overhead is the same as in the normal version of CTSS (.39 of the total processor time per interaction). These assumptions are reasonable since the auxiliary processor will probably be somewhat slower than the IBM 7094, but its swapping overhead will be lower because of its smaller programs and the

possibility of "read-only" routines for common functions that are not swapped. Using the data of Chapter 2 and the definitions of the preceding pages (processor number 1 will be the auxiliary):

$a_1 = .48$
$a_2 = .52$
$P_1 = .63/.61 = 1.03$ second
$P_2 = 1.11/.61 = 1.82$ second
$T_1 = 39.3$ second
$T_2 = 31.3$ second

W is found as described before and is plotted in Figure 4.9 along with the values for normal CTSS as a function of the total number of interacting users. In both cases the Markov model was

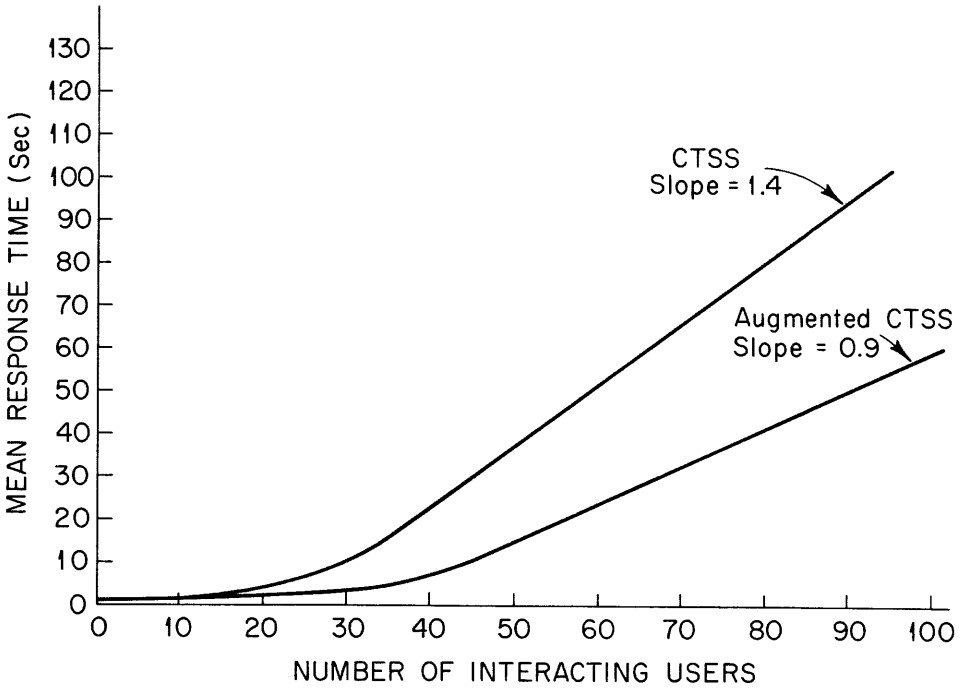

Figure 4.9. Mean response time vs. number of interacting users for CTSS and "augmented" CTSS.

used to obtain the W_i. It is interesting to note, for example, that the "augmented" version of CTSS yields the same mean over-all response time with a load of 45 users as the normal CTSS

does with 30. The distribution of over-all response time for each system will, in general, have different shapes. The normal CTSS over-all mean response time should have a variance smaller than that for the augmented system. Figure 4.10 shows a plot of

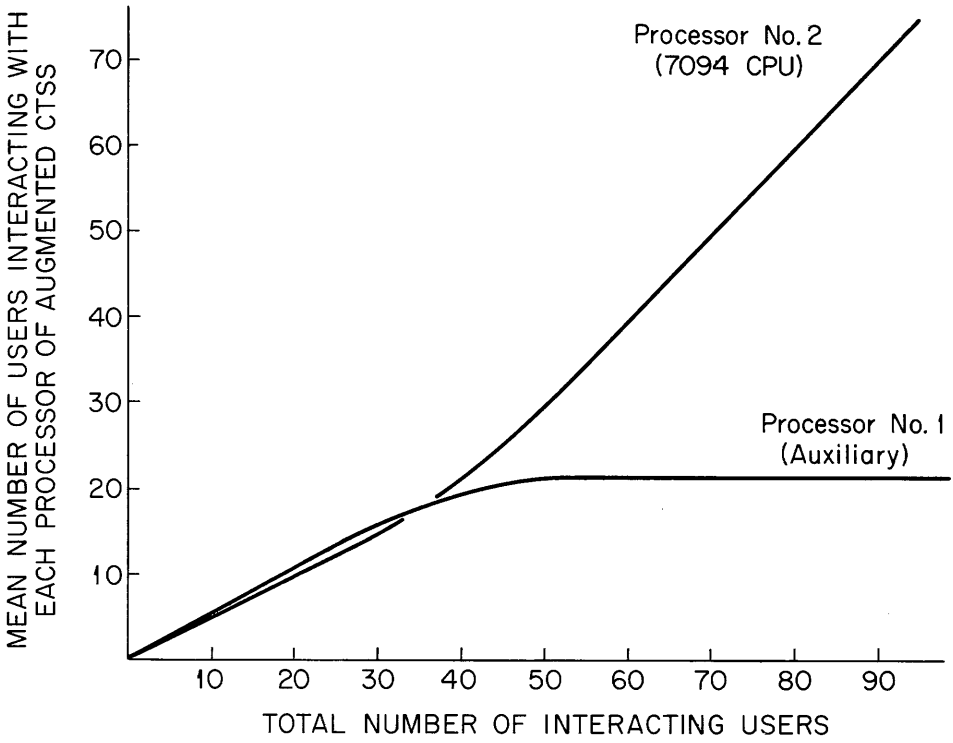

Figure 4.10. Mean number of users interacting with each processor of augmented CTSS vs. total users.

the mean number of users interacting with each portion of the system as a function of the total number of users. Since the capacities of the two systems and the user preferences between them (the a_i) are nearly equal, the n_i are nearly equal until the total number of users in the sytem reaches approximately 40. At this point the values of the n_i diverge. The mean number of users interacting with the less heavily loaded auxiliary processor remains constant at 21, and any additional users added to the entire system show up in the mean number of users interacting with the 7094 central processor. The reasoning explaining this phenomenon is that because the 7094 processor is more heavily loaded than the auxiliary one, the mean response

time for the 7094 processor increases more for each user added to the system than that of the auxiliary. As a response time increases, more users are held waiting for the corresponding processor on the average, and the response time increases still more. This effect is analogous to a positive feedback situation in an amplifier: saturation occurs. As the total number of users increases, the 7094 processor is forced into saturation and holds more and more of the total number of interacting users, leaving the auxiliary processor with a steadily decreasing proportion of the user population.

By manipulating either the P_i or the a_i, or both, both processors can be kept out of saturation or at least equally saturated. Taking the expression for n_1 presented in the previous discussion and substituting for $W_i + T_i$ the value yields an equation in terms of the P_i, π_i, and a_i which must be satisfied in the steady state:

$$\frac{\alpha_2 P_2}{(1 - \pi_{02})} = \frac{\alpha_1 P_1}{(1 - \pi_{01})} \tag{4.5}$$

Both processors can be kept equally saturated if the π_i's are set equal. This will insure that neither processor saturates leaving the other unsaturated (i.e., $\pi_1 = 0$ and $\pi_2 \neq 0$). Doing this yields the balance condition:

$$\alpha_1 P_1 = \alpha_2 P_2 \tag{4.6}$$

Looking at the example under consideration and keeping the constant capacity restriction in mind, the only way to achieve balance would be either to encourage users to shift their interaction mixes toward the File Manipulation and Program Input and Editing, or to speed up the 7094 processor (perhaps by installing a Model II 7094) and/or allow a slight decrease in the performance of the auxiliary processor.

Returning to the example, the effects of adding the auxiliary processor is to increase the number of users that can be effectively served by 50 per cent. The additional cost of the system is just that of the auxiliary processor and the hardware necessary to access a disk file from two different processors simultaneously. Certainly this additional cost will be well under 50 per cent of the cost of the IBM 7094 CTSS configuration. Any further discussion of costs at this point would involve going beyond the purposes of this example.

4.4.3. Serial Multiple Systems. A serial multiple system is one in which users' interactions are processed by two or more processors in sequence. That is, after a user completes the "think" time, his job requires P_1 seconds of processor time on the first time-shared processor, P_2 seconds on the second time-shared processor, etc. For two processors, the think time T and the response time of the first processor W_1 appear to the second processor as the actual think time. Therefore, using the results from the single-processor Markov model:

$$W_2 = \frac{nP_2}{1 - \pi_{02}} - W_1 - T \tag{4.7}$$

Similarly, for the first processor:

$$W_1 = \frac{nP_1}{1 - \pi_{01}} - W_2 - T \tag{4.8}$$

The solution to these two equations is obtained by simply adding them and solving for the net response time, $W_1 + W_2$:

$$W = W_1 + W_2 = \frac{n}{2}\left[\frac{P_1}{1 - \pi_{01}} + \frac{P_2}{1 - \pi_{02}}\right] - T \tag{4.9}$$

A serial system such as this one would be a good model for a real system if all of the jobs processed by the main computer (processor 1) are then sent to an output computer for post-processing. For nonsaturation operation, a relaxation technique such as described previously must be used to find the steady-state values for W_1 and W_2 separately.

Chapter 5

CONCLUSIONS

The purpose of this chapter is to analyze the generality of the techniques and models presented as well as to summarize and discuss the specific results obtained. The use of mean response time as a performance metric is discussed and the generality of the techniques and models for predicting the performance of interactive time-shared systems is evaluated. Furthermore, results concerning the specific systems studied are outlined and analyzed.

The most important performance aspect of any man-machine system is the quantity of results obtained (in terms of "significant" problems solved) per unit time. At this time, such a quantity is unmeasurable. The productivity of a machine user might be represented in simpler, more tractable terms. By excluding an evaluation of the relative merits of the problems solved, productivity can be measured in terms of subsolutions (or "microsolutions") obtained per unit of time. In this way the use of the time rate of interaction can be defended as being a well-defined, measurable quantity which approximates (however roughly) the ultimate performance metric. Since the number of interactions per unit time can be derived in terms of the response time and the user think time distributions and because the use of the response time allows the machine performance to be separated from users' performance, the distribution of response times was chosen to be the primary system performance measure. Other parameters of interest, such as the mean number of users queued at a particular processor, can be derived from mean response time.

The problem can also be viewed from the hardware aspect. A system that performs a useful processing function at a certain performance level and cost (measured both in hardware and user time) is clearly "better" than a functionally equivalent one with the same performance level but a higher cost. In a sense, a measure of cost is the complexity and duty factor (percentage of use) of the hardware portion of the system. Moreover, there is

an infinite variety of possible information processing services that can be utilized by a user, and an infinite number of possible system organizations to provide him these services. The tasks a system performs, as well as the level of acceptable performance and hardware complexity, are obviously important issues. However, primary attention is paid to system performance as characterized by the distribution of response times which are, in turn, derivable from the characterizations of both the user and the interactive system.

There are many metrics that can be used to describe a distribution of response times. For some purposes the maximum might be equated with the "worst-case"; the minimum, with the best achievable service. The mode of the distribution indicates the most probable level of service. A simple metric, such as the mean (with or without a measure of expected deviation), characterizes most of these aspects, is readily computed, and is extremely useful in modeling. For this reason, it is used as the principle characterizing feature of interactive performance.

A fairly simple model for a generalized interactive system can be developed in terms of the mean response times. This model can be viewed as an arbitrary network of unilateral delays through which the processes necessitated by user interaction pass. The delays are of two types: those due to the user, and those due to the hardware system. User caused delays represent the time it takes for users to observe console output from previous interactions, to think over their next moves, and finally to produce console input which will request further services from the hardware system. All of these activities are lumped under the single term "thinking." The system-induced delays represent the response times of the various subsystems that comprise the hardware portion of the entire interactive system. Each node in the network may have any number of branches entering or leaving it provided there is at least one in each direction. Each branch leaving a node has a probability of being selected by a user in preference to the others. This probability is a function of the specific service or type of interaction selected by the user. The allocation of hardware resources to various jobs by the system will be taken into account in the processing delays. In general, the delays and probabilities may be any arbitrary function of the state of the entire system or its past history. The user-induced delays represent the think

times for various types of interactions. The processing delays take into account the capacity of the processors, the distribution of processor time, etc. for the particular type of interactions being delayed.

There may be portions of the system that are inaccesible to a number of its users. After assuming that certain numbers of users are interacting with each portion, the object is to determine the steady-state values of every delay in the network. In general, there may not be a steady-state solution if arbitrary delay functions are allowed. However, if all delays are positive and do not decrease with the addition of users to the corresponding branch, an analogy can be made to passive electrical circuits; and the existence of a steady-state solution can be shown to exist always.

A subclass of this generalized interactive system model has been analyzed. This subclass differs from the general model because of two simplifying constraints: First, the probabilities for choosing an exit branch from a node are assumed to be constants. These probabilities represent the users' choice of interaction types. Since the CTSS data showed that the mix of the five interaction types remained nearly constant from day-to-day over a period of more than two months (see Section 2.3), this simplication is justifiable. Second, there are restrictions on the functional dependence of the delays. In all of the systems discussed, the think times were constants. This was done on the basis of the CTSS data, as was discussed in Section 2.3. The only restriction on the delays is that they be some known function of the flow of users' interactions through the network.

The processing delays were determined with the use of the single and multiple-processor Markov models, but they may be determined by simulation of the individual subsystems themselves. Note that it should not be necessary to simulate the entire interactive system; simulation is necessary only for those portions of the system that are not mathematically analyzable. In some cases it may be necessary to do a few simulations of a subsystem in order to determine the net processor time per interaction. In other words, the processor time per interaction required by the user is known, but the effective expansion of this processor time due to overhead factors, such as nonoverlapped swapping, etc., may have to be determined by

simulation. The Markov models are useful if: (1) the subsystem is time-shared; and (2) the net processor time per interaction is a simple function of the processor time the user originally requires. This second condition can be amplified by the observation that the terms (1/p) and (j/p) in the original rate matrices for the Markov models (see Sections 3.4 and 3.5) can easily be made into more complicated functions and still be soluble for the steady-state occupancy probabilities. If the system is not time-shared (in the sense used in this monograph) or the accuracy of the Markov models is not sufficient, some other Markov process or method of analysis can be employed. The delay networks for a few specific examples of interactive systems are shown in Figure 5.1.

There may be the case that processor time per interaction is not the appropriate parameter, because the processor is not the rate-determining factor in a subsystem. In all of the examples studied, the processor speed determined the basic rate of operation. It may well be that there exist, or will some day exist, systems whose rate-determining element is, for example, the bulk-storage device. For such a case it might make more sense to use the term "bulk -storage use per interaction" instead of "processor time per interaction." Thus, in all of the discussions of processor time the possibility of substituting the time required for some other device should be kept in mind.

At this point, some observations will be made about the specific systems studied. First, some general remarks concerning the operation of CTSS-like systems will be presented and then some of the implications of the specific results obtained will be analyzed.

