

NRL Report 6531

Multiprocessor Operating Systems

BRUCE WALD

*Intercept and Signal Processing Branch
Electronic Warfare Division*

April 11, 1967



NAVAL RESEARCH LABORATORY
Washington, D.C.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

CONTENTS

Abstract	ii
Problem Status	ii
Authorization	ii
INTRODUCTION	1
HISTORICAL BACKGROUND	2
Multiprocessing	2
Multiprogramming	4
Timesharing	5
COMMONALITY AND CONTRASTS	6
Multiprocessing and Multiprogramming	6
Multiprocessing and Timesharing	8
Multiprogramming and Timesharing	9
Synthesis	10
CURRENT STATUS	10
Problems Nearly Solved	10
Unsolved Problems	13
SUGGESTED ATTACKS	15
Analytic	15
Monte-Carlo Approach	18
SPECIFIC PLANS	18
CONCLUSIONS	21
BIBLIOGRAPHY	22
Referenced in Text	22
Unreferenced in Text	24

ABSTRACT

The history and present status (1965) of multiprocessing, multiprogramming, and timesharing are reviewed. It is concluded that, despite their diverse histories, these techniques are destined to be intertwined. Although the mechanical problems in operating systems that exploit these techniques have largely been solved and the difficult memory allocation problem is on the brink of solution, the important question of optimum operating system strategy in initiating, suspending, and terminating jobs is largely unexplored. Suggestions are made concerning models which might be suitable for both analytic and Monte-Carlo approaches to the optimization of operating system strategy and to the selection of optimum hardware mixes. An extensive bibliography is included.

PROBLEM STATUS

This is an interim report; work on this problem is continuing.

AUTHORIZATION

NRL Problem R06-07
Project RF-001-08-41-4552

Manuscript submitted December 14, 1966.

MULTIPROCESSOR OPERATING SYSTEMS

INTRODUCTION

This report surveys the current state of the art in operating systems for multiprocessor digital computers; it provides a historical perspective to the current interest in this field, and indicates some of the unsolved problems in the optimization of such systems. In the definition of the term "multiprocessor" we exclude such computers as the Honeywell H-800 (5) and its successors which are single-processor computers with special features to facilitate multiprogramming. We also exclude systems, such as the current Project MAC (8) configuration, which are multiple-access multiprogrammed systems, although we shall find a considerable body of problems common to multiprogrammed and multiprocessor systems.

We shall define multiprocessors as systems containing more than one identical arithmetic/control module, each of which can associate with random access memory and input/output (i/o) facilities to form a stored-program computer. The requirement that each arithmetic/control module (hereinafter called computer module) be an autonomous computer when equipped with memory and i/o eliminates from our definition such systems as the Control Data Corporation CDC 6600 (34) which contain multiple but highly specialized arithmetic units each of which is capable of executing only a fraction of the instruction repertoire of the system. The requirement that the computer modules be identical eliminates the many "computer with on-line satellite" systems now extant. Finally, we shall exclude from consideration such systems as the SOLOMON (14,28) computer which contain a large number of identical but not autonomous computer modules and are intended for the parallel solution of different parts of a single physical system.

We also require some criteria for distinguishing between multiprocessor systems and systems consisting of two or more conventional "computers" (i.e., computer modules plus memory and i/o) coupled either through their i/o facilities or through common memory. For this purpose we characterize a multiprocessor system as one in which all the main memory (i.e., directly addressable memory) is totally shared among all computer modules (except perhaps for a relatively small amount dedicated to computer module internal housekeeping functions) and in which any program can be executed with equal facility by any computer module.

Under these definitions there is now only one multiprocessor in use today, the AN/GYK-3(V) (Burroughs D-825) (1,2), although there have been many such systems announced within the last year which will presumably be operational in the next year or two. Among this latter group are versions of the IBM System 360 (19), the CDC 3870 (31), the General Electric Company GE-635 (20), and the Sperry Rand Corp. UNIVAC 1108 (29). Although experience with the AN/GYK-3(V) has overcome the initial skepticism as to whether such multiprocessors could function at all, little is known about the features an operating system must possess in order to allow the multiprocessor to function efficiently.

Note: This report was originally submitted to the Faculty of Electrical Engineering at the Graduate School of the University of Maryland in October 1965 in partial fulfillment of the requirements for the Ph.D. degree, while the author held a Thomas A. Edison Memorial Graduate Training Fellowship. It is being issued as an NRL report at this time because of the continuing growth in interest in and utilization of multiprocessors.

HISTORICAL BACKGROUND

Multiprocessing

The original interest in multiprocessing derives from the use of digital computers as information processing elements in military applications such as command and control, missile tracking and guidance, and tactical data processing. For many of these applications hardware malfunction would cause intolerable effects, and means were sought to reduce the probability of failure. Even the most stringent application of conventional quality-control measures often leaves an unacceptably high failure rate in high-component-count devices such as digital computers; redundancy techniques were therefore explored.

There are many levels at which redundancy techniques can be applied to digital computers. The most straightforward approach is to replicate at a low level of circuit complexity, either at the component level or at the single function level (e.g., gate or flip-flop). For example, if each diode is replaced by four diodes in series-parallel, the combination is invulnerable to any single diode failure, and the probability of two diodes out of four failing is much lower than the probability of a single diode failure. This technique has the disadvantage of greatly increasing the number of components in the computer and, therefore, the rate of component failure. However, it is an acceptable solution for situations such as missile guidance where the mission time is short and the computer is either expended in the mission or has long servicing and checkout periods between missions, when failed components can be identified and replaced.

A further disadvantage of redundancy at the lowest level is the increase in the number of component interconnections. The number of interconnections critical to system operation may well be higher in the redundant than in the simple system.

Redundancy at an intermediate level is usually implemented by checking and occasionally by correcting the arithmetic and transfer operations. While simple parity checking has become standard practice in commercial and scientific applications, the detected error may well be no less catastrophic than the undetected error to the real-time military application. Error correction, on the other hand, often involves substantial increases in component count. For example, meaningful voting requires the voting circuitry and three independent subassemblies for casting ballots and leads to a nearly fourfold increase in circuit complexity (18).

In summary, redundancy on the low or intermediate levels is most applicable when the basic computing requirement is modest, for it is in these situations that the increase in circuit complexity is tolerable. These situations would often require the development of a custom computer, for environmental or other reasons, and the redundancy techniques can be applied during this design. Furthermore, such computers are often single-function, and error detection and correction techniques specialized to the computation being performed can be employed.

When the computing requirement is high, however, these techniques are less attractive. The expense of quadruplicating a large computer is unattractive as is the development cost of both the hardware and the software for such a system. The usual approach during the late 1950's and early 1960's was to employ a state-of-the-art large-scale computer (usually an IBM 704 early in this time period, or an IBM 7090 later) and to couple pairs of these computers to the system. This was a satisfactory solution, at least for ground environments, since well debugged computers with extensive software libraries were available at no additional development cost. It was assumed that the internal error detection features of these computers were sufficiently effective so that one could be kept on line, the other could perform internal diagnostics, and they could be switched if one failed.