Perhaps the most important result of the research in this report is that time-shared systems and their users can be successfully modeled. The consistent, predictable behavior of users interacting with time-shared systems was not a foregone conclusion at the onset of the research; it had to be established. That a simple model is sufficient to represent a time-shared system user did not come as a complete surprise, but neither was it obvious from considering the situation a priori. The same statement can be made of the modeling of CTSS. The accuracy of the single-processor Markov model in predicting the CTSS mean response time function was somewhat more surprising. The implications of this accuracy will be discussed later in this

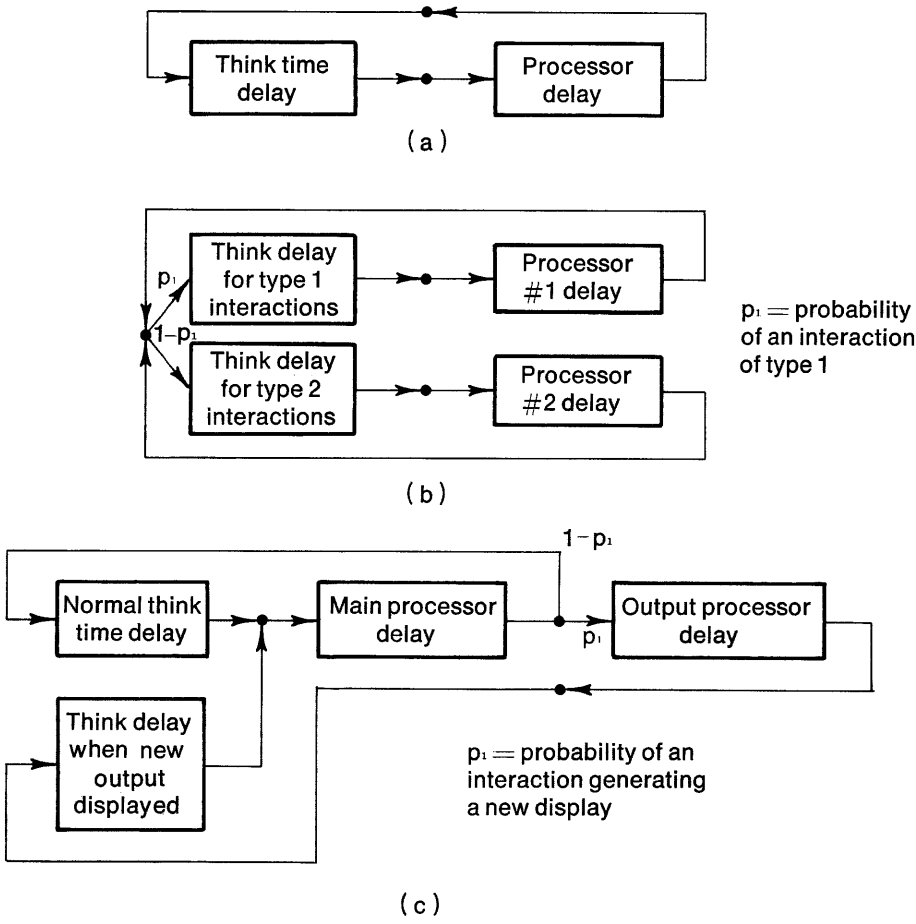

Figure 5.1. Delay networks for some interactive systems.

chapter.

As was seen during the discussion of mean response time as a function of the number of interacting users (Section 4.1), the operation of CTSS can be split into two segments, saturated and unsaturated. The exact point of crossover between these two regimes of operation is somewhat hazy, since saturation may only be defined with reference to a probability. That is, a time-shared system enters saturation when the probability of zero users waiting for service becomes less than some small number. Theoretically, the point where there is zero probability of an

empty queue does not exist except for an infinite load on the system. The most important factor determining when a system enters saturation is the mean net processor time per interaction. In nonsaturated operation the addition of another interacting user has little effect on the mean response time of the system, whereas for saturated operation an additional user makes a noticeable difference. There is a distinct difference in the way a CTSS-like system responds on the two sides of the saturation point. The difference in mean response time for loads of twenty and thirty users is much more noticeable (to the CTSS user) than that between loads of ten and twenty users.

Another definition of the saturation point can be based on the Markov model. Using the expression for the response time in a saturated m-processor, n-user system, $W = (nP/m)-T$. Extending this straight line to the axis (see Figure 4.8, for example) yields a crossing point of $n = m\ T/P$. This is the point at which the system first starts into saturation. For CTSS, this saturation point is equal to $(35.2/1.44) = 24.4$ users. This is a method of very roughly finding the capacity of a time-shared system measured in the number of interacting users it can handle. This is not to say the response times with $m(T/P)$ users will be satisfactory, however.

Another interesting phenomenon is the effect that changes in the performance of a time-shared system have on the behavior of the users. All of the data and results presented so far in this report were taken during a period when the operating characteristics of CTSS were stable. However, if a heavily used command is changed so that its average processor time is increased (thus increasing the mean response time), users are likely to decrease their usage of this command. If a new command is added which accomplishes the same function as an established command in a faster or more elegant manner, changes in the distributions of think time and processor time per interaction should be expected.

There are other factors that can affect user characteristics. If the scheduling procedure gives low priority to a user's program because of one of its characteristics (e.g., program size), users seem to try to eliminate these "objectionable" features from their programs and interaction usage (e.g., they attempt to write and use smaller programs). Thus it seems that scheduling, if properly executed, could be

used to "mold" the users to some extent by assigning priority on the basis of job type, program size, program running time, user think time, etc., etc. Then the user, in trying to "beat the system," will tend to conform to the image of what the writers of the scheduling program considered to be the ideal user. Fortunately (or unfortunately) users are generally not so flexible that they can all arrange their usage so as to be always in the highest priority group. If users were this flexible, all would have the same priority; and any scheduling procedure, no matter how complex at the onset of its use, would respond with simple round-robin behavior.

The fact that the single-processor Markov model produces accurate predictions of the mean response time as a function of the number of users interacting with the system has several interesting implications. The model (see Section 3.4) is a highly simplified version of the processes occurring in a highly complex hardware-software system. Many features that would seem essential to the operation of actual time-shared systems are not present in the Markov model: swapping (except as a constant overhead factor), quantum time, priority scheduling, etc. Moreover, the distributions of think time and processor time per interaction are fit in the model by exponentials with the proper means. The fact that all of this detail may be omitted and still leave a model that accurately predicts mean response time is startling. The implication is that only mean think time, mean processor time (including swapping), and the number of users interacting with the system are of first-order effect, and that the rest of the details have second- or third-order effect.

It might be possible to obtain better mean response times than those predicted by the Markov model with the use of a "shortest job first" procedure. The ability to predict the length of a job accurately is essential to such a scheduler. As the accuracy of its prediction mechanism is reduced, performance rapidly approaches the level of a first-come first-served or random scheduler.

The chief effect of the quantum size (except in predictive schedulers) is to change the swapping overhead and thereby affect the mean response time. In a system where the quantum time has no effect on the swapping overhead (such as in a system with complete swapping overlap), the quantum size has no effect on the mean response time. The quantum size, however, always has an

effect on the shape of the dependence of response times on processor time per interaction.

There is no reason to believe that the Markov model for a multiple-processor system serving a single queue of users is not just as accurate as the single-processor model. The results it predicts are completely reasonable: that for equal capacity systems the performance of the single-processor system will be better than that of the two-processor system. Furthermore, given equal capacity systems with the same number of processors, the system with the fewest number of queues will yield the best performance. Extrapolating these results, the conclusion can be made that, for equal capacity systems, the smaller the number of processors and queues, the better the mean response time. The observation should be made that at saturation, all equal capacity systems have the same performance, and a system with fewer processors should require no less main storage or swapping activity than a system with more processors. However, if traditional computer economics hold, the fastest possible processor will provide the best processing capacity per dollar.

Asymmetric multi-processor systems, appearing to yield the worst performance from the equal capacity comparison, begin to be very attractive with the introduction of economic considerations. In the example discussed (see Part 4.4.2), the performance of CTSS could have been improved to the point of allowing approximately 45 users to be served with the same mean response time as 30 with the addition of hardware representing less than ten per cent of the monthly rental of CTSS (an IBM 1400 series machine and an additional channel, file control unit, and access arms for the 1302 disk).

What constitutes acceptable response time is a question that is completely open to discussion. However, there is certainly no relationship between the point at which a system saturates and whether or not the response time is at that point are acceptable. Thus, the question of whether it is desirable to operate a system in saturation must be settled by other considerations. Given a maximum acceptable mean response time, the maximum number of users to be served, and a specification of user characteristics, the most desirable system meeting these requirements will be the cheapest. And the cheapest system is very likely to be the one with the lowest processing capacity consistent with meeting the specifications. This system will be run in saturation.

So far in this discussion, most of the results cited were derivable from considerations of the CTSS data and the Markov models. There are, of course, many factors which the Markov model cannot be made to predict, and a more detailed model must be used. As was seen, the simulation models were able to predict the distribution of response times, hardware usage, response time as a function of processor time required, and so on. Most of these parameters were discussed sufficiently as they were presented. Perhaps the most interesting result of the simulations is the fact that good accuracy can be obtained from a fairly simple model (when compared to the actual system). The biggest problem in simulation modeling, as in all model building, is to retain all "essential" detail and remove the nonessential features. It is felt that the efforts in this direction in the case of the simulation models was successful.

Summarizing the over-all accomplishments of this research, the proof that time-shared systems and users can be accurately modeled is of first importance. Secondly, if mean response time is of primary interest, simple mathematical models can be constructed having good generality and yielding accurate results. Excellent predictions of parameters other than mean response time can be obtained from more detailed but still highly simplified simulation models.

With the advent of new time-sharing techniques and systems, much more analysis and measurement must be accomplished if these systems are to be designed and used intelligently. It is clear to the author that no small effort should be expended in order to monitor and measure the performance and use of any operating time-shared system. It is through intelligent use of this type of data that improvements can be made to present systems and that proposed systems can be evaluated.

Appendix A

DESCRIPTION OF CTSS

A.1. The CTSS Hardware

Simply stated, the CTSS configuration is an IBM 7094 (Model I) computer, augmented by disk and drum storage, connected through an IBM 7750 transmission controller to users' remote consoles, each consisting of a keyboard and printer. By typing at the console, a user may communicate with either the CTSS Supervisory Program or a program activated by this Supervisor. A line of input interpreted by the Supervisor is called a "command." Commands cause programs to be loaded from disk storage. These programs are queued, and each is executed for a short period of time, not necessarily to completion. The sequence of programs to be run and the duration of each "burst" of processor time is determined by a subroutine in the Supervisor called the "Scheduling Algorithm." The following is a list of the hardware elements in the system and a brief description of the function of each.

1. Central Processing Unit (CPU) - normal 7094 with the addition of automatic relocation and memory protection. Automatic relocation is not used on the present system as all programs are loaded beginning with location zero. In the protection mode, all memory references are checked against pre-set "memory bounds"; and if an out-of-bounds reference occurs, the CPU is trapped. Programs are given only enough space as is required. The limits on this space are location zero and the memory bound. In addition, the execution of any channel operating or tape handling instructions causes a protection mode violation and a trap to the Supervisor. Also, an Interval Timer is attached. This device can cause a trap after a program-specified time interval. The time unit is 1/60-th of a second.

2. Core Memory - 2 modules of 32,768 36-bit words. The two units are designated "Core A" and "Core B." These units do not operate independently, and only one of them can operate during a

given cycle. The access time is 2 microseconds. Core A holds only the CTSS Supervisory Program and Core B is used to hold users' programs as they are being executed.

3. The system configuration during the time the data was taken consisted of up to seven I/O channels:

A. Conventional tape channel with six tapes, printer, punch, and card reader. Used for batch processed programs only.

B. Conventional tape channel with six tapes. Used for batch-processed programs only.

C. Direct Data Channel - used to connect non-standard I/O devices to system. At present, only the Electronic System Laboratory's Display Console is connected. This unit consists of a CRT, light pen, buttons, etc. Displays are originated and maintained by the 7094, but light pen tracking and coordinate rotation, translation, and expansion are done by local circuitry.

D. Disk and Drum Channel and File Control - A 1301 Model 2 disk file and a 7320 drum were connected. The 1301-2 contains 20,000 tracks of 466 words each. Access time consists of arm positioning, an average of 165 ms., and rotational delay, 19 ms. Words are transmitted at a rate of 66.6 microseconds/word. This disk was replaced by a 1302 Model 2 disk on January 12, 1965, but because of the way it was used its characteristics remained practically the same as the 1301-2 for the period of the monitoring. The 1302-2 has a capacity of 40,000 tracks. A 466 word track size was used (half capacity) to maintain system continuity while new system programs were developed. The drum has a capacity of 186,400 words (400 tracks). Reading heads are switched electronically so that the average access time is just the average rotational delay, 8.6 ms. The transmission rate is approximately 30 microseconds/word.

E. IBM 7750 Transmission Control - a stored program computer used to coordinate input-output through a telephone switchboard to user's consoles, IBM 1050's and Teletype Model 35's. Both of these units type output at approximately 10 characters per second, including the timing for carriage returns.

F. Disk and Drum Channel and File Control - Same as Channel D.

G. High speed Drum Channel and File Control - A model 7320A drum is used with a capacity of 208,608 words, access time of 8.6 ms. average, and transmission rate of approximately 9 microseconds/word. On the installation of this channel (September 14, 1964), Channel F was removed and its disk and drum files were placed on Channel D. The 1302 Model 2 replaced both 1301-2 disks on January 12, 1965.

The almost 20 million words (40,000 tracks) of disk file storage are primarily for the use of the approximately 300 CTSS users at Project MAC. Each user is allotted between 50 and about 1000 tracks of space for his own use. Typically, a users's tracks will contain files of source decks of programs in MAD, FAP, FORTRAN, etc., binary decks ready for the conventional BSS loader, data files, core-images, etc. A core-image is an exact copy of the contents of the core memory after a program has been loaded, linked, and possibly run, along with the status of the CPU. The core-image is only as long as it has to be; unused cells are not saved. The CTSS Supervisory Program has an area on the disk which is used to hold the core-images of nearly 80 command programs, a library for the BSS loader, etc.

Drum storage is used to hold core-images of user programs queued for execution. When a user's program is selected for running, it is transferred from the drum to Core B. The core-image previously in Core B is dumped to the drum. However, only the CPU status and enough of the old core-image to make room for the new one is placed on the drum. All core-images are always loaded into Core B starting at the first location, thus there will never be more than one complete core-image in Core B at a time. Other core-images, as many as four, may be split between Core B and drum storage. This process of dumping and loading user's programs is called "swapping."

A.2. The CTSS Software

Users are assigned one of six internal states:

0. Dead. User has no core-image on the drum and is not waiting for service.

1. Dormant. User has a core-image on the drum, but is not waiting for service.

2. Working. User has a core-image and is being served. That is, his program is either being run by the CPU or it is in

the queues waiting for service.

3. Waiting Command. User has no core-image and is waiting to be given service for the first time. A core-image will be obtained from the disk and when loaded, the user's status will be changed to "Working".

4. Input Wait. The user's program requires console input. The core-image is on the drum, and the user is no longer in the queues. Upon completion of a line of input, the user is placed back into the Working state.

5. Output Wait. The user's program has filled its console output buffers and attempted to add more. After the buffer empties to a certain point, the user is returned to the Working status and further output can occur. While a user's program is in Output Wait, the core-image is on the drum, and the user is no longer in the queues.

The difference between the Dead and Dormant states is simply whether or not a user is left with a core-image after a command program finishes. Most commands finish in the Dead status. However, commands for the loading, linking, and running of BSS files leave a core-image. This is done so that post-mortems, traces, etc. may be initiated. In addition, any program can be stopped by the user from his console by typing a special character sequence. This action, called a "quit," puts the program into Dormant. It may be restarted at any time by the appropriate command. A core-image may be saved on the disk in a permanent file and then restored at a later time by the appropriate sequence of commands.

Swapping and the use of the CPU are non-overlapped in the version of CTSS studied. During either of these activities, control is returned to the Supervisor via traps for various reasons. Every time a character is received at the 7750 console, the 7750 channel traps the 7094 CPU. The input character along with a source identification is immediately transmitted to Core A, with no processing being done. Every 200 milliseconds a clock trap occurs. At this time all input characters received since the previous clock trap are sorted by user and appended to the corresponding input line. Any user status changes brought about by the completion of an input line are communicated to the Scheduling Algorithm. Output from programs is handled in much the same way except that the buffering is in the 7750.

Whenever the Scheduling Algorithm determines that a user's program is to be stopped or pre-empted, the swapping procedure is started. Priorities are assigned user's programs on the basis of length and previous running time. In order to keep efficiency up, longer programs are given slightly less priority because they require longer swapping time. Longer programs, when they finally are to run, are usually allowed a longer period of CPU time. In more detail, the Scheduling Algorithm has nine queues in which users wait. These queues are ordered; the zeroth queue has the highest priority, and the eighth, the lowest. When users enter the Command Wait status from Dormant via a command, a queue assignment is made based on the size of the command core-image. If the memory bound is greater than 4096, queue number three is assigned, otherwise queue number two. The next user to run is always selected from the highest priority, nonempty queue. If a user remains in the same queue for more than 60 seconds without being run, he is moved to the end of the next higher priority queue.

A user is normally allowed a burst of CPU time equal to .5 seconds multiplied by 2 to the power of the user's queue number. Thus, a user of level 3 would normally run for (.5) (2) (2) (2) = 4 seconds. If a user exceeds this time while running, he is removed from his present queue and placed at the end of the next lower priority one. A user is pre-empted, that is, another user will be swapped in, if the current user is no longer the first user in the queues, and he has run as long as the new first user will run, computing the burst as above. When a user returns to the Working status from Input or Output Wait, his queue number is re-computed by size as above, but it remains the same if it is already as good as or better than queue 2. If a user goes from Dormant to Working via a program generated "sleeping period," the old priority level is used. If a program generated command is encountered, the new queue is either the old one or is computed by size, whichever yields the lower priority. A listing of the Scheduling Algorithm appears below.

```
SCDA    R********** TIME SHARING SCHEDULING ALGORITHM ***************
        R     T. HASTINGS AND R. DALEY
        R
        R  THE SCHEDULING ALGORITHM PERFORMS THE FOLLOWING FUNCTIONS
        R
        R   1. DETERMINES WHICH USER IS TO RUN NEXT
        R   2. DETERMINES WHEN NEXT USER IS TO RUN
        R   3. DETERMINES HOW LONG NEXT USER IS TO RUN
        R   4. CHARGES USERS FOR SWAPPING AND RUNNING TIME
        R   5. KEEPS TRACK OF THE STATUS OF EACH USER
        R
        R  THE SCHEDULING ALGORITHM IS CALLED FROM THE SUPERVISOR BY
        R   EXECUTE SCHED.(EVENT, USER, ARG)
        R     AFTER ALL TRAPS HAVE BEEN DISABLED
        R  'USER' IS BETWEEN 0 AND THE MAX. NO. OF USERS, 'MXUSR
        R  THE SIGNIFICANCE OF 'USER' AND 'ARG' DEPEND ON 'EVENT'
        R     OR ARE MEANINGLESS AS DESCRIBED BELOW
        R     'EVENT'   DESCRIPTION
        R     0         INITIALIZATION OF SCHED.
        R     1          CLOCK INTERRUPT
        R     2         'USER' HAS CHANGED TO STATE 'ARG'
        R     3         BEGINNING OF SAVING 'USER' CORE IMAGE
        R     4         BEGINNING OF RESTORING 'USER' CORE IMAGE
        R     5         'USER' BEGINS RUNNING, AFTER SWAP
        R     6         'USER' CORE IMAGE NOW HAS LENGTH 'ARG'
        R     7          OPERATOR SET BACKGROUND KEYS TO 'ARG'
        R     8         'USER' LOGGED IN, 'ARG' IS LINE MULTIPLIER
        R     9         'USER' LOGGED OUT
        R    10          IS 'NEWUSR' STILL RUNABLE
        R    11          'USER' DIALED UP COMPUTER
        R
        R  TO CLARIFY THE ORDER IN WHICH EVENTS HAPPEN, BLOCKS
        R     OF CODING BRACKETED BY COMMENTS HAVE BEEN PLACED IN
        R     TYPICAL ORDER OF EXECUTION FOR A COMMAND
        R  ALL TIME IS KEPT IN SIXTIETHS OF A SECOND AND VARIABLES
        R     ENDING WITH 'TIM' ARE TIMES SINCE SYSTEM WAS
        R     LOADED WITH THE EXCEPTION OF 'SYSTIM'
        R  SCHED. HAS SOLE RESPONSIBILITY FOR SETTING AND CHANGING
        R     THE FOLLOWING COMMON ARRAYS AND VARIALBES
        R
        R  THE FOLLOWING COMMON ARRAYS ARE USED
        R   'STATUS' - THE STATUS OF EACH USER
        R   WHERE STATUS(J) MAY BE
        R         0 DEAD - NOT WAITING TO RUN AND NO CORE IMAGE
        R         1 DORMNT - NOT WAITING TO RUN
        R         2 WORKING - WAITING IN QUEUES OR RUNNING
        R         3 WAITING COMMAND - WAITING IN QUEUES FOR COM.
        R         4 INPUT WAIT - PROGRAM WAITING FOR INPUT
        R         5 OUTPUT WAIT - OUTPUT BUFFERS FILLED
        R    'LENGTH' - LENGTH OF USER CORE IMAGE IN WORDS
        R    'LEVEL' - USER'S PRIORITY LEVEL(0, ... , 'MAXLVL')
        R    'TIMLEV' - ELAPSED TIME RUN AT CURRENT LEVEL
        R    'WATTIM' - THE LAST TIME THAT A USER BEGAN TO WAIT
        R    'LINMUL' - USER LINE MULTIPLIER
        R    'PLIST' - THE POSITION LIST SPECIFIES THE POSITIONS
        R             OF THE USERS WHICH ARE IN THE WORKING QUEUE
        R    'ULIST' - THE USER LIST INDICATES THE USER NUMBERS
        R             WHICH CORRESPOND TO THESE QUEUE POSITIONS
        R    'ENDPTR' - ENDPTR(J) IS END OF QUEUE J IN PLIST
        R    'NOTIME' - NOTIME(J) IS SET TO 2 IF USER INACTIVE
```

```
R   AND USER J WILL SUBSEQUENTLY BE LOGGED OUT
R  PB' - PERCENTAGE USER MUST RUN WHILE IN WORK STAT
R
R   THE FOLLOWING COMMON VARIABLES ARE USED
R 'MXUSRS' - MAX. NO. OF FOREGROUND USERS
R 'CURUSR' - CURRENT USER, RUNNING OR SWAPPING
R 'OLDUSR' - LAST USER TO BE RUN, WHEN 'SWAP' .NE. 0
R 'NEWUSR' - NEXT USER TO BE RUN, WHEN 'SWAP' .NE. 0
R 'PAYUSR' - THE USER CURRENTLY PAYING FOR TIME
R 'SYSTIM' - TIME SYSTEM WAS INITIALIZED
R 'SYSDAT' - DATE SYSTEM WAS INITIALIZED
R 'BEGTIM' - THE LAST TIME 'CURUSR' BEGAN TO RUN
R 'QUANTM' - MAXIMUM RUNNING TIME AT LEVEL 0
R 'MAXTIM' - USER RUNS AT SAME LEVEL UNTIL 'MAXTIM'
R 'TBASE' - BASE TIME FOR COMPUTING 'MAXTIM'
R 'PAYTIM' - LAST TIME A USER WAS CHARGED FOR TIME
R 'LEVTIM' - LAST TIME 'CURUSR' WAS RUNNING AT CURRENT LEVEL
R 'SWAP' - NON-ZERO REQUESTS SUPERVISOR TO RUN
R           'NEWUSR' AS SOON AS IT CAN
R 'MAXLVL' - THE MAXIMUM PRIORITY LEVEL(0 ... 'MAXLVL')
R 'MINLVL' - THE MINIMUM PRIORITY LEVEL ALLOWED
R 'FULLVL' - INIT. LEVEL FOR 'FULLEN' TO FULL CORE USER
R 'EMPLVL' - INITIAL LEVEL FOR EMPTY CORE USER
R 'FULLEN' - LENGTH FOR ENTRY AT LEVEL 'FULLVL'
R 'ESTTIM' - TIME LIMIT FOR CURRENT FIB JOB
R 'QNTWAT' - QUANTM WAITING TIME BEFORE LEVEL CHANGE
R            TO NEXT HIGHEST PRIORITY LEVEL
R 'LEVINC' - AMOUNT PRIORITY LEVEL IS INCREASED WHEN
R   USER RETURNS TO WORKING FROM INPUT OR OUTPUT WAIT
R 'INACTV' - MAX. TIME INACTIVE BEFORE LOGOUT
R 'HANGUP' - MAX. TIME BEFORE INACTIVE LINE IS HUNGUP
R
R   COMMON VARIABLES REFERRED TO BY SCHED. BUT
R     NOT SET OR CHANGED BY SCHED.
R 'BKGTIM' - TOTAL TIME BACKGROUND HAS RUN
R 'SWPSW' - NON-ZERO WHEN SUPERVISOR IS SWAPPING AND
R    COMMAND LOADING
R 'PROBN(J)' - NON-ZERO WHEN USER J IS LOGGED IN
R 'ADOPT(J)' - PROBN(J) .AND. ADOPT(J) .E. 1B, THEN
R   USER J IS ADOPTED
R
R   SCHED. CALLS THE FOLLOWING SUBROUTINES
R INITQ. - INITIALIZES QUEUES
R HEDUSR. - RETURNS THE HEAD OF QUEUE USER
R      AT HIGHEST NON-EMPTY PRIORITY LEVEL OR 0
R DELQUE.(J) - DELETES USER J FROM QUEUES
R ENDQUE.(J) - PLACES USER J AT END OF QUEUE LEVEL(J)
R BEGQUE.(J) - PLACES USER J AT BEG OF QUEUE LEVEL(J)
R ILOG2.(N) - RETURNS INTEGER PART OF LOG TO BASE 2 N
R I.(J) - CONVERTS FORWARD INDEX 'J' TO BACKWARD
R          INDEX FOR REFERRING TO MAD ARRAYS
R INITIM. - INITIALIZE TIME ACCOUNTING
R INTIM. - USER 'U' LOGGED IN
R OUTIM. - USER 'U' LOGGED OUT
R CHARGE.(U,T) - CHARGE USER 'U' FOR TIME 'T'
R GETOTL. - RETURNS THE TOTAL TIME SYSTEM HAS RUN
R DELTIM.(T) - RETURNS DELTA 'T' - THE DIFFERENCE
R      BETWEEN 'GETOTL.()' AND TIME 'T'
R      TIME 'T' IS ALSO SET TO GETOTL.(0)
R CURTIM.(0) - RETURNS THE CURRENT TIME SINCE MIDNIGHT
```

```
        R       OF DAY SYSTEM WAS INITIALIZED
        R   MONSC1.(EVENT, USER, ARG) MONITORS SCHED.
        R   MONSC2. IS CALLED WHEN SCHED. CHANGES COMMON
        R   PLOT1.(EVENT, USER, ARG) PLOTS SYSTEM ON ESL SCOPE
        R   PLOT2. IS CALLED WHEN SCHED. CHANGED COMMON
        R
          EXTERNAL FUNCTION(A, B, C)
          ENTRY TO SCHED.
          NORMAL MODE IS INTEGER
        R
        R.. SHORTEN LINKAGE, SETUP USER INDEX, CALL MONITORING SUB.,
        R..   CALL PLOTTING ROUTINE
        R..   ASSUME COMMON WILL BE CHANGED, AND DISPATCH ON 'EVENT'
        R
          EVENT = A
          USR = B
          IUSER = I.(USR)
          ARG = C
          EXECUTE MONSC1.(EVENT, USR, ARG)
          EXECUTE PLOT1.(EVENT, USR, ARG)
          MONITR = CHANGE
          STATEMENT LABEL MONITR, RETURN, CHANGE
          TRANSFER TO EVNT(EVENT)
        R
        R.. 'EVENT' .E. 0, INITIALIZE SCHEDULING ALGORITHM FOR N USERS
        R..   INITIALIZE INDEPENDENT COMMON VARIABLES
EVNT(0)           MXUSRS = 31
          MAXLVL = 8
          MINLVL = 0
          FULLVL = 3
          EMPLVL = 2
          FULLEN = 4096
          QNTWAT = 377777777777K
          LEVINC = 0
          QUANTM = 30
          INACTV = 216000
          HANGUP = 7200
        R
        R..   INITIALIZE QUEUES AND TIME ACCOUNTING
          EXECUTE INITQ.
          EXECUTE INITIM.
        R
        R..   INITIALIZE TABLES
          THROUGH JLOOP, FOR J = 0, 1, J .G. UMAX
            JUSER = I.(J)
            PB(JUSER) = 0
JLOOP                 LINMUL(JUSER) = 1
        R
          SYSTIM = CURTIM.(0)
          SYSDAT = DATEYR
          SWAP = 1B
          FIRST3 = 1B
          BGMAX = 120
          TRANSFER TO CHANGE
        R
        R.. SCHEDULE BACKGROUND (USER 0) AT LEVEL 9
EVNT(12)            EXECUTE DELQUE.(0)
          LEVEL (0) = 9
          STATUS (0) = 2
          TRANSFER TO CHANGE
```

```
          R
          R.. 'EVENT' .E. 1, CLOCK INTERRUPT
          R..   ASSUME COMMON WILL NOT BE CHANGED
EVNT(1)             MONITR = RETURN
            ICUR = I.(CURUSR)
            T = GETOTL.(0)
          R.. DO THE FOLLOWING CHECKING EVERY 10 SECONDS
          R..   CHARGE PAYING USER FOR TIME
          R..   MOVE LONG WAITING USERS UP IN PRIORITY
          R..   ALSO LOGOUT INACTIVE USERS
          R..   AND HANGUP INACTIVE LINES
          R..   DECREASE LEVEL OF USER NOT MEETING PERCENTAGE BY 1
          R..     PERCENTAGE CALCULATED ON TIME SINCE USER ENTERED
          R..     WORKING STATUS
            WHENEVER T .G. CHECKT
              CHECKT = T + 600
              EXECUTE CHARGE.(PAYUSR, DELTIM.(PAYTIM))
              THROUGH KLOOP, FOR K = 0, 1, K .G. UMAX
                WHENEVER K .E. CURUSR, TRANSFER TO KLOOP
                KUSER = I.(K)
                WHENEVER K .G. 2
                DELT = T - WATTIM(KUSER)
                WHENEVER STATUS(KUSER) .E. 3 .OR. STATUS(KUSER) .E. 2
                  WHENEVER DELT .G. QNTWAT .AND. LEVEL(KUSER) .G. MINLVL
                    MONITR = CHANGE
                    EXECUTE DELQUE.(K)
                    LEVEL(KUSER) = LEVEL(KUSER) - 1
                    EXECUTE ENDQUE.(K)
                    WATTIM(KUSER) = T
                    TIMLEV(KUSER) = 0
                  END OF CONDITIONAL
                OR WHENEVER PROBN(KUSER) .NE. 0
                  WHENEVER DELT .G. INACTV
                    MONITR = CHANGE
                    NOTIME(KUSER) = 2
                    WATTIM(KUSER) = T
                  END OF CONDITIONAL
                OTHERWISE
                  WHENEVER DELT .G. HANGUP .AND. ADOPT(KUSER) .E. 0
                    MONITR = CHANGE
                    NOTIME(KUSER) = 4
                    WATTIM(KUSER) = T
                  END OF CONDITIONAL
                END OF CONDITIONAL
                END OF  CONDITIONAL
                WHENEVER 100 * WRKTIM (KUSER) .L. PR (KUSER) * (TIMNOW -
         1     STRTIM(KUSER)) .AND. (STATUS(KUSER) .E. 2 .OR.
         2     STATUS(KUSER) .E. 3) .AND. LEVEL(KUSER) .G. MINLVL
                  MONITR = CHANGE
                  EXECUTE DELQUE.(K)
                  LEVEL(KUSER) = LEVEL(KUSER) - 1
                  EXECUTE ENDQUE.(K)
                END OF CONDITIONAL
     KLOOP          CONTINUE
            END OF CONDITIONAL
          R
          R..   MOVE LONG RUNNING 'CURUSR' DOWN IN PRIORITY
            WHENEVER T .G. MAXTIM .AND. .NOT. SWAP
              MONITR = CHANGE
              EXECUTE DELQUE.(CURUSR)
```

```
            WHENEVER LEVEL(ICUR) .L. MAXLVL,
          1      LEVEL(ICUR) = LEVEL(ICUR) + 1
            EXECUTE ENDQUE.(CURUSR)
            LEVTIM = T
            TIMLEV(ICUR) = 0
            MAXTIM = T + TRUN.(CURUSR, LEVEL(ICUR))
          END OF CONDITIONAL
          TRANSFER TO DECIDE
        R
        R.. 'EVENT' .E. 6, 'USR'('IUSER') CORE IS OF LENGTH 'ARG'
        R..   JUST BEFORE ENTERING WAITING COMMAND
        R..   OR LENGTH CHANGED WHILE RUNNING
EVNT(6)          LENGTH(IUSER) = ARG
          TRANSFER TO CHANGE
        R
        R.. 'EVENT' .E. 2, 'USR'('IUSER') CHANGED STATE
        R..   DISPATCH ON NEW STATE, IGNORE REDUNDANT TRANSITIONS
EVNT(2)          WHENEVER USR .NE. 0, TRANSFER TO STAT(ARG)
          TRANSFER TO RETURN
        R
        R..   'USR'('IUSER') BEGAN WAITING FOR A COMMAND
   STAT(3)          LEV = LEVELF.(LENGTH(IUSER))
          WHENEVER STATUS(IUSER) .E. 2 .OR. STATUS(IUSER) .E. 3
            WHENEVER LEV .G. LEVEL(IUSER)
              EXECUTE DELQUE.(USR)
              TRANSFER TO COMAND
            END OF CONDITIONAL
          OTHERWISE
    COMAND          LEVEL(IUSER) = LEV
            EXECUTE ENDQUE.(USR)
            TIMLEV(IUSER) = 0
            WATTIM(IUSER) = GETOTL.(0)
          END OF CONDITIONAL
          STATUS(IUSER) = 3
          TRANSFER TO DECIDE
        R
        R.. 'EVENT' .E. 10, IS 'NEWUSR' STILL RUNABLE
EVNT(10)          NEWUSI = I. (NEWUSR)
          WHENEVER STATUS (NEWUSI) .E. 2
        1    .OR. STATUS (NEWUSI) .E. 3, TRANSFER TO RETURN
          SWAP = 0B
          TRANSFER TO DECIDE
        R
        R.. THE NEXT THREE EVENTS ALWAYS OCCUR IN SEQUENCE
        R..   WHEN CONTROL IS TRANSFERRED FROM 'OLDUSR' TO 'NEWUSR'
        R..   AS A RESULT OF 'SWAP' BEING SET NON-ZERO.
        R.. 'NEWUSR' PAYS FOR CORE FREEUP AND HIS OWN LOAD
        R
        R.. 'EVENT' .E. 3, SAVING OF 'USR'('IUSER') IS BEGINNING
        R..   EVENT 3 MAY BE CALLED FOR ANY OF THE FOLLOWING:
        R     1. FREEING UP CORE B BECAUSE 'CURUSR' EXTENDED SIZE
        R     2. FREEING UP CORE A DRUM BUFFERS FOR SWAPPING
        R     3. DUMPING 'OLDUSR'
        R     4. DUMPING OTHER USERS TO MAKE ROOM FOR 'NEWUSR'
          BOOLEAN SWPSW, FIRST3, SWAP
EVNT(3)          WHENEVER SWPSW
            WHENEVER FIRST3
              FIRST3 = 0B
              CHARGE.(PAYUSR, DELTIM.(PAYTIM))
              OLDUSI = I. (OLDUSR)
```

```
                TIMLEV (OLDUSI) = TIMLEV (OLDUSI) + DELTIM.(LEVTIM)
                PAYUSR = NEWUSR
              END OF CONDITIONAL
              TRANSFER TO CHANGE
            END OF CONDITIONAL
            TRANSFER TO RETURN
          R
          R.. 'EVENT' .E. 4, RESTORING OF 'NEWUSR' IS BEGINNING
EVNT(4)           OLDUSI = I. (OLDUSR)
            WHENEVER STATUS (OLDUSI) .E. 2,
          1    WATTIM (OLDUSI) = GETOTL.(0)
            CURUSR = NEWUSR
            TRANSFER TO CHANGE
          R
          R.. 'EVENT' .E. 5, 'NEWUSR' BEGINS RUNNING AFTER RESTORE
          R.. NEWUSR'S TIME ALLOTMENT IS SET TO THE QUANTUM AT THIS
          R.. LEVEL LESS HIS TWO-WAY SWAP TIME FROM DRUM, AND LESS
          R.. ANY TIME ALREADY RUN AT THIS LEVEL.
EVNT(5)           TDEL = DELTIM.(PAYTIM)
            WHENEVER TDEL .G. BGMAX
              EXECUTE CHARGE.(PAYUSR, BGMAX)
              EXECUTE CHARGE.(OLDUSR, TDEL-BGMAX)
            OTHERWISE
              EXECUTE CHARGE.(PAYUSR, TDEL)
            END OF CONDITIONAL
            NEWUSI = I. (NEWUSR)
            WHENEVER STATUS (NEWUSI) .E. 3
              STATUS (NEWUSI) = 2
              WRKTIM (NEWUSI) = 0
              STRTIM (NEWUSI) = TIMNOW
            END OF CONDITIONAL
            BEGTIM = GETOTL.(0)
            LEVTIM = BEGTIM
            MAXTIM = BEGTIM + TRUN.(NEWUSR, LEVEL (NEWUSI))
          1    -LENGTH (NEWUSI)/1024 - TIMLEV (NEWUSI)
            SWAP = 0B
            FIRST3 = 1B
            TRANSFER TO DECIDE
          R
   STAT(4)
          R..  'USR'('IUSER') ENTERED INPUT WAIT.
          R
   STAT(5)
          R..  'USR'('IUSER') ENTERED OUTPUT WAIT.
          R
   STAT(6)
          R..  'USR'('IUSER') ENTERED FILE WAIT
          R     TRIED TO ACCESS INTERLOCKED FILE.
          R
   STAT(7)
          R     USER (FIB) ENTERED FIB WAIT.
          R     FIB CANNOT GO DORMNT BUT CAN SCHEDULE ITSELF
          R     DORMNT.  THUS, STATUS 7 IS TOTALLY UNNECESSARY.
          R
            WHENEVER STATUS(IUSER) .E. 2
              EXECUTE DELQUE.(USR)
              STATUS(IUSER) = ARG
              WATTIM(IUSER) = GETOTL.(0)
              TRANSFER TO DECIDE
            END OF CONDITIONAL
```

```
          TRANSFER TO RETURN
        R
        R..    'USR'('IUSER') TO BEGIN WORKING AFTER I/O WAITING
        R..      OR ALARM CLOCK RETURN FROM DORMANT TO WORKING
STAT(2)            WHENEVER STATUS(IUSER) .GE. 4 .OR. STATUS(IUSER) .E. 1
        R..INITIALIZE TIME USER HAS WORKED (WRKTIM)
        R..SET TIME USER ENTERED WORKING STAT (STRTIM)
          WRKTIM(IUSER) = 0
          STRTIM(IUSER) = TIMNOW
            WHENEVER STATUS(IUSER) .NE. 1
               LEVEL(IUSER) = LEVELF.(LENGTH(IUSER))
               TIMLEV(IUSER) = 0
            END OF CONDITIONAL
            EXECUTE ENDQUE.(USR)
            WATTIM(IUSER) = GETOTL.(0)
            STATUS(IUSER) = 2
            TRANSFER TO DECIDE
          END OF CONDITIONAL
          TRANSFER TO RETURN
        R
        R
        R..    'USR'('IUSER') WENT DORMANT WHILE RUNNING
        R..      OR PUSHED QUIT BUTTON
STAT(1)           WHENEVER STATUS(IUSER) .L. 7
           EXECUTE DELQUE.(USR)
           STATUS(IUSER) = 1
            WATTIM(IUSER) = GETOTL.(0)
           WHENEVER USR .E. CURUSR, TRANSFER TO DECIDE
           TRANSFER TO CHANGE
          END OF CONDITIONAL
           TRANSFER TO RETURN
        R
        R..    'USR'('IUSER') WENT DEAD, EVENT 6 WILL NOT OCCUR
STAT(0)           WHENEVER STATUS(IUSER) .L. 6
           EXECUTE DELQUE.(USR)
           STATUS(IUSER) = 0
           WATTIM(IUSER) = GETOTL.(0)
           TRANSFER TO DECIDE
          END OF CONDITIONAL
           TRANSFER TO RETURN
        R
        R
        R.. 'EVENT' .E. 7, OPERATOR SET KEYS TO 'ARG'
EVNT(7)           KEYS = ARG
          BACKGR = ARG
          TRANSFER TO DECIDE
        R
        R.. 'EVENT' .E. 8, 'USR'('IUSER') LOGGED IN PROPERLY
EVNT(8)           LINMUL(IUSER) = ARG
          EXECUTE INTIM.(USR)
          TRANSFER TO CHANGE
        R
        R.. 'EVENT' .E. 9, 'USR'('IUSER') LOGGED OUT
        R.. THIS EVENT IS NOW CALLED FROM CORE-B LOGOUT
        R.. AND MAY THEREFORE BE INTERRUPTED AFTER CALL
        R.. BUT BEFORE LOGOUT IS COMPLETE.
        R.. THE PROBLEM NO OF THE USER HAS NOT BEEN SET
        R.. TO ZERO WHEN THIS CALL TAKES PLACE.
EVNT(9)           EXECUTE OUTTIM.(USR)
          TRANSFER TO CHANGE
        R
        R.. 'EVENT' .E. 11, 'USR'('IUSER') DIALED UP COMPUTER
EVNT(11)          WATTIM(IUSER) = GETOTL.(0)
          NOTIME(IUSER) = 0
          TRANSFER TO CHANGE
```

```
          R
          R.. COMMON EXIT FROM SCHED.
          R.. DECIDE IF IT IS TIME TO RUN A NEW USER
          R
          R..   NO DECISION WHILE SWAPPING
DECIDE              WHENEVER SWAP, TRANSFER TO MONITR
          R
          R
          R.. WHENEVER NO-INTERRUPT SWITCH IS ON
          R.. USER CANNOT BE SWAPPED
            CURUSI = I. (CURUSR)
            WHENEVER STATUS (CURUSI) .E. 2 .AND. USWICH (CURUSI)
          1.A.NINTBT .NE. 0, TRANSFER TO MONITR
          R
            U = HEDUSR.(0)
            WHENEVER BACKGR .NE. 0 .OR. KEYS .NE. 0 , U = 0
          R
          R..   RUN USER 'U' IF 'CURUSR' HAS RUN AS LONG AS 'U' WOULD
            WHENEVER U .NE. CURUSR .AND.
          1    (PREMPT.(TRUN.(U, LEVEL(I.(U)))) .OR. CURUSR .L. 3)
          2    .OR. STATUS(I.(CURUSR)) .NE. 2 .OR. BACKGR .NE. 0
               MONITR = CHANGE
               SWAP = 1B
               NEWUSR = U
               OLDUSR = CURUSR
               BACKGR = 0
            END OF CONDITIONAL
          R
          R.. CALL MONSC2. IF COMMON CHANGED, ELSE JUST RETURN
            TRANSFER TO MONITR
CHANGE            CONTINUE
            EXECUTE MONSC2.
            EXECUTE PLOT2.
RETURN            FUNCTION RETURN
          R
          R
          R.. INTERNAL FUNCTIONS
          R..   'TRUN' - COMPUTES RUN TIME FOR USER 'DU' AT LEVEL 'DL'
            INTERNAL FUNCTION TRUN.(DU, DL) = QUANTM*(2.P.DL)
          R
          R..   'LEVELF' - COMPUTE PRIORITY LEVEL BASED ON LENGTH 'LEN'
            INTERNAL FUNCTION(LEN)
            ENTRY TO LEVELF.
            WHENEVER LEN .GE. FULLEN
              L = FULLVL
            OTHERWISE
              L = EMPLVL + ILOG2.(LEN/(FULLEN/(2 .P. (FULLVL-EMPLVL))))
            END OF CONDITIONAL
              FUNCTION RETURN L
            END OF FUNCTION
          R
          R..   'PREMPT' - IS TRUE IF PREMPTION IS PERMITTED
          R..     BASED ON TIME INTERRUPTER WILL RUN 'INTRUN'
            BOOLEAN PREMPT.
            INTERNAL FUNCTION PREMPT.(INTRUN) =
          1    INTRUN .L. GETOTL.(0) - BEGTIM
          R
          R
          R.. COMMON VARIABLES
            INSERT FILE MADCOM
          R.. USER MACHINE CONDITIONS STATUS TABLE
          R..   (NOT REFERRED TO BY MAD PROGRAMS)
            END OF FUNCTION
```