A complication occurs in updating the memory of the standby computer; later systems kept the standby computer in a "shadow" mode in which it "eavesdropped" on the inputs of the prime computer, produced the same outputs as the prime computer (although these outputs went nowhere), and was always ready to assume the functions of the prime computer. In principle the outputs of the prime and the shadow computers could be compared, but unless a third computer was available to resolve disagreements this would add little to the internal error detection capabilities of the computers. Sometimes a "cross-talk" channel would be provided between the computers (10).

These systems often had such peripherals as tapes and drums. In a few systems these were not duplicated but were connectable to either computer with a switch. The total complement of tapes would be higher than the minimum necessary for operation, but the redundancy was less than the 100 percent of the computers.

The most refined form of such systems is exemplified by a system (13) in which a computer was deliberately chosen that was too slow to perform the total computational requirement. Instead, a number of computers were interconnected with the peripherals through a switch and the problem was segmented so it could be performed by any three computers. Five computers were installed to provide a high probability of always having three computers available with a computer redundancy of only 67 percent. Other systems of this type have been described (4).

A few years before this system was installed, however, it became clear to a number of investigators that the principle of the "switch" and a redundancy of less than 100 percent could be carried into the design of the computer itself. It would be necessary to segregate the "computer" into computer modules, memory modules, and i/o modules and allow them to interact freely through a high-speed switch as well as allow the i/o modules to interact with the peripheral complement through another switch. By matching the power of the modules to the computational task in such a way that several modules of each type would be required to fulfill the task, high reliability could be assured at a cost of only modest redundancy by providing one or two additional modules of each type.

Two of these proposals circulating in 1960 were selected by the Government for development. The first was a proposal by the Ramo-Wooldridge Corporation for a machine dubbed the RW-400 (23-25) and was supported by the Air Force; the second was a proposal by the Burroughs Corporation for a machine dubbed the D-825 and was supported by the Navy and now bears the military nomenclature AN/GYK-3(V).

Although some RW-400 hardware was built, the development was never completed. Chief among the weaknesses of the design was a centralized switch and a special control processor to control it. Failures in the switch and the control processor would lead to system failure, and the control processor contained a substantial number of components.

In the AN/GYK-3(V), however, the "switch" was distributed among the memory modules. Each memory module was equipped with five busses over which requests could be made and which contained the necessary circuitry to queue simultaneous requests. Each computer module (normally the system accommodated three) was connected to a bus on each memory module. A subset of i/o modules was connected to a switch which was connected to a bus on each memory module, and a second subsystem of i/o modules was connected to another switch connected to the fifth memory bus. Each i/o module was connected to all peripherals through its own switch. Thus most component failures would disable only one module, although a few failures in an i/o bus could disable half the i/o modules. The computer modules were equipped with relative address registers to facilitate multiprogramming and with a small thin-film memory for internal register housekeeping. Typical installations have had about 40 percent module redundancy.

Multiprogramming

At this point it would be appropriate to interrupt the history of multiprocessing to consider the history of multiprogramming. The earliest digital computers were equipped with on-line i/o equipment (punched paper tape and punched card) through which programs and data were read and outputs punched. Later, high speed printers were available for on-line operation. Nevertheless, the speed of computers quickly outstripped the speed of peripherals, and users found their expensive computers idle a substantial proportion of the time while waiting for i/o operations. Three attacks have been made on this problem.

The first minimizes i/o time by increasing the speed of i/o operations. Because the speed of printers, card readers, etc., cannot be increased without limit, it has become common practice to convert the input information to magnetic tape which the computer reads, and to have the computer write its outputs on magnetic tape which is later used to drive a printer.

A second approach is to perform these conversions under the control of a satellite computer which is used to control the peripherals and to write the input tapes and read the output tapes. Direct core-to-core communication between the main computer and the satellite computer via i/o channels is also possible.

The third approach involves the provision of sufficient control in the i/o channels so that they need a minimum of supervision by the computer. In this way the need for a satellite is eliminated. It is this third approach that naturally leads to multiprogramming, which may be defined as having several independent programs in a partially run or a ready-to-run state (usually, but not invariably in main memory) and cycling the computational service from one program to another in an attempt to insure that the computational facilities of the system are never idle.

Multiprogramming may be accomplished entirely by hardware, almost entirely by software, or by intermediate mixes. One of the early attempts to provide multiprogramming by hardware was the Honeywell H-800 (5,21,22,25). This machine provided special registers to store the state (e.g., the location of the next instruction) of up to eight programs. The control section operated in a fixed cycle, executing one instruction from each program and then going to the next program. If a program was performing an i/o operation, it would be skipped in every cycle until it was ready to resume execution. Since the H-800 was a three-address machine, most instructions were self-contained and the "housekeeping" in transferring between programs was tolerable.

One disadvantage of this type of multiprogramming is that, when no i/o operations are being performed, each program is being executed at one-eighth the rate it would be executed if it were the sole occupant of the system. In addition to intolerably long completion times for programs of high intrinsic priority (e.g., a real-time program controlling an external device and having a high ratio of computation to i/o), this also results in each program residing in core for a long time. While it is desirable that there be more than one program in core simultaneously to assure a high probability of there being a program not performing i/o and therefore capable of absorbing the attentions of the computational facility, the scheme employed in the H-800 tends to be prodigal with core memory -- a resource often costing a substantial fraction of the entire system.

Multiprogramming can also be performed primarily by software. The minimum hardware requirements are a good interrupt system and perhaps a set of "relocation" or "base" registers. Several programs will occupy core at the same time; since the run-time environment is unknown at compile time, they must be compiled and filed in "relocatable-binary" form. At run time the operating system loads the program into available memory and either modifies the program addresses or, preferably, inserts the address of loading into a relocation register.

The usual strategy of such a multiprocessing system is to require that all programs perform i/o operations by making macro calls on the operating system. The operating system institutes the i/o operation and turns over control to another program. When the i/o operation is complete, the operating system is notified by an interrupt and has the option of turning control back to the first program or permitting the second program to continue until its next i/o operation. Unnecessary transfers of control are avoided not only because core can be used economically and programs can be completed in minimum total time but also because substantial processor time is required for the housekeeping associated with the transfer. When a program is completed (or releases a substantial block of core space), the operating system selects a program from its file of programs to be run, loads it in core, and notes the necessary relocation information.

Some operating systems perform a very limited amount of multiprogramming while primarily relying on other techniques to keep the processor busy. Typically only one program is active at a time, with i/o performed to a drum or other high transfer rate device, but the operating system must devote processor time to reading outputs of the last program from drum to core, formatting them, and transmitting them to conventional peripherals; it must also devote processor time to reading input devices for the next program, formatting the data, and transferring the images to the drum.

Timesharing

The third thread that must be considered, which has generated much interest of late, is timesharing. A timesharing system will typically have a substantial number of local or remote terminals, each of which is capable of independently generating requests for computer service. These terminals may be mechanical, such as multiple computer-controlled machine tools or the communication circuits of a message store-and-forward switching center, or they may be human operated. In the latter case the scope of computer service required may be quite narrow as it is for an airline reservation or inventory control system, where the terminals are simply remote peripherals transmitting data to and receiving data from a conventional computer program, or the service requested may be the full resources of the computer (as delegated through the operating system) in the case of a multiple user on-line program debugging system.

The common characteristics of timesharing systems are: computer service is required by terminals over periods much longer than the processor time utilized during the period, and the system must be available for servicing each user's request with a waiting time much shorter than the period of connection.

The simplest timesharing systems involve only a single program which provides appropriate responses to the terminals. An example would be the Control Data Corporation "Remote Calculator" (31). The calculator itself is simply a keyboard and a set of indicators. The calculator is given characteristics intermediate between those of a desk calculator and a stored-program computer by the program within the central computer. It is a relatively simple matter to provide sufficient space within this program to accommodate numerous remote calculators. It should be noted that all transmissions from the remote calculator to the computer, whether they are considered data or instructions by the user of the calculator, are data to the computer program which interprets them. Thus the user has no knowledge of the computer resources activating his calculator and has no way to exercise them, except within the vocabulary acceptable to the program in the central computer.

The IBM "Mohansic System" (19) lies at the other extreme of the timesharing spectrum. In this system each user "owns" the entire system, except for the common operating system which handles i/o and owns a disc. When a user requests service, a disc-to-core transfer reestablishes "his" system; when his request has been completely serviced, a core-to-disc transfer makes the machine available to some other user.

One of the best known timesharing systems, Project MAC (8) currently uses an IBM 7094 computer with certain modifications to afford relocation capabilities. Both types of service are provided to the terminals. Interpretive systems provide highly specialized problem-oriented languages to terminals. Other terminals may deal in the conventional machine-oriented business of compile requests, run requests, etc. Similar systems are described elsewhere (26,27).

COMMONALITY AND CONTRASTS

Multiprocessing and Multiprogramming

It is obvious that a multiprogramming system need not be a multiprocessing system; the question of whether a multiprocessing system need be a multiprogramming system requires some discussion.

It should be pointed out that if the only incentive for choosing a multiprocessor is reliability at modest redundancy, and if the computational load is absolutely constant so that there is no desire to make use of the redundant modules whose availability cannot be assured, there is no reason to insist that a multiprocessor have multiprogramming capabilities.

As an example we may consider an Air Force application of the AN/GYK-3(V) to an air defense problem (30). The total equipment complement consists of two computer modules, six memory modules, and three i/o modules. Of these, one module of each type is reserved to form a "confidence system," whereas the operational program is written as a simple, nonmultiprocessing, nonmultiprogramming system and is constrained to run in the remaining modules while diagnostic programs run in the confidence system. In the case of hardware malfunction the scope of the diagnostic program is increased to examine all modules; the offending module is replaced by a corresponding module from the confidence system while the offending module is being repaired.

This system can be criticized on the grounds that there is no way the operational program can make use of the redundant modules. The criticism is defended against on the grounds that the operational program has no desirable but nonessential components appropriate for performance by redundant modules; therefore, it is not worthwhile to solve the problems of memory allocation, etc., inherent in a multiprogramming system. Nevertheless, it is understood that the Air Force (37) has recently expressed interest in providing a multiprogramming capability in these systems.

Actually, it was always intended that the AN/GYK-3(V) be used as a multiprogrammed system, even in single computer module configurations, and considerable attention was paid in the hardware design. An "Automatic Operating and Scheduling Program" (AOSP) was proposed concurrently with the hardware proposal, and work was initiated by Burroughs Corporation personnel (33). In recent years this work has been supported by NRL both in connection with a particular Navy application and for the advancement of the operating system art.

At this point we shall enumerate some of the features of the AOSP and attempt to justify them as required for multiprocessing, multiprogramming, or both.

1. The operating system must be executable by any computer module out of any memory module(s). This requirement derives from the necessity of maintaining the survival potential of a multiprocessor; consequently it becomes impossible to dedicate particular modules to the execution of the operating system. This leads to the concept, first expressed by Wilkinson (36), that, in a multiprocessor, programs (including the operating system) are not executed by computers but control whatever computers and

other resources are assigned to the execution. In a multiprogrammed system this is not an absolute requirement. However, it is often convenient to divide the operating system into a "resident" which always occupies a fixed area in core and dependent subroutines which are called when required and are allocated available memory space in the same fashion as user programs.

2. A user program must be executable out of any memory. This requirement is common to both multiprocessing and multiprogramming, although it is more stringent in connection with multiprogramming. In strict multiprocessing it is only necessary to be able to "rename" memory modules when a redundant module is substituted for an unserviceable module. In multiprogramming, on the other hand, it is impossible to predict at compile time what other programs will be in core at run time and hence what core will be available to a given program. Therefore, each program must be executable out of any available core memory.

3. A distinction must be maintained between PA's and DA's. A PA (program area) is a pure procedure, i.e., the string of instructions that control a computer. A DA (data area) is the data and working locations involved in the particular execution of a program. It is desirable to maintain this distinction in a multiprocessor, enabling a single PA to control several computer modules which are performing identical processes on distinct DA's. Similarly, in a multiprogrammed system, it is desirable that several programs running "simultaneously" (i.e., in an interleaved fashion) need not require separate core allocated to private copies of subroutines common to more than one program. Furthermore, this distinction facilitates recursive programming and reentrant subroutines.

Some additional terminology will be introduced at this point. Henceforth we shall use the term "program" to refer to the combination of a PA and a DA and the phrase "incarnation of a program" to refer to the combination of the PA and one of several DA's. SS (simple subroutine) will denote a procedure that has no DA, i.e., one which has its immutable constants integrated with its PA and either confines its writing to the registers of the computer it is controlling or to the DA of the program that called it. DO (data object) will denote an object external to a program but which may be read or modified by one or more programs. The term "object" will be used to refer to a PA, DA, SS, or DO. The term "job" will be used to refer to a task assigned to the system from the external world. A job always involves the execution of at least one program and may involve multiple incarnations of that program and linkage to other objects.

4. The objects involved in the performance of a job must be independently allocatable. While the initiation of a job will involve the specification of one program and perhaps some parameters, that program may refer to other objects and also require replication of its own DA. It is essential that in collecting the necessary objects the operating system be able to use those objects previously allocated into core without restructuring the memory. It is desirable that the process continue beyond load time through run time; i.e., that it be possible for a running incarnation of a program to request that the operating system link to it an object that may or may not be already in core and to inform the operating system that it no longer requires an external object. This capability leads to the more efficient use of core memory and is required for efficient multiprogramming. Maintaining this capability in multiprocessing complicates the system, but it is a prerequisite for the use of multiple computers on a single job.