Appendix B

ADDITIONAL CTSS DATA

This appendix contains statistics describing console input-output and the internal states of users as well as the relative usage of the CTSS Commands.

Table B.1 shows the measured steady-state probabilities of a user console state (idle, input, or output) and of the internal states (Dead or Dormant, Working or Command, Wait, Input Wait, and Output Wait). For example, the probability of finding a user's console idle with an internal state of Dead or Dormant is .276. The probability of an idle console (regardless of internal state) is .502 , etc. Table B.2 shows the mean occupancy times for these states.

Table B.3 gives a list of each of the CTSS console commands, separated by task type with the relative usage in a sample of 110,050 commands.

TABLE B.1

Steady-State Probabilities

	Internal State				
Console State	Dead or Dormant	Working or Command Wait	Input Wait	Output Wait	Total
Idle	.276	.136	.090	0	.502
Input	.057	.005	.119	0	.182
Output	.114	.038	.059	.105	.316
Total	.448	.179	.268	.105	1.000

TABLE B.2

Mean Occupancy Time (seconds)

	Internal State			
Console State	Dead or Dormant	Working or Command Wait	Input Wait	Output Wait
Idle	36.8	8.6	11.7	
Input	7.6	0.3	15.5	
Output	15.2	2.4	7.6	77.8

TABLE B.3

Relative Frequency of Command Usage

Commands of Type 1 (File Manipulation)		Commands of Type 2 (Program Input and Editing)		Commands of Type 3 (Program Running and Debugging)		Commands of Type 4 (Compilation and Assembly)		Commands of Type 5 (Miscellaneous)	
ARCHIV	.017	CTEST9	.001	CTEST1	.001	CTEST2	.011	GENCOM	.000
CHMODE	.018	REVISE	.000	CTEST3	.006	CTEST6	.001	MODIFY	.001
COMBIN	.011	INPUT	.010	CTEST4	.002	CTEST8	.000	RESUME	.187 (includes R)
COMFIL	.018	EDIT	.040	FAPDBG	.005	DYNAMO	.001	RUNCOM	.009
CRUNCH	.016	FILE	.045	LOADGO	.031	MADTRN	.010	RUNOFF	.008
CTEST5	.000	ED	.045	MADBUG	.001	BEFAP	.000	DITTO	.001
CTEST7	.000	TFILE	.002	NCLOAD	.003	BLODI	.000	MEMO	.000
DELETE	.060			OCTPAT	.001	COMIT	.000	SD	.004
EXTBSS	.001	Total	.142	OCTTRA	.000	GPSS	.000	SP	.000
PRINTF	.060			RESTOR	.003	LISP	.002	SAVE	.060
REMARK	.000			LDABS	.000	AED	.002	TYPSET	.009
RENAME	.014			OCTLK	.008	FAP	.013		
RQUEST	.017			PATCH	.000	MAD	.037	Total	.278
LISTF	.058			LOAD	.009	OPL	.000		
PRBIN	.002			PM	.006	SNOBOL	.001		
PRBSS	.005			TRA	.000				
PRINT	.014			USE	.003	Total	.086		
COPY	.021			START	.029				
LOG	.000			VLOAD	.005				
SPLIT	.002			STOPAT	.000				
UPDATE	.010			STRACE	.001				
UPDBSS	.004								
				Total	.114				
Total	.349								

Note: LOGIN, LOGOUT, etc. were not included in these counts and account for .030 of the usage.

Appendix C

THE SIMULATION PROGRAMMING SYSTEM

This section describes a means for preparing and running programs to simulate digital computer systems. A language, based on the Michigan Algorithm Decoder (MAD, see ARDEN, 1963) and a scheme for organizing such programs is presented. The simulation language is based both on the way digital systems are organized and on the methods a designer uses to specify such a system. The organization of the operating system for a simulation is explained and the specific implementations are discussed. Finally, this simulation programming system is compared to several other widely used ones. Examples and information required to use the system are also presented.

C.1 Organization of a Simulation

The design of a large digital computer system is generally divided into several parts. For example, the memory system is usually not designed by the same people who design the instruction processing unit or the input/output channels and devices. Moreover, the software and environment aspects fall into completely different provinces. Therefore, it is felt that this same modularity should be preserved in the specification of a simulation of such a system. Such an organization would have several advantages: (1) different people could independently specify the simulation programs for their modules, needing to work together only to the extent that the original designers did (ideally, the designer of the module could also write the simulation program); (2) a change in the internal operation of one of the modules or elements in the simulation would not disturb the rest; (3) the traditional, well-known organization of a digital computer system can be maintained thereby increasing the understandability of the simulation program; etc.

The place of the software parts of the system is clear in simulations with great detail: The programs are simply loaded into the memory simulation element and executed. However, in a less detailed simulation, the software functions can either be

distributed among the various hardware elements or, more naturally, be lumped into the control section of the simulated system.

The specification of a program to simulate a large system is now reduced to the specification of the elements which comprise the system. Two factors must be defined in this specification: the series of events occurring within the element and the timing of these events with respect to events occurring in the remainder of the system. Note that this problem is identical to the one faced by the original system designer.

The implementation of the element specification as a module in a simulation program should involve nothing more than expressing the series of events and their timing in some convenient notation. Since most events within elements of a computer system require some computation, an algebraic programming language such as FORTRAN, MAD, etc., is sufficiently well-known and convenient to provide this capability. Communications with other elements within an event could be accomplished by "sending" outputs from one element to another. Timing, then, depends on the occurrence of inputs, fixed time increments, variable increments, and combinations of these. The elements specification language to be developed incorporates communications with other elements and the definition of timing with a conventional algebraic language in a natural way. Translation in this language from the timing charts and flow diagrams of the hardware designer should then be as natural as that from software flow diagrams to an algebraic language.

Looking at the simulation structure from an overall view, a digital system will be represented by a number of programs, one for each element of the system. Since the real elements operate simultaneously, their programs should at least appear to run simultaneously. Thus, element specifications can be written as if they will be operating in parallel with the rest of the system, thereby preserving the organizational relationships of the original system.

The element specification language itself is an augmented version of MAD. It includes all of the MAD declarations and statements plus additional statements for defining timing and communication. The translation to machine language is accomplished by a preprocessor and the normal MAD compiler. Element specifications become relocatable subroutines (in the

usual sense). The system is implemented for use on both CTSS and the IBM 7090 Fortran Monitor System.

C.2. System Variables

One of the factors of an element specification is the definition of its interconnection with other elements. For this purpose a special type of variable is introduced, the "system variable." As in real digital systems, elements may have inputs and outputs, an output originating from a single element and fanning out to one or more others. System variables must be referred to by the same name wherever they are used and must be declared as either input or output system variables by each element specification in which they appear. Two statements are provided for the declaration of system variables:

OUTPUT VARIABLES 'list' (1)

INPUT VARIABLES 'list' (2)

where 'list' is a list of MAD variable names separated by commas. The mode (i.e., floating-point, integer, etc.) of a system variable may be declared at the programmer's option; however, it should be the same mode in all elements. A system variable may be an array; but if it is multidimensional, it must have the same dimension vector in every element in which it occurs.

There are no restrictions on the naming of system variables other than those of the MAD compiler. Input system variables must not appear on the left side of a substitution statement. Moreover, no system variable can be declared as an output of more than one element. In operation, whenever a MAD substitution statement having an output variable on the left side is executed, the new value is instantaneously transmitted to the elements where this system variable is an input.

C.3. Timing of Events

As was previously stated, an event consists of communication, next event selection, and normal computation. By use of the system variable declarations and the statement repertoire of MAD these functions can be accomplished.

Timing may be specified in two different types of statement. In every case, the last statement of one event and the first statement of the next are separated by a timing statement. First, the time between events may depend on the state of the element at the beginning of this delay. Such delays can be specified by either:

DELAY OF 'expression' (3)

WAIT UNTIL 'expression' (4)

where 'expression' may be any legal MAD arithmetic expression. A statement of type (3) causes a delay between events equal to the value of 'expression.' The type (4) statement causes a delay between events such that the event occurs when the value of simulation "time" reaches the value of 'expression.' Thus, simulation time is incremented by the value of the delay as the delay statement is executed. Moreover, the state of the remainder of the elements in the simulation is brought "up to date" during this execution.

Alternately, the delay between events can depend on the time of the arrival of inputs from other elements. Such delays are specified by the following statement:

WAIT FOR INPUT (5)

where the delay is equal to the increment necessary to bring the simulation time up to the point of the next input from another element. In dealing with elements which have many inputs from various sources, the following statements are useful:

WAIT FOR INPUT $'input variable name'$ (6)

WAIT FOR INPUT FROM $'element name'$ (7)

Statement type (6) causes a delay until the specified input variable is sent, type (7), until an input originates from the specified element. Neither (6) nor (7) have been implemented in the present system, but experience has indicated that they would be useful if available. Timing can be specified by combinations of the two basic types of delay statements. For example,

WAIT FOR INPUT OR UNTIL 'expression' (8)

causes a delay until the time of the next input or until the point where simulation time reaches the value of 'expression', whichever occurs first. The statement

WAIT FOR INPUT AND UNTIL 'expression' (9)

causes a delay until the time of the next input or until the point where time equals 'expression', whichever occurs last. Similarly, the following statements are also allowed:

WAIT FOR INPUT OR DELAY OF 'expression' (10)

WAIT FOR INPUT AND DELAY OF 'expression' (11)

and their use is similar to that of (8) and (9).

C.4. Element Specification-Form and Restrictions

The first line (exclusive of comments) of an element specification defines the name of the element:

ELEMENT 'name' (12)

where 'name' is any legal MAD variable name. At the start of the simulation, time zero, control passes to the statement just after the element name defining statement (type 12). Any desired initialization should be placed between this point and either the execution of the first delay statement or the first appearance of an output variable on the left side of a substitution statement. Also, all system variable declaration statements (types 1 and 2) must occur in this initialization segment and must not be placed in such a way as to cause them to be executed more than once. The last line of an element specification is:

END OF SPECIFICATION (13)

All lines following this statement will be ignored.

In addition, there are two other items of importance. Any element specification may refer to a variable named "TIME" whose value is always the current value of simulation time. The mode of TIME must be declared by the programmer (or left as normal mode). In any case, this mode (integer or floating-point) must be the same in every element in a simulation. Furthermore, care must be taken that if the mode of TIME is integer, all expressions in the delay statements (types 3 - 11) must be integer expressions and not of mixed mode. This restriction is due to a foible of the MAD compiler. Care should be taken that the smallest non-zero time increment used in a delay statement with floating-point TIME is never so small that it will be truncated when added to TIME. For integer time, TIME may go up to a 35-bit number if care is taken and up to a 27-bit number in any case. A second variable named "INPUT" is available. It is of the Boolean mode and its value is "1B" if and only if an input was received during the previous delay. That is, INPUT is reset to "0B" at the beginning of any delay and set to "1B" on the arrival of any input.

Element specifications are compiled in the form of MAD external functions. Internal functions containing delays, etc., may be used; but outside of an internal function definition, no use should be made of the following MAD statements:

```
FUNCTION RETURN
ENTRY TO 'name'.
END OF FUNCTION
END OF PROGRAM
```

Furthermore, should control pass to the 'END OF SPECIFICATION' statement or to a FUNCTION RETURN, the simulation will immediately stop.

Diagnostic printouts from either the MAD preprocessing program, the MAD compiler, or the simulation operating system will occur in the event that an error occurs. Most of the restrictions are covered in this way.

A final area of trouble is the use of output variable setting substitution statements as the last statement in a "THROUGH" loop. The exact reason for this problem is easily solved by making the last statement in such loops "CONTINUE".

C.5. Examples of Element Specification

The following are three examples of element specification. They are fairly simple but serve to illustrate some of the features of the simulation language.

1. Oscillator

This element will have no inputs and a Boolean output named "TRIG" which oscillates with a period of 10.

```
          ELEMENT OSC                    first line.
          OUTPUT VARIABLES TRIG          initialization.
          TRIG = 0B                      initialize TRIG.
L         DELAY OF 5.                    wait half the period.
          TRIG = .NOT. TRIG              change output.
          TRANSFER TO L                  continue.
          BOOLEAN TRIG                   mode declaration.
          END OF SPECIFICATION           last line.
```

TIME, in the above element, is floating point.

2. Gated Oscillator

This element is the same as in Example 1, except that it has a Boolean input, GATE, which, when 1B causes the oscillator to start. When GATE is reset to 0B, the oscillator will immediately stop and its output return to zero.

```
          ELEMENT GATOSC
          INPUT VARIABLES GATE           initialization.
          OUTPUT VARIABLES TRIG
L1        TRIG = 0B
L2        WAIT FOR INPUT                 wait for GATE to come up.
          WHENEVER .NOT. GATE, TRANSFER TO L2
L3        WAIT FOR INPUT OR DELAY OF 5.  wait half period or for
```

```
        WHENEVER INPUT, TRANSFER TO L1   GATE to come down.
        TRIG = .NOT. TRIG                change output.
        TRANSFER TO L3                   continue.
        BOOLEAN TRIG, GATE
        END OF SPECIFICATION
```

The simplifying assumption is made that after GATE comes up the only input arriving will be GATE coming down. Notice that if the "OR" of the compound delay at L3 were made an "AND" the current output pulse would always be finished completely before the oscillator turned off.

3. Memory

The following element specification will simulate a 1000 word memory with an access time of 2 units. Assuming three inputs:

ADDR The memory address being referred to.

MEMIN The word to be stored by the memory.

MEMGO The "go" signal. Its value is zero for no action, positive to write and negative to read. When it changes to non-zero, the other inputs are assumed to be set up.

and two outputs:

MEMOUT The word being read out by the memory.

MEMRDY The ready signal from the memory. It is non-zero when ready.

The element specification is

```
        ELEMENT MEM
        OUTPUT VARIABLES MEMOUT, MEMRDY  initialization.
        INPUT VARIABLES ADDR, MEMIN, MEMGO
        MEMRDY = 1B                      send fact that memory is
       R                                   ready.
 L1     WAIT FOR INPUT                   wait for "go" signal.
        WHENEVER MEMGO .E. 0,            if "go" zero,
       1  TRANSFER TO L1                   resume waiting.
        MEMRDY = 0B                      turn off ready.
        DELAY OF 2                       wait access time.
        WHENEVER MEMGO .G. 0
          CONT(ADDR) = MEMIN             if writing, put new
       R                                   contents in.
        OTHERWISE                        if reading, output
          MEMOUT = CONT(ADDR)              requested cell.
```

```
          END OF CONDITIONAL
          MEMRDY = 1B
L2        WAIT FOR INPUT                  interlock, reception of
          WHENEVER MEMGO .NE.0,             ready should cause
         1  TRANSFER TO L2                  "go" to drop.
          TRANSFER TO L1                  go back and wait for the
         R                                  next service request.
         R                                remark card.
          NORMAL MODE IS INTEGER
          BOOLEAN MEMRDY
          DIMENSION CONT(1000)            dimension memory size.
          END OF SPECIFICATION
```

Notice that if the "go" signal were to come up again immediately after the interlock, continuous memory operation would result. In this case the ready signal will stay up for zero time. This is sufficient because of the interlock. TIME is used as an integer variable in this element.

The problem of interlocking signals is sometimes complex both in real and simulated systems. To insure that an output is received at more than one place it is usually most convenient to have this output hold its value for a finite length of time (perhaps 1 unit). Interlock techniques are usually a matter of taste and several different schemes are used in the element specifications shown.

C.6. The Simulation Operating System

The simulation operating system is a group of programs which cause the element specification programs to appear as if they are being simultaneously executed. Since events occur in instants of time, the only real problem is with simultaneous events. Events not occurring at the same time are simply queued in the order of their occurrence and sequentially executed. Events set to occur simultaneously are executed in an arbitrary, but not random, order.

Basically the operation of the system is a follows (let n be the number of elements in the system):

1. Let j = 1, let TIME = 0.

2. Enter the jth element at its entry point (just after the "ELEMENT 'name'" statement).

3. Control returns to the operating system via

   a. a system variable declaration statement. The name and location of the variable are added to the proper list and control

is returned to the element at the point from which it came.

b. a delay statement. The time for the next event, for this element, as specified by the delay statement, is placed in a list of next-event times. If the element is waiting for an input, a notation is made. The location of the delay statement is recorded. The next operation is step 4.

c. an output being sent; error, the simulation is stopped. (At least one delay must precede the first "send." This is taken care of by the translator.)

4. Let j = j+1, if j is less than n, go to step 2.

5. The list of next-event times is scanned for the lowest time. In case of duplicates, the first is used. If all elements are waiting for inputs, the simulation is stopped. TIME is incremented to the value of this nearest event. In case TIME is greater than this value, no change is made; i.e., a negative delay is made a zero delay. Control is now transferred to the point of the last delay in the element corresponding to this nearest event.

6. Control is returned to the operating system via

a. a system variable declaration statement or a FUNCTION RETURN -- error, simulation is stopped.

b. a delay statement. The time for the next event for this element, as specified by the delay statement, is placed in a list of next-event times. Notations are made of the location of the delay statement, the type of delay, etc. The next operation is step 5.

c. an output being sent. Checks are made to determine that this is a valid "send." A linear subscript is computed by taking the difference between the location originally declared and the location being sent. If the variable is not subscripted, the linear subscript will be zero. The new value for the variable is sent to all elements where it is an input using the linear subscript. The status of any waiting element receiving an input is appropriately changed. Control returns to the element at the point from which it came. The simulation operating program has the following entry points:

MAIN99	Original entry to begin simulation.
IN9999	Entry to declare input variables.
OUT999	Entry to declare output variables.
DELAY9	Simple delay entry.

WAIT99 Simple "wait for input" entry.

WVD999 "Wait or delay" entry.

WAD999 "Wait and delay" entry.

TIME99 Entry to return the value of TIME in the accumulator (for routines other than element specifications).

RESTRT Entry to restart the simulation at time zero.

TRACER Entry to start the diagnostic trace (see below).

STOPTR Entry to stop the diagnostic trace (see below).

PRSTAT Entry to print out the status of the simulation (see below).