5. If the operating system diverts a computer module from the execution of some incarnation of a program, it must be prepared to restore control to that program either with external objects unchanged and unmoved or with the links in the DA of the diverted incarnation (and the links of other suspended DA's utilizing such objects) suitably modified to point at the new locations of such objects. This is a fundamental requirement in multiprogramming and even in some conventional operating systems which do not allow "user" multiprogramming but multiprogram utility functions along with a single user job.

Naturally it behooves the hardware designer to make the process of restoring control and links as efficient as possible. It should be noted that the AOSP employs the rule of not disturbing the memory allocation or the links of a suspended incarnation of a program and its external objects and thus sidesteps the difficult problem of dynamic reallocation (15) of memory at the expense of not having the allocated memory available for other tasks during the period of suspension. More recent computer designs have adopted such techniques as mixed hardware/software attachment registers, associative allocation memories, dynamic paging, to reduce the necessity of performing dynamic reallocation. Multiprocessing complicates the situation somewhat, since objects referred to by the suspended incarnation may also be required by some other incarnation of the same or a different program still undergoing execution by other computer modules.

6. If a running incarnation relinquishes execution to await some external condition (e.g., passage of time, completion of i/o, external interrupt, a computation by another incarnation, etc.), the operating system must reestablish execution when the specified criteria have been met with external objects, and links must be preserved. This requirement is contained within the previous one, but the distinction is in the case of voluntary relinquishment when it may be reasonable to require that the user program perform some administrative work to prepare for its reestablishment.

7. A job must be capable of initiation in any computer module, and a suspended job must be capable of reinitiation in any computer module. This is a fundamental requirement of multiprocessing.

8. The operating system must perform i/o operations for users and inform users as to the status of these operations. The decoupling of i/o is a fundamental requirement of multiprocessing, as the suspension of a job while i/o is taking place and the dedication of computational resources to another job during this period is the primary motivation of multiprocessing. It should be noted that two or more jobs, segments of which are being executed alternatively, may demand the use of the same physical peripheral device, and the operating system must be capable of satisfying the needs or both. In a multiprocessing system, two or more jobs actually being executed concurrently may demand the same peripheral device.

9. The operating system must schedule the execution of tasks, taking note of intrinsic priorities, queue status, conflicts between resources required for different tasks, and resources that can efficiently be shared among tasks. This is a fundamental requirement of any worthwhile multiprocessing system; yet it remains a neglected requirement. Despite the "S" in the acronym "AOSP," the scheduling in that system remains primitive, taking little cognizance of potential conflicts before they occur, and makes no effort to identify and facilitate resource sharing. It is the investigation of means to combat this deficiency that is the motivation for the research proposed. The deficiency is more serious in multiprocessing, as the potentials for cooperative resource use (e.g., objects used in common by two or more jobs running concurrently) are higher.

Multiprocessing and Timesharing

The relation between multiprocessing and timesharing may at first seem rather tenuous, especially when it is considered that a number of reasonably effective timesharing systems now exist and none involve multiprocessing. Nevertheless, one well-known system (Project MAC) has recently announced plans to use a multiprocessing General Electric GE-635 computer system, and it seems safe to predict that, if timesharing proves itself economically advantageous, systems designed for timesharing (as distinguished from conventional systems modified to be feasibility demonstrators of, or test beds for, timesharing) will be multiprocessors.

The primary motivation for the incorporation of multiprocessors into timesharing systems will be identical to that which led to the development of multiprocessors for incorporation into military systems — reliability. In a batch-processing closed-shop computing center, system failure is tolerable provided that repairs are accomplished with sufficient speed to prevent substantial increments to the normal turnaround time of the installation, and provided that total availability is sufficiently high to permit the processing of the workload. In a timesharing system, on the other hand, there are many users who expect service in times (microseconds to seconds) commensurate with their requirements and who will be intolerant of system unavailability.

Fundamentally, timesharing is attractive for the same reasons that public utilities represent an efficient method of providing electrical power or telephone communications — it is more economical to share large central plants than to provide each customer with an electric generator or a radio transceiver capable of meeting his peak demands. On the other hand, the public utility must have redundant generators, alternate long-haul trunks, etc., to minimize the probability of complete system failure. Furthermore, the system must be so organized that a large fraction of the plant is usually performing economically useful functions, and it must be possible to rapidly reconfigure the network to meet instantaneous demands. These are precisely the arguments that lead to the design of multiprocessors with flexible multiprogramming capabilities.

An example of the close connection between the multiprocessor and the public utility may be found in the new A.T. & T. ESS (Electronic Switching System) (16). Viewed from the outside, this system is similar in function to the conventional telephone system with some improvements in flexibility and presumably substantial improvements in economy and reliability. Viewed from the inside, however, this system resembles a multiprocessor with many flexibly interconnected computational and memory modules. Indeed, the designers of this system have referred to it as an "immortal computer," and, if not for its limited instruction repertoire and single function (although not single program), we might find this system falling within the central area of interest of this report.

The other motivation for the incorporation of multiprocessors into timesharing systems is responsiveness to peak demands. While one could conceive of a very fast single computer module that could handle the peak computational requirement by sequencing through the active programs, the overhead of this sequencing might well be substantial; it would appear more attractive to provide a multiplicity of processors, each handling fewer requirements, with several combined on a substantial problem requiring rapid service.

Multiprogramming and Timesharing

It seems evident that a timesharing system should be a multiprogramming system, although a few counterexamples have already been given. It should be recognized that the timesharing system allows a wider range of multiprogramming techniques than does the batch-programmed multiprogrammed system.

In the batch-programmed multiprogramming system, the main incentive to switch program control is to find useful work for the computer module during times when the program is awaiting i/o operations. In the most elementary multiprogramming systems, this work is limited to jobs preassigned to the operating system itself; in more general systems, other user programs may be considered. However, the number and type of i/o operations that can be done before it is appropriate to resume the suspended program is quite limited; an attempt to run another user program would be made only if it was already in core or could be brought to core in a time short compared with the expected time-to-completion of the i/o operations of the suspended program, without interfering with these operations.

In a time sharing system, however, programs will often be suspended because they have met the immediate requirements of the user, and it is therefore appropriate to consider i/o operations requiring many milliseconds in order to find useful work.

An additional complication that timesharing adds to multiprogramming is that of maintaining "turnaround time" at a level appropriate to each customer. Multiprogramming operating systems for a batch-programmed system need only optimize processing rate; they can do this by selecting a set of jobs from the job queue that tends to keep all modules busy, although it has been observed that current-day operating systems are not very sophisticated in their choice. In a timesharing system it may be necessary to allow inefficient instantaneous states (i.e., not all modules busy) in order that each customer's response time requirements may be met.

Synthesis

The discussions of this section may be summarized by observing that although there are certain problems which in the strictest sense apply only to multiprogramming, these problems must be resolved for efficient multiprocessing. Furthermore, timesharing simply represents a particular class of problems that are appropriately performed on a multiprogrammed multiprocessor.