Two features are added as an aid to debugging. First, there is an entry which prints the status of the simulation. This information includes the name, location, value, source, and destinations for every system variable as well as the name and status of every element. The status of an element is described by the location of the last delay, its type, and the time associated with it. This time is the value at which the delay is to end or, in the case of a simple wait, the value of TIME when it began. This status can be printed by a call to "PRSTAT" at any point in the simulation except during the initialization period. The other debugging feature is the trace routine. The execution of every delay statement and the sending of every output variable (except those which have no destination) is noted by: the name of the element in which the statement occurs, the location of this statement, the current value of TIME, and either the type of delay or the name, subscript, and value of the system variable being sent. A trace may be started at any time after initialization and then stopped, restarted, etc.

This simulation programming system was in almost continuous use by the author and some others for approximately eighteen months and is free of errors. Aside from simulating time-shared systems, use has been made of this system for the simulation of a storage system, sequential logic circuits, etc.

The following pages contain listings of the element specifications used in the simulations.

```
        ELEMENT MAIN
      R..MAIN CONTROL ELEMENT FOR CTSS SIMULATION.
      R  THIS ELEMENT SUPERVISES THE ENTRY OF PROGRAMS INTO ACTIVE
      R  STATUS, CALLS THE SCHEDULING ALGORITHM FOR CLOCKED ENTRIES,
      R  PROGRAM STATUS CHANGES,ETC., SUPERVISES DUMPING AND LOADING
      R  AND SIGNALS THE CONSOLES.  THE VARIABLES USED ARE--
      R   STATUS....STATUS OF PROGRAM (SAME AS IN SCHED.)
      R   LENGTH....LENGTH OF PROGRAMS
      R   INTCYC....NO. OF CYCLES TO END CURRENT INTERACTION.
      R   SWAPGO....SIGNAL TO STORAGE ALGORITHM TO BEGIN SWAP. VALUE
      R             IS THE SIZE OF THE PROGRAM TO BE SWAPPED. SIGN IS
      R             POSITIVE IF PRG ON DRUM, NEG. IF ON DISK.
      R   OLDSTA....STATUS OF PROGRAM BEING DUMPED
      R   NXTUSR....NEXT USER TO BE RUN (NEWUSR)
      R   NEWUSR....NEXT USER TO BE RUN
      R   CURUSR....CURRENT USER
      R   OLDUSR....USER BEING DUMPED
      R   GO........SIGNAL FROM CONSOLE INDICATING INPUT FINISH.
      R   RDY.......SIGNAL TO CONSOLE INDICATING INPUT WAIT (IF
      R             NEGATIVE) OR PROGRAM FINISH (IF POSITIVE).
      R   SWPDON....SIGNAL FROM STO. ALG. INDICATING SWAP FINISH
      R   CPUGO.....START SIGNAL FOR CPU.
      R   CPUBSY....BUSY SIGNAL FROM CPU
      R   CYCDON....SIGNAL FROM CPU INDICATING NO. OF INSTRUCTIONS
      R             EXECUTED.
      R
      R..I/O VARIABLES.
        OUTPUT VARIABLES CPUGO,SWAPGO,OLDSTA,NXTUSR,LSTUSR,RDY
        INPUT VARIABLES SWPDON,CPUBSY,GO,CYCDON
      R
      R..INITIALIZE SCHEDULING ALGORITHM.
        NC=NCONS.(0)
        EXECUTE SCHED.(0)
        EXECUTE SCHED.(6,0,32767)
        NXTIME=TIME
        SWPSW=0
        THROUGH L0, FOR J=1,1,J.G.NC
        RDY(J)=1
L0      LINMUL(1.(J))=1
      R
      R..MAIN TIMING LOOP.
LOOP    WAIT FOR INPUT OR UNTIL NXTIME
      R
      R SEE IF TIME FOR A CLOCKED ENTRY TO SCHED.
        WHENEVER TIME.GE.NXTIME
          EXECUTE SCHED.(1)
          NXTIME=TIME+CLKINT
          WHENEVER .NOT.INPUT .AND. SWAP.E.0, TRANSFER TO LOOP
        END OF CONDITIONAL
      R
```

```
          R..CHECK INPUT STATUS OF EACH CONSOLE.
            THROUGH L1, FOR J=1,1,J.G.NC
          R
            WHENEVER GO(J).NE.0
              WHENEVER STATUS(I.(J)).E.4
          R     USER IS COMING OUT OF INPUT WAIT.
                INTCYC(J)=CYCINT.(J)
                RDY(J)=0
                EXECUTE SCHED.(2,J,2)
              OR WHENEVER STATUS(I.(J)).LE.1
          R     USER JUST FINISHED COMMAND.
                INTCYC(J)=CYCINT.(J)
                EXECUTE SCHED.(6,J,PRGSIZ.(J))
                RDY(J)=0
                EXECUTE SCHED.(2,J,3)
              END OF CONDITIONAL
L1          END OF CONDITIONAL
          R
          R..CHECK STATUS OF SWAP, IF GOING ON.
            WHENEVER .NOT.SWPDON, TRANSFER TO L2
          R..SWAP FINISHED, SET SWITCHES, RESTART CPU.
            SWPSW=0
            SWAPGO=0
LW1         WAIT FOR INPUT
            WHENEVER SWPDON, TRANSFER TO LW1
            WHENEVER CURUSR.E.0, TRANSFER TO L2
            CPUGO=INTCYC(CURUSR)
LW2         WAIT FOR INPUT
            WHENEVER .NOT.CPUBSY, TRANSFER TO LW2
            TRANSFER TO LOOP
          R
          R..SEE IF SWAP IN PROGRESS.
L2          WHENEVER SWAP.E.0, TRANSFER TO L5
            WHENEVER SWAPGO.NE.0, TRANSFER TO LOOP
            SWPSW=1
            WHENEVER OLDUSR.E.0, TRANSFER TO L3
          R..STOP CPU SO SWAP CAN BEGIN.
            CPUGO=0
LW3         WAIT FOR INPUT
            WHENEVER CPUBSY, TRANSFER TO LW3
            INTCYC(OLDUSR)=INTCYC(OLDUSR)-CYCDON
            WHENEVER INTCYC(OLDUSR).LE.0, INTCYC(OLDUSR)=1
L3          WHENEVER STATUS(I.(NEWUSR)).E.3
              SWAPGO=-LENGTH(I.(NEWUSR))
            OTHERWISE
              SWAPGO=LENGTH(I.(NEWUSR))
            END OF CONDITIONAL
            NXTUSR=NEWUSR
            LSTUSR=OLDUSR
            OLDSTA=STATUS(I.(OLDUSR))
            TRANSFER TO LOOP
          R
          R..NOT SWAPPING, CHECK CPU.
L5          WHENEVER CURUSR.E.0 .OR. CPUBSY, TRANSFER TO LOOP
            CPUGO=0
```

```
          PUBSY=1B
          EQUIVALENCE (CPUBSY,PUBSY)
LW4       WAIT FOR INPUT
          WHENEVER CPUBSY, TRANSFER TO LW4
          INTCYC(CURUSR)=0
        R
        R..SEE IF PROGRAM GOES DEAD, DORMANT, OR INTO I/O WAIT.
          TEM=CONSTA.(0)
          WHENEVER TEM.E.3
            INTCYC(CURUSR)=CYCINT.(0)
            EXECUTE SCHED.(6,CURUSR,PRGSIZ.(0))
          OTHERWISE
            RDY(CURUSR)=TIME
          END OF CONDITIONAL
          EXECUTE SCHED.(2,CURUSR,TEM)
        R..'SWAP' MUST BE SET, BEGIN SWAPPING.
          WHENEVER SWAP.E.0, FUNCTION RETURN
          SWPSW=1
          TRANSFER TO L3
        R
        R
          BOOLEAN CPUBSY,KILL,PUBSY,SWPDON
          DIMENSION GO(50),RDY(50),INTCYC(50)
          NORMAL MODE IS INTEGER
          VECTOR VALUES CLKINT=200000
          INSERT FILE COMN1A
          END OF SPECIFICATION
```

```
          ELEMENT STOALG
         R..THIS ROUTINE TAKES CARE OF SWAPPING PROGRAMS.
         R..THE VARIABLES USED ARE--
         R   CORUSR....LIST OF USER NOS. FOR CORE USERS.
         R   COREUB....UPPER BOUND LIST FOR CORE USERS.
         R   CORELB....LOWER BOUND LIST FOR CORE USERS.
         R   NINCOR....NUMBER OF USERS IN CORE.
         R   MXCORE....MAXIMUM NUMBER OF USERS ALLOWED IN CORE.
         R   SWAPGO....GO SIGNAL TO BEGIN SWAP.  VALUE IS
         R           +SIZE OF NXTUSR   WHEN NXTUSR ON DRUM.
         R              0              FOR NO ACTION.
         R           -SIZE OF NXTUSR   WHEN NXTUSR ON DISK.
         R   NXTUSR....NO. OF NEXT USER (NEWUSR).
         R   XMIT......GO SIGNAL TO BULK STORAGE TO XMIT A PROGRAM
         R               BETWEEN CORE AND DRUM OR DISK, DEPENDING
         R               ON SIGN OF 'XMIT'.
         R   XMTRDY....READY SIGNAL FROM BULK STORAGE.
         R   SWPDON....READY SIGNAL TO MAIN CONTROL.
         R   CONDS.....NO. OF WORDS FOR MACHINE CONDITIONS AND DISK
         R               STATUS OF A USER. ALWAYS DUMPED AND LOADED.
         R
         R..I/O VARIABLES.
          INPUT VARIABLES SWAPGO,OLDSTA,NXTUSR,XMTRDY,LSTUSR
          OUTPUT VARIABLES XMIT,SWPDON
         R
          VECTOR VALUES CONDS=714
         R..INITIALIZATION.
          XMIT=0
          NINCOR=0
          SWPDON=0B
         R
         R..WAIT HERE FOR SWAP GO SIGNAL FROM MAIN CONTROL.
WAIT      WAIT FOR INPUT
          WHENEVER SWAPGO.E.0, TRANSFER TO WAIT
         R
         R..DUMP OLD USER, TELL SCHED. ALG.
          EXECUTE SCHED.(3)
         R..IF LAST USER NOT 0, DUMP MACHINE CONDITIONS TO DRUM.
          XMIT=CONDS
LW00      WAIT FOR INPUT
          WHENEVER .NOT.XMTRDY, TRANSFER TO LW00
          XMIT=0
LW01      WAIT FOR INPUT
          WHENEVER XMTRDY, TRANSFER TO LW01
          WHENEVER OLDSTA.NE.2, TRANSFER TO BIGDMP
          WHENEVER NINCOR.G.NBUFF, TRANSFER TO DMPONE
         R
         R..MAKE ENOUGH ROOM IN CORE FOR NEW USER.
FREEUP    WHENEVER NINCOR.E.0
            XMIT=SWAPGO
            TRANSFER TO DMPDON
          OR WHENEVER NXTUSR.E.CORUSR(1)
            XMIT=CORELB(1)
            TRANSFER TO DMPDON
```

```
            OR WHENEVER .ABS.SWAPGO.LE.CORELB(1)
              XMIT=SWAPGO
              TRANSFER TO DMPDON
            OR WHENEVER .ABS.SWAPGO.L.COREUB(1)
              XMIT=.ABS.SWAPGO-CORELB(1)
LW1           WAIT FOR INPUT
              WHENEVER .NOT.XMTRDY, TRANSFER TO LW1
              XMIT=0
LW2           WAIT FOR INPUT
              WHENEVER XMTRDY, TRANSFER TO LW2
              XMIT=SWAPGO
              CORELB(1)=.ABS.SWAPGO
              TRANSFER TO DMPDON
            OTHERWISE
              XMIT=COREUB(1)-CORELB(1)
LW3           WAIT FOR INPUT
              WHENEVER .NOT.XMTRDY, TRANSFER TO LW3
              XMIT=0
LW4           WAIT FOR INPUT
              WHENEVER XMTRDY, TRANSFER TO LW4
              THROUGH L2, FOR J=1,1,J.GE.NINCOR
              CORUSR(J)=CORUSR(J+1)
              COREUB(J)=COREUB(J+1)
L2            CORELB(J)=CORELB(J+1)
              NINCOR=NINCOR-1
            END OF CONDITIONAL
            TRANSFER TO FREEUP
          R
          R..ALL DUMPING DONE, LOAD HAS JUST STARTED. INFORM
          R..SCHEDULING ALGORITHM AND WAIT FOR COMPLETION.
DMPDON      EXECUTE SCHED.(4)
LW5         WAIT FOR INPUT
            WHENEVER .NOT.XMTRDY, TRANSFER TO LW5
            XMIT=0
LW6         WAIT FOR INPUT
            WHENEVER XMTRDY, TRANSFER TO LW6
          R
          R..PROG. LOADED, LOAD MACHINE CONDITIONS, ETC. FROM DRUM,
          R.. ALG., GIVE FINISH SIGNAL TO MAIN CONTROL ELEMENT.
            XMIT=CONDS
LW100       WAIT FOR INPUT
            WHENEVER .NOT.XMTRDY, TRANSFER TO LW100
            XMIT=0
LW101       WAIT FOR INPUT
            WHENEVER XMTRDY, TRANSFER TO LW101
            EXECUTE SCHED.(5)
            SWPDON=1B
            WHENEVER NINCOR.G.0 .AND. NXTUSR.E.CORUSR(1)
              CORELB(1)=0
              TRANSFER TO LW7
            END OF CONDITIONAL
            CORELB(0)=0
            COREUB(0)=.ABS.SWAPGO
            CORUSR(0)=NXTUSR
            NINCOR=NINCOR+1
```

```
          THROUGH L3, FOR J=NINCOR,-1,J.LE.0
          CORELB(J)=CORELB(J-1)
          COREUB(J)=COREUB(J-1)
L3        CORUSR(J)=CORUSR(J-1)
LW7       WAIT FOR INPUT
          WHENEVER SWAPGO.NE.0, TRANSFER TO LW7
          SWPDON=0B
          TRANSFER TO WAIT
        R
        R
        R..NO MORE WRITE BUFFERS, COMPLETE DUMP OF USER
        R..CLOSEST TO BEING COMPLETELY DUMPED.
DMPONE    JJ=2
          SIZE=COREUB(2)-CORELB(2)
          THROUGH L4, FOR J=3,1,J.G.NINCOR
          WHENEVER SIZE.G.COREUB(J)-CORELB(J)
            JJ=J
            SIZE=COREUB(J)-CORELB(J)
L4        END OF CONDITIONAL
          XMIT=SIZE
LW8       WAIT FOR INPUT
          WHENEVER .NOT.XMTRDY, TRANSFER TO LW8
          XMIT=0
LW9       WAIT FOR INPUT
          WHENEVER XMTRDY, TRANSFER TO LW9
          THROUGH L5, FOR J=JJ,1,J.GE.NINCOR
          CORELB(J)=CORELB(J+1)
          COREUB(J)=COREUB(J+1)
L5        CORUSR(J)=CORUSR(J+1)
          NINCOR=NINCOR-1
          TRANSFER TO FREEUP
        R
        R
        R..OLD USER NOT IN WORKING STATUS, DUMP ENTIRELY.
BIGDMP    WHENEVER OLDSTATE.0, TRANSFER TO DELETE
          XMIT=COREUB(1)
LW10      WAIT FOR INPUT
          WHENEVER .NOT.XMTRDY, TRANSFER TO LW10
          XMIT=0
LW11      WAIT FOR INPUT
          WHENEVER XMTRDY, TRANSFER TO LW11
DELETE    THROUGH L6, FOR J=1,1,J.GE.NINCOR
          CORELB(J)=CORELB(J+1)
          COREUB(J)=COREUB(J+1)
L6        CORUSR(J)=CORUSR(J+1)
          NINCOR=NINCOR-1
          TRANSFER TO FREEUP
        R
        R
          NORMAL MODE IS INTEGER
          BOOLEAN SWPRDY,XMTRDY,SWPDON
          DIMENSION CORELB(5),COREUB(5),CORUSR(5)
          VECTOR VALUES NBUFF=4
          END OF SPECIFICATION
```

```
          ELEMENT BULK
         R..THIS ELEMENT SIMULATES BOTH THE DISK AND DRUM.
         R
         R..I/O VARIABLES.
           INPUT VARIABLES XMIT
           OUTPUT VARIABLES XMTRDY,DSKUSE,DRMUSE
         R
           DSKUSE=0
           DRMUSE=0
LOOP       XMTRDY=0B
WAIT1      WAIT FOR INPUT
           WHENEVER XMIT.E.0, TRANSFER TO WAIT1
           WHENEVER XMIT.G.0
             TEM=DRMDEL.(XMIT)
             DRMUSE=DRMUSE+TEM
           OTHERWISE
             TEM=DSKDEL.(.ABS.XMIT)
             DSKUSE=DSKUSE+TEM
           END OF CONDITIONAL
           DELAY OF TEM
           XMTRDY=1B
WAIT2      WAIT FOR INPUT
           WHENEVER XMIT.NE.0, TRANSFER TO WAIT2
           TRANSFER TO LOOP
         R
           NORMAL MODE IS INTEGER
           BOOLEAN XMTRDY
           END OF SPECIFICATION
```

```
          ELEMENT CPU
         R..THIS ELEMENT SIMULATES A CPU WITH NO SIGNIFICANT
         R..CHANNEL OPERATION GOING ON WHILE IT IS OPERATING.
         R
         R..I/O VARIABLES.
          INPUT VARIABLES CPUGO
          OUTPUT VARIABLES CPUBSY,CYCDON,CPUUSE
         R
          CPUUSE=0
LOOP      CPUBSY=0B
WAIT1     WAIT FOR INPUT
          WHENEVER CPUGO.E.0, TRANSFER TO WAIT1
          CPUBSY=1B
          STARTT=TIME
          TEM=FTOI.(ITOF.(CPUGO)/INRATE.(0))
          FINTIM=TIME + TEM
          TEM=CPUGO
WAIT2     WAIT FOR INPUT OR UNTIL FINTIM
          WHENEVER CPUGO.E.0
            CYCDON=(TIME-STARTT)*INRATE.(0)
            WHENEVER CYCDON.GE.TEM, CYCDON=TEM-1
            CPUBSY=0B
            TRANSFER TO STOP
          OR WHENEVER TIME.GE.FINTIM
            CYCDON=TEM
            CPUBSY=0B
WAIT3       WAIT FOR INPUT
            WHENEVER CPUGO.NE.0, TRANSFER TO WAIT3
            TRANSFER TO STOP
          END OF CONDITIONAL
          TRANSFER TO WAIT2
         R
STOP      CPUUSE=CPUUSE+TIME-STARTT
          TRANSFER TO LOOP
         R
          NORMAL MODE IS INTEGER
          FLOATING POINT INRATE.,ITOF.
          BOOLEAN CPUBSY
          END OF SPECIFICATION
```

```
          ELEMENT CONS
         R..THIS ELEMENT SIMULATES ALL OF THE CONSOLES.
         R..THE VARIABLES ARE--
         R   GO.........SIGNAL TO MAIN CONTROL INDICATING CONSOLE'S
         R               COMPLETION OF INPUT.
         R   RDY........SIGNAL FROM MAIN CONTROL INDICATING EITHER
         R               USER'S PROGRAM IS IN INPUT WAIT (IF NEG.) OR
         R               FINISHED (IF POSITIVE).
         R   NXTACT.....LIST OF TIMES FOR CONSOLES TO FINISH INPUT,
         R               IF ZERO, CONSOLE IS WAITING FOR PROGRAM.
         R   NXTIME.....TIME OF NEXT EVENT (CONSOLE FINISHING INPUT).
         R   NCONS......NUMBER OF CONSOLES.
         R
         R..I/O VARIABLES.
           INPUT VARIABLES RDY
           OUTPUT VARIABLES GO
           NC=NCONS.(0)
         R
         R..INITIALIZATION.
           DELAY OF 1
           THROUGH L0, FOR I=1,1,I.G.NC
           GO(I)=0
           NXTACT(I)=0
L0         CONTINUE
         R
         R..MAIN LOOP.
LOOP       NXTIME=0
           THROUGH L1, FOR I=1,1,I.G.NC
           WHENEVER NXTACT(I).NE.0
             WHENEVER NXTACT(I).G.TIME, TRANSFER TO JOINT
             GO(I)=TIME
LW1          WAIT FOR INPUT
           WHENEVER GO(I).NE.TIME
             PRINT FORMAT ERR,TIME
             EXIT.
             V'S ERR=$10HCONS ERR. K12*$
           END OF CONDITIONAL
             WHENEVER RDY(I).NE.0, TRANSFER TO LW1
             NXTACT(I)=0
             GO(I)=0
             TRANSFER TO L1
           OTHERWISE
             WHENEVER RDY(I).E.0, TRANSFER TO L1
             NXTACT(I)=TIME+INTDEL.(I)
           END OF CONDITIONAL
JOINT      WHENEVER NXTIME.G.NXTACT(I).OR.NXTIME.E.0, NXTIME=NXTACT(I)
L1         CONTINUE
           WHENEVER NXTIME.E.0
             WAIT FOR INPUT
           OTHERWISE
             WAIT FOR INPUT OR UNTIL NXTIME
           END OF CONDITIONAL
           TRANSFER TO LOOP
         R
```

```
R
  NORMAL MODE IS INTEGER
  DIMENSION RDY(50),GO(50),NXTACT(50)
  END OF SPECIFICATION
```

```
          R.. ELEMENT TO CONTROL CPU AND SWAPPING FOR OVERLAPPED
          R.. OPERATION OF DISK, DRUM, AND PROCESSOR.
          R
            ELEMENT CPUCTL
            INPUT VARIABLES STARTU,GO,CPUBSY,CYCDON
            OUTPUT VARIABLES NEEDU,RDY,CPUGO,NEWUSR
            OUTPUT VARIABLES NQ,Q
            NORMAL MODE IS INTEGER
          R
            CPUGO=0
            NUSERS=NCONS.(0)
            THROUGH L0, FOR J=1,1,J.G.NUSERS
            RDY(J)=1
L0          CONTINUE
            NQ=0
            NEWUSR=0
            NEEDU=1B
          R
          R.. HERE WHEN IDLE, WAIT FOR NEW USER TO COME ALONG.
IDLE        WAIT FOR INPUT
            CHECKU.
            WHENEVER NEWUSR.E.0, TRANSFER TO IDLE
          R
          R.. WAIT FOR LOAD OF NEW USER.
LOADWT      WAIT FOR INPUT
            CHECKU.
            WHENEVER STARTU.NE.NEWUSR, TRANSFER TO LOADWT
          R
          R.. LOAD DONE, INTERLOCK.
LOOP        NEEDU=0B
            CPUGO=INTCYC(NEWUSR)
            SWPSW=0
            EXECUTE MONSC1.(5,NEWUSR,0)
            STATUS(1.(NEWUSR))=2
            GETNXT.
LW1         WAIT FOR INPUT
            WHENEVER STARTU.NE.0, TRANSFER TO LW1
            WHENEVER .NOT.CPUBSY, TRANSFER TO LW1
            CHECKU.
          R
          R.. HERE WHEN USER STARTED.
            FINTIM=TIME + BURSTT.(0)
HOLD        WAIT FOR INPUT OR UNTIL FINTIM
            WHENEVER .NOT.CPUBSY
          R.. CPU STOPPED. FINISH OFF USER.
PUSTOP      CPUGO=0
              PUBSY=1B
              EQUIVALENCE (CPUBSY,PUBSY)
LW2           WAIT FOR INPUT
              WHENEVER CPUBSY, TRANSFER TO LW2
              TEMSTA=CONSTA.(0)
              WHENEVER TEMSTA.NE.3
                RDY(CURUSR)=TIME
                DELQ.(CURUSR)
              END OF CONDITIONAL
```

```
                EXECUTE MONSC1.(2,CURUSR,TEMSTA)
                SWPSW=1
                EXECUTE MONSC1.(3,CURUSR,0)
                STATUS(I.(CURUSR))=TEMSTA
                NEEDU=1B
                CHECKU.
                WHENEVER NEWUSR.E.0, TRANSFER TO IDLE
LW3             WHENEVER STARTU.E.NEWUSR, TRANSFER TO LOOP
                CHECKU.
                WAIT FOR INPUT
                TRANSFER TO LW3
              OR WHENEVER TIME.GE.FINTIM
                WHENEVER NEWUSR.NE.0, NEEDU=1B
LW4             WHENEVER .NOT.CPUBSY, TRANSFER TO PUSTOP
                CHECKU.
                WHENEVER NEWUSR.NE.0 .AND. .NOT.NEEDU
                  NEEDU=1B
                END OF CONDITIONAL
                WHENEVER STARTU.NE.0, TRANSFER TO STOPPU
                WAIT FOR INPUT
                TRANSFER TO LW4
STOPPU          CPUGO=0
                SWPSW=1
                EXECUTE MONSC1.(3,OLDUSR,0)
LW5             WAIT FOR INPUT
                WHENEVER CPUBSY, TRANSFER TO LW5
                INTCYC(OLDUSR)=INTCYC(OLDUSR)-CYCDON
                WHENEVER INTCYC(OLDUSR).LE.0, INTCYC(OLDUSR)=1
                NEEDU=1B
LW6             CHECKU.
                WHENEVER NEWUSR.E.STARTU, TRANSFER TO LOOP
                WAIT FOR INPUT
                TRANSFER TO LW6
              END OF CONDITIONAL
              CHECKU.
              TRANSFER TO HOLD
            R
              BOOLEAN CPUBSY,PUBSY,NEEDU,CHANGE
              DIMENSION GO(50),RDY(50),INTCYC(50),Q(50)
            R
              INTERNAL FUNCTION
               NTRY TO GETNXT.
              CHECKU.
              WHENEVER NQ.E.0 .OR. (NQ.E.1 .AND. CURUSR.NE.0)
                NEWUSR=0
                FUNCTION RETURN
              OR WHENEVER CURUSR.E.0
                NEWUSR=Q(1)
                FUNCTION RETURN
              END OF CONDITIONAL
              THROUGH GETN1, FOR J=1,1,J.G.NQ
GETN1         WHENEVER Q(J).E.CURUSR, TRANSFER TO GETN2
              PRINT COMMENT $CURRENT USER NOT IN QUEUE.$
              EXIT.
GETN2         J=J+1
```

```
            WHENEVER J.G.NQ, J=1
            NEWUSR=Q(J)
            FUNCTION RETURN
            END OF FUNCTION
          R
            INTERNAL FUNCTION
            ENTRY TO CHECKU.
            CHANGE=0B
            THROUGH CHECK1, FOR J=1,1,J.G.NUSERS
            WHENEVER RDY(J).E.0 .OR. GO(J).E.0, TRANSFER TO CHECK1
            WHENEVER STATUS(I.(J)).LE.1
              EXECUTE MONSC1.(2,J,3)
              STATUS(I.(J))=3
              LENGTH(I.(J))=PRGSIZ.(0)
              MONSC1.(6,J,LENGTH(I.(J)))
            OTHERWISE
              EXECUTE MONSC1.(2,J,2)
              STATUS(I.(J))=2
            END OF CONDITIONAL
            INTCYC(J)=CYCINT.(0)
            RDY(J)=0
            WHENEVER NEWUSR.E.0
              NEWUSR=J
            END OF CONDITIONAL
            NQ=NQ+1
            Q(NQ)=J
            CHANGE=1B
CHECK1      CONTINUE
            WHENEVER CHANGE, SORTQ.
            FUNCTION RETURN
            END OF FUNCTION
          R
            INTERNAL FUNCTION
            ENTRY TO SORTQ.
            CHANGE=0B
            THROUGH SORT2, FOR J=1,1,J.GE.NQ
            THROUGH SORT1, FOR JJ=J+1,1,JJ.G.NQ
            WHENEVER LENGTH(I.(Q(J))).G.LENGTH(I.(Q(JJ)))
          1 .AND. CHANGE, TRANSFER TO SORT0
            WHENEVER LENGTH(I.(Q(J))).L.LENGTH(I.(Q(JJ)))
          1 .AND. .NOT.CHANGE, TRANSFER TO SORT0
            TRANSFER TO SORT1
SORT0       TEM=Q(J)
            Q(J)=Q(JJ)
            Q(JJ)=TEM
SORT1       CONTINUE
SORT2       CHANGE=.NOT.CHANGE
            FUNCTION RETURN
            END OF FUNCTION
          R
            INTERNAL FUNCTION (A)
            ENTRY TO DELQ.
            THROUGH DEL1, FOR J=1,1,J.G.NQ
DEL1        WHENEVER Q(J).E.A, TRANSFER TO DEL2
            PRINT COMMENT $USER TO BE DELETED NOT IN QUEUE.$
```

```
          PRINT FORMAT XX,A,CURUSR,OLDUSR,NEWUSR,NQ,Q(1)...Q(25)
          V'S XX=$4HARG=I2,3H CUI3,3H OUI3,3H NUI3,3H NQI3,
         1 /25I3*$
           EXIT.
DEL2       THROUGH DEL3, FOR J=J+1,1,J.G.NQ
DEL3       Q(J-1)=Q(J)
           NQ=NQ-1
           SORTQ.
           FUNCTION RETURN
           END OF FUNCTION
         R
           INSERT FILE COMN1A
           END OF SPECIFICATION
```

Note that MONSC1 is a data taking routine called with the same arguments that the CTSS Scheduling Algorithm used.

```
          R.. SWAP CONTROL FOR OVERLAPPED SWAPPING.
            ELEMENT BLKCTL
          R
          R.. I/O VARIABLES.
            OUTPUT VARIABLES OLDUSR,CURUSR
            INPUT VARIABLES NEEDU,NEWUSR,XMTRDY
            OUTPUT VARIABLES STARTU,XMIT
            NORMAL MODE IS INTEGER
            BOOLEAN NEEDU,XMTRDY
          R
          R..INITIALIZATION.
            STARTU=0
            CURUSR=0
            OLDUSR=0
            XMIT=0
            LSIZE=0
            DSIZE=0
            LENGTH(0)=77776K
            STATUS(0)=2
            VECTOR VALUES STASIZ=714
          R
          R.. HERE WHEN CURRENT USER RUNNING AND NO NEW USER.
IDLE        WAIT FOR INPUT
          R
LOOP        WHENEVER NEWUSR.NE.0, TRANSFER TO LOAD
            WHENEVER .NOT.NEEDU .OR. CURUSR.E.0, TRANSFER TO IDLE
          R
          R.. DUMP CURRENT USER, NO NEW USER YET.
            OLDUSR=CURUSR
            CURUSR=0
            WHENEVER STATUS(I.(OLDUSR)).E.0
              DSIZE=0
            OTHERWISE
              DSIZE=LENGTH(I.(OLDUSR))
            END OF CONDITIONAL
            BULKGO.(DSIZE+STASIZ)
            DSIZE=0
            WHENEVER NEWUSR.NE.0
              CURSIZ=0
              TRANSFER TO LOAD1
            END OF CONDITIONAL
            BULKGO.(STASIZ+LENGTH(0))
            CURUSR=0
            TRANSFER TO LOOP
          R
          R.. LOAD NEW USER WHILE OLD USER RUNNING.
LOAD        CURSIZ=LENGTH(I.(CURUSR))
LOAD1       LSIZE=LENGTH(I.(NEWUSR))
            WHENEVER LSIZE.G.(77777K-CURSIZ), TRANSFER TO HARDFT
          R
          R.. NEW USER FITS INTO CORE WITH CURRENT USER.
            WHENEVER STATUS(I.(NEWUSR)).E.3
              BULKGO.(STASIZ)
              BULKGO.(-LSIZE)
            OTHERWISE
```

```
              BULKGO.(STASIZ+LSIZE)
            END OF CONDITIONAL
            LSIZE=0
LW1         WHENEVER NEEDU, TRANSFER TO L1
            WAIT FOR INPUT
            TRANSFER TO LW1
L1          OLDUSR=CURUSR
            STARTU=NEWUSR
            CURUSR=NEWUSR
LW2         WAIT FOR INPUT
            WHENEVER NEEDU, TRANSFER TO LW2
            STARTU=0
            WHENEVER STATUS(I.(OLDUSR)).E.0
              DSIZE=0
            OTHERWISE
              DSIZE=CURSIZ
            END OF CONDITIONAL
            BULKGO.(DSIZE+STASIZ)
            DSIZE=0
            TRANSFER TO LOOP
          R
          R.. NEW USER WILL NOT FIT INTO CORE WITH CURRENT USER.
HARDFT      LSIZE=CURSIZ+LSIZE-77777K
            WHENEVER STATUS(I.(NEWUSR)).E.3
              BULKGO.(STASIZ)
              BULKGO.(CURSIZ-77777K)
            OTHERWISE
              BULKGO.(STASIZ+77777K-CURSIZ)
            END OF CONDITIONAL
LW3         WHENEVER NEEDU, TRANSFER TO L2
            WAIT FOR INPUT
            TRANSFER TO LW3
L2          STARTU=-NEWUSR
            OLDUSR=CURUSR
            WHENEVER STATUS(I.(OLDUSR)).E.0
              DSIZE=0
              BULKGO.(STASIZ)
            OTHERWISE
              BULKGO.(STASIZ+LSIZE)
              DSIZE=CURSIZ-LSIZE
            END OF CONDITIONAL
            WHENEVER STATUS(I.(NEWUSR)).E.3
              NOSEEK.(1)
              BULKGO.(-LSIZE)
              NOSEEK.(0)
            OTHERWISE
              BULKGO.(LSIZE)
            END OF CONDITIONAL
            LSIZE=0
            STARTU=NEWUSR
            CURUSR=NEWUSR
LW4         WAIT FOR INPUT
            WHENEVER NEEDU, TRANSFER TO LW4
            STARTU=0
            WHENEVER DSIZE.G.0, BULKGO.(DSIZE)
```

```
          DSIZE=0
          TRANSFER TO LOOP
        R
        R.. INTERNAL FUNCTION TO WORK DISK AND DRUM.
          INTERNAL FUNCTION (ARG)
          ENTRY TO BULKGO.
          XMIT=ARG
B1        WAIT FOR INPUT
          WHENEVER NEWUSR.NE.0 .AND. CURUSR.E.0
            CURUSR=NEWUSR
          END OF CONDITIONAL
          WHENEVER .NOT.XMTRDY, TRANSFER TO B1
          XMIT=0
B2        WAIT FOR INPUT
          WHENEVER XMTRDY, TRANSFER TO B2
          FUNCTION RETURN
          END OF FUNCTION
        R
          INSERT FILE COMN1A
          END OF SPECIFICATION
```

```
          R.. THIS ELEMENT SIMULATES BOTH THE DRUM AND THE DISK.
          R.. IT PROVIDES A FACTOR, 'IOFACT', TO THE CPU ELEMENT GIVING
          R.. THE AVERAGE FACTOR OF DEGRADATION BETWEEN 0 AND 1.
          R
            ELEMENT BULK
            INPUT VARIABLES XMIT,CPUBSY
            OUTPUT VARIABLES XMTRDY,IOFACT,DKOUSE,DSKUSE,DMOUSE,DRMUSE
          R
            DSKUSE=0
            DRMUSE=0
            DKOUSE=0
            DMOUSE=0
LOOP        XMTRDY=0B
            IOFACT=1.
WAIT1       WAIT FOR L PUT
            WHENEVER XMIT.E.0, TRANSFER TO WAIT1
            WHENEVER XMIT.G.0
              TEM=DRMDEL.(XMIT)
              IOFACT=.7
              DRMUSE=DRMUSE+TEM
              WHENEVER CPUBSY, DMOUSE=DMOUSE+TEM
            OTHERWISE
              TEM=DSKDEL.(.ABS.XMIT)
              DSKUSE=DSKUSE+TEM
              IOFACT=.988
              WHENEVER CPUBSY, DKOUSE=DKOUSE+TEM
            END OF CONDITIONAL
            DELAY OF TEM
            XMTRDY=1B
WAIT2       WAIT FOR INPUT
            WHENEVER XMIT.NE.0, TRANSFER TO WAIT2
            TRANSFER TO LOOP
          R
            FLOATING POINT IOFACT
            NORMAL MODE IS INTEGER
            BOOLEAN XMTRDY,CPUBSY
            END OF SPECIFICATION
```

```
          R.. CPU FOR OVERLAPPED CPU AND BULK MEMORY OPERATION.
          R.. SAME AS OLD CPU ELEMENT, EXCEPT THAT 'IOFACT' GIVES
          R.. FACTOR OF SLOWDOWN BECAUSE OF OVERLAP.
          R
            ELEMENT CPU
            INPUT VARIABLES CPUGO,IOFACT
            OUTPUT VARIABLES CPUBSY,CYCDON, PUUSE,NPUUSE
            GPUUSE=0
            NPUUSE=0
          R
LOOP        CPUBSY=0B
WAIT1       WAIT FOR INPUT
            WHENEVER CPUGO.E.0, TRANSFER TO WAIT1
            CPUBSY=1B
            CYCDON=0
            PUGO=CPUGO
          R
AGAIN       FINTIM=TIME+FTOI.(ITOF.(CPUGO-CYCDON)/(INRATE.(0)*IOFACT))
            FACT=IOFACT
            STARTT=TIME
WAIT2       WAIT FOR INPUT OR UNTIL FINTIM
            WHENEVER CPUGO.E.0 .OR. IOFACT.NE.FACT
              CYCDON=CYCDON+FTOI.(ITOF.(TIME-STARTT)*INRATE.(0)*FACT)
              GPUUSE=GPUUSE+TIME-STARTT
              NPUUSE=NPUUSE+FTOI.(FACT*ITOF.(TIME-STARTT))
              WHENEVER CYCDON.GE.PUGO
                CYCDON=PUGO
                STARTT=TIME
                WHENEVER CPUGO.E.0, TRANSFER TO STOP
              END OF CONDITIONAL
              WHENEVER CPUGO.NE.0, TRANSFER TO AGAIN
              CPUBSY=0B
              TRANSFER TO LOOP
            OR WHENEVER TIME.GE.FINTIM
STOP          CYCDON=PUGO
              GPUBSY=GPUBSY+TIME+STARTT
              NPUBSY=NPUBSY+FTOI.(FACT*ITOF.(TIME-STARTT))
              CPUBSY=0B
WAIT3         WAIT FOR INPUT
              WHENEVER CPUGO.E.0, TRANSFER TO WAIT3
              TRANSFER TO LOOP
            END OF CONDITIONAL
            TRANSFER TO WAIT2
          R
            NORMAL MODE IS INTEGER
            FLOATING POINT ITOF.,INRATE.,IOFACT,FACT
            BOOLEAN CPUBSY
            END OF SPECIFICATION
```