Thus an efficient multiprocessor operating system will provide full multiprogramming capability and will be able to accommodate timesharing applications as well as conventional batch-processing and the real-time requirements of military and industrial control systems.

CURRENT STATUS

Problems Nearly Solved

We now consider a number of problems in the design of multiprocessor operating systems that were considered critical several years ago but which are now either solved or on the threshold of solution.

It is no longer controversial that operating systems can be built that keep computer modules from interfering with each other while permitting system throughput nearly as many times greater than that obtained with a single computer module as there are computer modules, provided that a large amount of core memory is available and there is a statistical universe of problems to be processed.

When a multiprocessor is used to attempt to raise the throughput on a single problem, the potential gain may not be realized unless the programmer is careful to identify potential parallel paths and bottlenecks. It would be fair to state that this has been accomplished in several systems programmed by NRL, although the insensitivity of current compilers to these problems has made the work difficult by forcing programmers to keep track of relationships which would ordinarily be handled by compilers. Nevertheless, this problem can be classed as "nearly solved," since suitable compilers can be built to imitate the clever programmer whenever there is sufficient incentive to do so.

The problem of conflicting requests for i/o devices is in a similar state. The technique used in the AOSP, while workable, is not wholly satisfactory. This technique permits programs to be exclusive or nonexclusive users of a device. A drum is an example of a device that would normally have many nonexclusive users. On the other hand, a line printer would normally have one exclusive user at a time to prevent the commingling of outputs from several programs. The technique is not wholly satisfactory, because an

exclusive user of a device remains so even while suspended, and a program also requiring this device could not be run during this period of suspension. A better technique which is being considered for incorporation within the AOSP, as well as being planned for several more modern operating systems, is to further decouple the user programs from i/o by making their macro calls to the operating system refer not to physical peripherals but rather to "logical devices," i.e., buffers which the operating system can unload to the physical peripherals.

Perhaps the most difficult problem, one that can still be fairly classed as "almost solved," is that of memory allocation and protection. Current multiprogramming operating systems work well, provided that a large amount of memory is available. While the original motivation of multiprogramming was to keep computers busy, in a modern system the cost of core memories may exceed the cost of computer modules, making efficient use of memory a requirement. Furthermore, if it is desired to accommodate undebugged programs in the system and allow them full system resources (e.g., not insist that they be run interpretively), it is necessary to protect the system and the other users from undebugged programs.

Again the AN/GYK-3(V) will be used as an example of current art. The AOSP keeps track of available blocks of memory in a size-ordered linked-list. When it allocates a program it selects and loads the smallest blocks of sufficient size that accommodate the PA and the DA. It inspects the declarations of external objects and, if an object is declared as "necessary" and is already in core, loads the absolute address of that object in an ABI (Adaptor Block Item) Line of the program being allocated and increments the user count of the object. If an object is not in core, it loads it (and its necessities), fills in the ABI Line, and sets the user count to one. When all necessary objects are linked, the AOSP loads an upper and lower memory bounds register with the upper and lower limit address of the program and its dependent objects, and loads the location of the PA and the DA into the BPR (Base Program Register) and BAR (Base Address Register), respectively, of an available computer module.

The program can read and write in its DO's and in the DA's of linked programs by indirect addressing through the ABI Lines. Similarly, it can call a linked SS by an indirect subroutine jump through an ABI Line. Finally, it can call a linked program, by an indirect jump through an ABI Line, to the new program which will employ a "starter patch" to reset the BAR and BPR.

An external object can also be declared as "conditional," in which case allocation will not be attempted until the program executes a "find" macro call on the AOSP. Similarly, a program can execute a "release" macro call on the AOSP, in this case the ABI Line will be modified to prevent subsequent use, the user count on the external object will be decremented by one, and, if it thus reaches zero, the object will be released, i.e., the space it occupies will be relinked into the available space map. Once allocated, an object is never moved as long as it has a nonzero user count, i.e., no attempt is made at dynamic reallocation.

ABI Lines are also declared for i/o devices and at allocation time are filled in with the absolute address of that portion of the i/o control package which controls that particular device.

The AOSP has proved quite workable and succeeds in optimizing memory utilization. It accomplishes this by not insisting upon consecutive blocks of memory for each of the many objects involved in a job and by assigning each object to the smallest available block of memory that will contain it. It further permits objects to be shared by many jobs to the extent that their nature permits. Nevertheless, some penalties have been suffered in order to achieve efficient memory utilization.

The first problem is the indirect addressing involved in communicating with DO's. This indirect addressing is required by the independent allocation of the DO and costs an extra memory cycle time per reference.

The second problem is the lack of full memory protection. The memory protection registers span a continuous block of core, but since the objects of a job are independently allocated, they will not in general be contiguously located. Thus the memory protection limits will allow writing not only in the job being executed but also in objects belonging to other jobs which happen to lie between the limits. Furthermore, the ABI Lines themselves are not protected.

The third problem is that of memory fracturing. It is possible that, if the system operates for a long time readying and releasing objects of varying length, the available memory may become fractured into a large number of very small blocks any one of which is too small to contain the larger objects of a new job. While this phenomena has been only a minor annoyance in the NRL work, military systems usually involve considerable readying and releasing of the same objects which tend to get allocated to the same place (a block of space which fits them exactly), and in a "job-shop" application fracturing might be a real problem.

Nevertheless, these problems are classed as "almost solved," since new computer designs have met them head-on. To remove the indirect addressing penalty, the IBM System 360 Computer (19) employs a number of base registers in lieu of the pair (BAR and BPR) provided in the AN/GYK-3(V). The base registers then replace the ABI lines. While the solution is not wholly satisfactory considering that the use of base registers for this purpose reduces their use as index registers (indeed the multiple indexing facility of the AN/GYK-3(V) could have been used for this purpose if we had been willing to sacrifice one of the three subscripts), it does represent an improvement because 16 memory protect "keys" are provided. Thus, memory protection of other objects allocated between objects of the current job is possible.

The modifications of the General Electric Co. GE-635 for Project Mac and for Bell Telephone Laboratories (6) involve the use of hardware "attachment" registers, each involving a base address and memory protection limits. An associative memory is used to mediate between the limited number of hardware registers (8) and the potentially unlimited number of "ABI Lines" in a manner analogous to the suggestion of Conway (7). An almost identical system has been proposed by IBM for multiprocessor modifications of the System 360 computer (19).

The memory fracturing problem is generally being attacked through "paging." For example, in the CDC-3870 (31) the memory is treated as if it were organized as a set of pages. The page length is variable, but for the sake of illustration assume that it is fixed at 512 words. Under this assumption the lowest nine bits of an effective address may be considered as a word number (line number) within a page and the upper bits as a page address. An auxiliary allocation memory translates the page numbers appearing in the program (logical page numbers) into hardware page addresses (physical page numbers). Thus, while an object occupies a continuous block of memory as far as the programmer is concerned, it can be allocated into nonconsecutive physical pages, provided the allocation memory is suitably loaded. External objects can be referenced by assigning them distinct logical page numbers and suitably loading the allocation memory. While a job is running, the structure of the job is shown in the allocation memory; unallocated logical page numbers are made to refer to trapping routines. Thus the fracturing, memory protection, and external reference problem are solved with one technique, although at the expense of some wasted core, presumably one-half page per object.

Dynamic reallocation is another solution to the fracturing problem. For example, in the otherwise unambitious operating system for the GE-625 (user programs cannot share

objects external to the fixed-location operating system) dynamic reallocation is practiced (20). It should be noted, however, that such reallocation is much more difficult in a multiprocessor system or even in a multiprogrammed system allowing shared objects; no plans to accomplish dynamic reallocation in the more stringent environments have been announced.

The purpose of this discussion has been to justify a number of simplifying assumptions that will be made about multiprocessors and their operating systems. We assume that the main core memory of a multiprocessor consists of a number of independent, asynchronous, identical banks, each characterized by two parameters: size and access time.

We assume that the computer modules can be characterized by a mean instruction rate that does not depend on whether the data handled is in the executed program or an external object. We assume a small but finite processing requirement to switch the attention of a computer module from one job to another; this requirement is to be a function of the number of external objects and the existence in core of the program and the external objects. We also assume that i/o requirements never conflict except on a total demand for i/o channels or peripheral device basis.

Unsolved Problems

Under the assumptions just given, and under the assumption that the "mechanical" problems in writing an operating system are soluble, the major unsolved problem is specifying the strategies an operating system should employ in deciding which program to allocate and run from a population of run requests. Fortunately, the lack of work on this problem has not prevented the development of multiprocessor hardware or operating systems, although it would seem that the same economic considerations that have led to interest in multiprocessors in nonmilitary applications should have motivated considerable effort toward the optimization of operating systems. Although Conway (7) has opined that "parallel processing is not so mysterious a concept as the dearth of algorithms which specifically use it might suggest," there exist no general results which give reliable guidance to a programmer in his specification of parallelism within his own program; there exist no operating systems that use more than the crudest heuristics in deciding which program to allocate.

Let us trace the progress of a program through a multiprocessing system in an attempt to highlight some of the decisions that must be made by the operating system. The operating system is first informed of the existence of input media loaded in some input device (e.g., a deck of cards placed in the input hopper of a card reader) and presumably at its first opportunity reads in the information, transfers the bulk of it (e.g., programs, data) to some bulk storage device, and retains certain information (run-request parameters, declarations of external objects, etc.) within its own DA's or DO's. From this time, until the initiation of the job, certain costs are accruing. The first is the occupancy of the bulk storage device, a cost that may be small in comparison with core occupancy but which may not be negligible. The second cost involves the residence of the run request information within the operating system itself, a residence which probably must be maintained in core. The third cost is in the processor time used to consider this information each time the scheduling algorithm is run. The fourth, and possibly the most significant cost, is the cost to the user of the delay in the initiation of his program. This cost is a nonlinear function of time and is usually not stated by all users in a consistent measure.

Two major problems for the operating system are deciding how often to run the scheduling algorithm and how extensive each run should be. Frequent running implies a considerable processor load; infrequent running risks inefficiency in the use of system resources. It is possible that a distinction should be made between scheduling between

"hot jobs," which are those in core or in the process of being brought to core, and "cold jobs," which are uninitiated run requests, although in the implementation of the AOSP this distinction was dropped. The scheduling of computer modules among hot jobs is a relatively simple matter — if real-time requirements are momentarily neglected — as there is no reason to schedule unless a hot job relinquishes one or more computer modules for some internal reason (e.g., a wait for i/o, a wait for linkage to an external object requiring i/o operations to ready it, the termination of parallel execution path(s), or a programmed delay) or unless the operating system becomes aware of a condition that might recommend the resumption of a suspended hot job [(e.g., the completion of an i/o operation, the availability of an i/o device (or the buffer area equated to it), the completion of a delay, or the termination of a program and the consequent release of core which could be allocated to another program previously suspended because a request for additional core space could not be honored)].

The cold job scheduling problem is considerably more difficult, but the desirability of maintaining the distinction between hot and cold jobs may be seen by considering the problems encountered in the AOSP where the distinction is not maintained. Responder, the program that honors run requests, has highest system priority. For each run request it creates a "job table" describing the run request which resides in core within the AOSP. This job table has associated with it responder's pre-emptive priority which cannot be lowered until the job is initiated, so readying the job takes place. Readying usually involves bringing the program into core, linking it to the job table and to the declared necessary external objects, and bringing the latter into core if not already there. If the total of run requests exceeds the system core space, the lowest priority job(s) will be terminated; these jobs are the ones in progress, and that progress will be lost.

Although the situation could be improved by giving the process of readying a job the job's intrinsic priority rather than AOSP's pre-emptive priority, making it tempting to relegate the problem to the class of "mechanical" problems, there is a more fundamental difficulty. The decision to convert run requests into hot jobs immediately, was motivated not solely by the inability to find an algorithm to decide when the conversion should take place but to some extent by the desire to keep core full of hot jobs to occupy computer modules when their current job was suspended, awaiting i/o. Obviously, if a computer is expected to be free only for the length of time required to complete an i/o operation, it makes little sense to try to ready a new job to occupy that computer module if the readying also involves i/o. However, this argument neglects the fact that core requirements of running jobs fluctuate radically if the user programmers have conformed to the spirit of multiprogramming and request external objects only when actually needed. Thus, if the operating system packs the core with jobs and a running program requests the operating system to find and ready an external object, the running program will be terminated if its intrinsic priority is low, or else a job previously readied will be terminated and the labor expended in readying it will have been wasted. It has been suggested that this problem could be ameliorated by not terminating jobs when a core crisis occurs but instead rolling them out to bulk storage. This would be rather difficult in the AN/GYK-3(V), since the entire job structure would have to be restored to the same absolute locations, but it might be feasible in planned future systems employing paging and/or associative memories to circumvent the difficulties of dynamic reallocation. Nevertheless, there are costs in processor and i/o channel time in roll-out and roll-in. The fundamental problem is that the analysis has never been performed to develop algorithms which could be used to resolve the conflicting desires, keeping core space available to accommodate the dynamic space requirements of hot jobs, and filling core with jobs to minimize the probability of an idle computer module.