This program provides random numbers of various types, etc.

```
 EXTERNAL FUNCTION (ARG)
 INTEGER ARG
 ENTRY TO BURST.
R ENTRY TO GIVE BURST TIME IN 1/60THS.
 FUNCTION RETURN 120
R
R
 ENTRY TO BURSTT.
R GIVES QUANTUM TIME.
 FUNCTION RETURN 2 000 000
 ENTRY TO NCONS.
R GIVES NUMBER OF CONSOLES...
 FUNCTION RETURN 35
 ENTRY TO PRGSIZ.
R RETURNS PROGRAM SIZES ACCORDING TO FOLLOWING DISTRIBUTION.
 VECTOR VALUES SIZDIS = .155,
1 .045, .27, .03, .05, .03, .02, .0725, .015, .01, .035,
2 .025, .015, .005, .0025, .01, .01, .01, .005, .0025, .005,
3 .005, .01, .005, .00, .0025, .00, .005, .00, .005, .00,
4 .005, .005, .005, .005, .00, .00, .00, .0025, .00, .005,
5 .005, .0075, .00, .00, .00, .00, .00, .00, .005, .005,
6 .005, .005, .005, .00, .00, .00, .00, .00, .00, .015,
7 .0025, .00, .005, .0125, .04501
R LAST ENTRY IS .00001 TOO HIGH TO INSURE WORKING.
 X=RANNO.(0)
 TOT=0.
 THROUGH L1, FOR I=0.,1.,TOT.GE.X
L1  TOT=TOT+SIZDIS(I)
 INT=(I+TOT-X)*500.
 WHENEVER INT.G.32766.,INT=32766.
 TRANSFER TO RETURN
R
 ENTRY TO CONSTA.
R RETURNS THE NEW STATE FOR A PROGRAM. VALUE IS
R EITHER 0,1,3, OR 4.
 TOT=RANNO.(0)
 WHENEVER TOT.LE..631, FUNCTION RETURN 4
 WHENEVER TOT.LE..812, FUNCTION RETURN 0
 WHENEVER TOT.LE..88, FUNCTION RETURN 1
 FUNCTION RETURN 3
R
 ENTRY TO INRATE.
R ARBITRARY RATE OF INSTRUCTION EXECUTION.
 FUNCTION RETURN .2
R NO. OF INSTRUCTIONS EXECUTED FOR AN INTERACTION.
R.. VALUES FOR CPU TIME DISTRIBUTION.
 VECTOR VALUES CPU=
1 .5072, .0449, .0609, .0393, .0363,
2 .0304, .0248, .0250, .0217, .0186,
3 .0148, .0119, .0100, .0090, .0085,
4 .0076, .0070, .0065, .0058, .0053, .0052
 ENTRY TO INTCYC.
 ENTRY TO CYCINT.
 X=RANNO.(0)
```

```
            WHENEVER X.G..99
              CPUT=5. + 24.7*(-LOG.(1.-RANNO.(0)))
            OR WHENEVER X.G..97
              CPUT=4. + RANNO.(0)
            OR WHENEVER X.G..94
              CPUT=3. + RANNO.(0)
            OR WHENEVER X.G..895
              CPUT=2. + RANNO.(0)
            OTHERWISE
              TOT=0.
              THROUGH LL, FOR I=0.,1.,0B
              TOT= TOT + CPU(I)
LL            WHENEVER TOT.G.X, TRANSFER TO LLL
LLL           CPUT=I/10. + .1*RANNO.(0)
            END OF CONDITIONAL
            INT=CPUT/5E-6
            TRANSFER TO RETURN
           R
            ENTRY TO INTDEL.
           R DELAY BEFORE AN INTERACTION. 12 PERCENT OF THE TIME
           R AN INTERACTION HAS A ZERO CONSOLE PART (BECAUSE
           R OF A PROGRAM GENERATED COMM.). THIS IS TAKEN CARE
           R OF BY THE 'RING' ELEMENT. THE FOLLOWING DISTRIBUTION
           R YIELDS A TIME BETWEEN 0 AND 2
           R SECONDS WITH A PROBABILITY OF .0784 AND A TIME
           R DISTRIBUTED WITH A MAX OF .05104 AT 6. (8.)
           R AND A MEAN OF 41.34, ADDED TO 2.
            WHENEVER RANNO.(0).LE..0784
              INT=2E6*RANNO.(0)
            OTHERWISE
              INT=1E6*(2.+SPRAN.(.167,.8244,358.26))
            END OF CONDITIONAL
            TRANSFER TO RETURN
           R
            ENTRY TO DRMDEL.
           R DELAY FOR 'ARG' WORDS FROM DRUM.
            INT=ARG*8.4 + RANNO.(0)*17200.
            TRANSFER TO RETURN
           R
            ENTRY TO DSKDEL.
           R DELAY FOR 'ARG' WORDS FROM DISK.
            WHENEVER RANNO.(0).G..2
              BASIS=20.*466.
            OTHERWISE
              BASIS=1.4*466.
            END OF CONDITIONAL
            TOT=0.
            THROUGH L2, FOR INT=ARG+BASIS*RANNO.(0),-BASIS,INT.L.F*BASIS
            X=RANNO.(0)
            WHENEVER X.L..033
              X=50000.
            OR WHENEVER X.L..167
              X=120000.
            OTHERWISE
              X=180000.
            END OF CONDITIONAL
```

```
          TOT=TOT+X+RANNO.(0)*34000.
L2        CONTINUE
          INT=TOT+66.5927*ARG
RETURN    WHENEVER INT.LE.1., INT=1.
          FUNCTION RETURN FTOI.(INT)
          INTEGER FTOI.,F
         R
          ENTRY TO NOSEEK.
         R ENTRY TO SAY IF NO INITIAL DISK SEEK REQUIRED.
          F=ARG
          FUNCTION RETURN
         R
          INTERNAL FUNCTION
         R THIS ROUTINE GENERATES RANDOM NUMBERS OF TWO KINDS.
         R 'EXPRAN.' RETURNS A NUMBER EXPONENTIALLY DISTRIBUTED
         R FROM 0 (E.P.(-X)) WITH A MEAN OF 1.0
         R 'DMPRAN.' RETURNS A NUMBER WHICH IS DISTRIBUTED
         R ACCORDING TO A DAMPED EXPONENTIAL FROM 0 (X*E.P.(-X)) WITH A
         R MEAN OF 1.0.
         R
          ENTRY TO EXPRAN.
          FUNCTION RETURN (-LOG.(1.-RANNO.(0)))
         R
          ENTRY TO DMPRAN.
          FUNCTION RETURN (-LOG.(RANNO.(0)*RANNO.(0)))/2.
          END OF FUNCTION
         R
         R SUBROUTINE TO RETURN RANDOM NUMBER FROM DISTRIBUTION--
         R  P(A.P.2)(T)(EXP(-AT)) + Q(1/TAU) FOR T.LE.TAU   AND
         R   (A.P.2)(T)(EXP(-AT)      OTHERWISE.
         R
         R PROGRAM WORKS BY PICKING UNIFORM DISTRIBUTION WITH
         R PROBABILITY Q = 1-P, EXPONENTIAL WITH PROB. P.
         R
          INTERNAL FUNCTION (A,P,TAU)
          ENTRY TO SPRAN.
          WHENEVER RANNO.(0).G.P
            FUNCTION RETURN RANNO.(0)*TAU
          OTHERWISE
            FUNCTION RETURN 2.*DMPRAN.(0)/A
          END OF CONDITIONAL
          END OF FUNCTION
          END OF FUNCTION
```

BIBLIOGRAPHY

B. ARDEN, et al., The Michigan Algorithm Decoder, University of Michigan, November, 1963.

R. W. CONWAY, "An Experimental Investigation of Priority Assignment in a Job Shop," Memorandum RM-3789-PR, The Rand Corporation, February, 1964.

F. J. CORBATO, et al., The Compatible Time-Sharing System, The M.I.T. Press, Cambridge, 1962.

R. A. HOWARD, Dynamic Programming and Markov Processes, The M.I.T. Press, Cambridge, 1960.

IBM, Reference Manual, "General Purpose Systems Simulator II," Form B20-6346, 1963.

H. M. MARKOWITZ, et al., SIMSCRIPT -- A Simulation Programming Language, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1963.

J. H. SALTZER, "CTSS Technical Notes," Massachusetts Institute of Technology, Project MAC Technical Report MAC-TR-16, April, 1965.

INDEX