As previously suggested, once the job is running, scheduling is easier and mostly mechanical if real-time requirements are ignored. One problem remaining is the development of a "running" priority as distinguished from the intrinsic job priority. Consider two readied jobs denoted by A and B with A having higher intrinsic priority and starting

first. In this discussion assume that only one computer module is available (e.g., all other modules are executing still higher priority jobs). Suppose that A suspends itself for some reason such as awaiting i/o in connection with a slow device; B will now begin running. The interrupt at i/o complete time will invoke the scheduler, which must decide whether to resume A or continue B. In the AOSP this decision is made solely on the basis of job priority; thus in this case A will be resumed and B suspended because of the higher priority associated with A. But these priorities are simply the intrinsic job priorities; in the AOSP they are fixed, whereas in more elegant proposed systems they increase with time. In the most ambitious system imaginable, the priority would vary (generally in a nondecreasing fashion) to reflect the marginal cost of another delay of one time unit in the initiation of the program. It is believed that this intrinsic priority, while suitable for consideration in the loading and readying scheduler (along with core state and other variables), is inappropriate for determining the resumption of programs. For this purpose it should be replaced by a running priority which reflects the cost of keeping the hot job in a suspended state. Suppose that in the execution of B a very large structure of dependent external objects has been created and that B is near completion. It would then seem more sensible to continue B and free this space rather than to resume A and have little space available to the allocator.

It is realized that formidable difficulties exist, both in the extraction of the necessary parameters to calculate such a running priority and in the calculation itself, but attention to this problem is considered overdue.

Further complications ensue when consideration is given to real-time systems which may be defined, in analogy with Patrick, as systems containing programs whose marginal cost of delay contain delta functions. In a strict real-time system the requirements of all such programs must be met, even at the expense of seriously delaying service to the non-real-time users.

An intermediate situation arises in timeshared systems where the user is human, and in some systems where the user is a machine tool which will suspend work (and not damage the piece being worked), if computer service is suspended. In these situations intrinsic priority cannot be completely ignored in the calculation of running priority; serious consideration must be given to roll-out schemes, since the accrued cost during a roll-out roll-back cycle may well be tolerable. The situation is made more complex by the existence of "background" problems which have a low and relatively time-invariant intrinsic priority. The operating system must decide whether to commit resources (particularly core memory) in order to make progress on the background problems, or to leave resources idle to prevent serious delays in honoring resource request from those programs having some of the aspects of real-time jobs.

SUGGESTED ATTACKS

Analytic

An obvious approach toward the solution of these problems is through mathematical analysis. It must be recognized that analysis would have to depend heavily on probability theory, and that results would be in terms of expectation values. The various inputs would have to be modeled with enough detail to render the results meaningful, but could not be so specific that the range of applicability would be too narrow.

One could not hope to make an analysis that was completely hardware independent; indeed, the investigation should allow for the consideration of reallocation of the hardware budget among the modules for optimum response to a given distribution of computing requirements. Fortunately, enough production experience exists on the AN/GYK-3(V) to estimate the costs of various modules of differing capabilities. Production cost is

emphasized here because the various modules of a commercial product line are marketed at widely differing multiples of their production costs for reasons of competitive strategy.

The basic parameters characterizing a multiprocessor are the number of computer modules, memory modules, i/o modules, and the size, random access time, and transfer rate of bulk memory. All modules can be assumed asynchronous, and all modules of a given type can be assumed identical without serious loss of generality.

For the purposes of this study, computer modules will be described by only two parameters: a mean instruction execution rate, and the time required to transfer control of the computer module from one job to another. Loading of memory by a computer module can be deduced from the mean instruction execution rate and the memory module characteristics.

Memory modules are characterized by size and cycle time. It is not clear at this time whether it would also be necessary to distinguish between cycle time and read access time. Paging, allocation memory times, and the like would be charged to the computer module instruction execution rate.

No quantitative characterization of the i/o modules would be required. It would be assumed that each module would be capable of driving any peripheral device or portion of bulk memory on a one-device-at-a-time basis at any rate up to the limitation imposed by memory cycle time and that each module would tie up its associated memory for only the time required to exchange information one word at a time.

Modeling the work load would be expected to be much more difficult than modeling the hardware. It must be emphasized that each of the parameters detailed below must be described by a probability distribution rather than a deterministic value. To maintain this emphasis, the random deviates will be underlined.

Each type of job in the many types characterizing the workload has an associated arrival rate and arrival time. Each job has associated with it a preprocessing load and a bulk memory occupancy. Each job has an intrinsic priority which, it may be recalled, was associated with the time-varying marginal cost of delay in initiation. Once initiated, the job is characterized by its core occupancy, number of parallel paths, rates and nature of macro calls referring to external objects, and i/o requirements, as well as the total number of instructions to be executed. It must be emphasized that the statistics of these parameters may not be stationary but may also vary as execution progresses. After initiation a job has, in principle, a calculable running priority based on its interactions with other jobs, but in the case of real-time jobs and time-sharing jobs approximating real-time criteria, there may also be an associated intrinsic running priority.

Finally, in addition to evaluating all the costs associated with delays and attempting to minimize them, either on a total basis or perhaps with some constraints on the maximum delay costs that may be accrued on any job, the counterbalancing "profits" represented by progress on background problem(s) must be evaluated.

The literature is not rich in this type of analysis. The best example may be found in the considerations of Aoki, Estring, and Mandell (3). They considered the case of two computers assigned to perform a bivariate interpolation in which each computer was required to await the completion of the previous computation cycle of the other computer before proceeding. It may be worthwhile to quote extensively from Kleinrock's review (17) of this paper:

"The principal result . . . shows that for a distribution of computation time which is uniform over some interval the performance depends directly upon the ratio

σ/m of the distribution's standard deviation to its mean. This is a rather interesting and simple result; it must be recognized that it is applicable only to the specialized models of parallel computation considered in the paper. The performance measure R is defined as the ratio of expected computation time for the two-processor system to the expected time for a single-processor system. . . . Their results show that R is bounded from below by $1/2$ and that this lower bound may be achieved when $\sigma/m = 0$ Indeed, for such systems it may be shown that $1/2 \leq R \leq 1$ since at worst one of the two processors is always idle (giving $R = 1$), and at best both processors are always busy (giving $R = 1/2$)."

Another example of analytic investigation is given in (9).

It is necessary to determine the extent to which the results of queueing theory are applicable to this problem. Indeed, Kleinrock (17) suggested in his review that "It would be interesting to consider the analogies between the models presented in this paper and certain models of queueing theory, in particular, cyclic queueing models."

One of the most elementary examples of queueing theory, so elementary that Takács (32) quotes it in the introduction of his book, is the result of the "bus stop" problem.

"Suppose that at a given stop buses are arriving in accordance with a homogeneous recurrent process. The inter-arrival times have the distribution function $F(x)$ and β is the average inter-arrival time. What is the average waiting time of a passenger arriving at time t at the stop?

"The answer usually given is $\beta/2$. But this is so only if buses run at exactly β -time intervals. The average waiting time depends on the variance of the inter-arrival times. The right answer is

$$\frac{\beta}{2} + \frac{\sigma_{\beta}^2}{2\beta},$$

where σ_{β}^2 is the variance of the inter-arrival time."

The analogy between this result and Aoki's

$$R = \frac{1}{2} + K \frac{\sigma}{m}$$

is suggestive of the applicability of queueing theory to the multiprocessor problem.

On the other hand, we must note the caution given in the Aoki (3) paper that

". . . past analyses dealt mostly with queueing problems with a single server. Aside from a rather straightforward extension of queueing analyses with multiple (and perhaps heterogeneous) servers, systems with multiple processing units have produced an entirely new kind of problem in which the processors work interdependently."

Two investigators have followed this trail with some success. Fife and Rosenberg (11) have used a queueing model to predict the response delay characteristics of a time-shared system. Weingarten (35) used a queueing model to analyze memory loading in a store-and-forward message-switching system. It must be noted that the queueing models yield delays to various requesters but do not in themselves suggest how to minimize costs which are nonlinear, time-variant functions of these delays.

Monte-Carlo Approach

The relatively low yield of analytically derived results in comparison to the labor expended raises the possibility of using Monte-Carlo techniques programmed on a digital computer. It should be realized that the difficult model-making job is not ameliorated by this approach; only the actual calculation of performance effectiveness (be it Aoki's "R" or the total accrued delay costs suggested earlier) is simplified.

There are two compelling objections to the Monte-Carlo approach. The first is that any result applies only to the parameters selected for the run, and considerable experimentation would be required to identify critical parameters and their values which yield optimum performance. Analytic results, on the other hand, are amenable to conventional maximization techniques. The other objection concerns the labor involved in programming the simulated operating system. Stone (31) has estimated that 40 man-years are required to develop a modern multiprocessor operating system. While most of this would be devoted to i/o control packages, memory allocators, etc., which would not have to be simulated, a substantial portion is attributable to the scheduling algorithms, which are central to the proposed investigation.

Fortunately, it is not necessary to decide at this time whether Monte-Carlo techniques will be employed as the first stage of the investigation — modeling the hardware and the problem mix would be substantially the same under either approach.

SPECIFIC PLANS

The previous discussion has outlined goals and problem areas in the study of multiprocessor operating systems and has suggested a number of possible approaches. It now remains to present the specific plans for the research contemplated. These plans have been based on the desire to obtain some results of general interest while remaining within the scope of a thesis investigation.*

This presentation will be generally limited to the analytic phase of the investigation. At any point where the analytic difficulties become too formidable, the choice will have to be made between abandoning that particular line of attack or continuing with simulation and Monte-Carlo analysis.

The general goal is to devise means of evaluating the effectiveness of a hardware-software combination on a problem mix. From this, and from estimates of the marginal cost of adding hardware or software capabilities, the optimum combination for a given problem mix can be deduced. Such a study implies that assumptions will have to be made about (a) a measure of system effectiveness, (b) the nature and statistics of the problem mix, (c) the hardware options available and their relative cost, (d) the levels of software capability available and their relative cost, and (e) reasonable software strategies available within each capability level. It will be seen that there are interactions between the assumptions and that not all combinations of options are meaningful. Let us therefore start with an assumption concerning the nature of the problem mix and explore some possible associated measures and capability levels.

Assume an infinite well of jobs of zero priority, i.e., no costs associated with delays in the initiation or completion of the jobs, and that the possibility that a given job may never be completed carries no penalty. Under this assumption a reasonable definition of productivity is: processor time devoted to the execution of problems as distinguished from executions of operating system routines or idle processor time. One difficulty with

*See footnote on page 1.

this definition is that it makes no allowance for input or output volume. Consider two jobs requiring equal amounts of processor time where only one produces a substantial amount of output. The output-limited job requires more system resources than the other, not only in i/o equipment and channels but also in memory occupancy. If processor time is the only measure of productivity, no credit has been given for the extra resource utilization of the output-limited job.

One may escape this dilemma by assuming that the jobs are drawn at random from a statistically homogeneous population, with the specific characteristics of a job unveiled during its solution. Since the operating system would not be permitted to preselect the jobs, it would be proper, in an average sense, to give equal credit to each completed job even though their solution difficulty would be quite unequal. Under this argument the mean solution rate would be a just measure of productivity. Care must be taken, however, to assure that the unveiling of parameters does not occur too early in each problem, or else the operating system would be tempted to abandon difficult problems early and spuriously appear to raise productivity by concentrating on easy jobs. For this reason some search for safer measures of productivity would seem worthwhile.

The important parameters of each job are: memory requirements, computational requirements, and i/o requirements. Two models for generating these parameters will now be described. The first represents as simple a model as would be useful to analyze, and the second represents as complex a model as it seems practical to analyze. In both models the total memory size M is a parameter of the analysis. It may be necessary to terminate the analysis at some level of complexity intermediate to the two extremes.

Simplest Model: All jobs have a memory requirement consisting of three components, m , x , and y . Component m , which is assumed to be the same for all jobs, represents driver programs and data which must be present for the entire job from initiation to completion, irrespective of whether the job is actually running. Each job consists of the same number n of segments. In order for a segment to be run, additional memory $x+y$ must be temporarily allocated. Component x , which represents additional program and scratch, is assumed constant for all segments of all jobs, but component y , which represents space for output, is a random variable. Each segment requires one unit of time of one processor for execution, after which memory space x can be released. Output is presumed to occur at rate r ; thus y must be retained for an additional y/r units of time. The memory requirements of an allocated job are, therefore, $m+x+y$ for one unit of time while the segment is being processed, $m+y$ for y/r units of time during output, and m for a variable time until the next segment is initiated.

Complex Model: Jobs are characterized by the same parameters, but m , x , and n become random variables. The value of n for a specific job is not known until job completion; i.e., at the end of the i th segment there is some probability of another segment being required. This probability may be a function of i but is otherwise constant over the jobs. At the end of each segment there may be a delay in commencing output, since the number of output channels n_0 will be limited. The operating system will be permitted to start the execution of the $(i+1)$ th segment before output of the i th is complete provided that memory is simultaneously allocated for y_i , x_{i+1} , and y_{i+1} .

Obviously, the results will depend on the form of statistical distribution chosen. It is proposed to use exponential distributions initially but to consider other distributions as well.

Still retaining our assumptions concerning problem-mix nature and measure of effectiveness, we must now explore operating systems which can be characterized by both

their mechanical capabilities and their strategic principles. By mechanical capabilities are meant features that intuitively seem desirable, but which are not always implemented in operating systems, to deal with resource requirement conflicts. By strategic principles are meant the rules by which the operating system decides which of these capabilities to employ.

The levels of mechanical capability that would be considered are:

1. In case of memory conflict, requester is terminated, the space previously occupied is freed, and all past work on requester is wasted.

2. In case of memory conflict, requester is suspended, the memory space occupied just before the request must continue to be allocated to requester, and an attempt will be made to grant the conflicting request each time some other program releases memory space.

3. In case of memory conflict, either the requesting program or some other suspended program can be rolled out to bulk storage (subject to channel availability delays at a rate r') and can be brought back at a later time.

4. At output time the operating system has the option (subject to channel availability delays) of outputting at the device rate r or to bulk storage at a rate r' , intending at some later time to reallocate memory space y_i , to input from bulk storage at rate r' , and to output to the device at rate r .

In defense of the assumed range of mechanical capabilities it may be stated that: the AOSP, as originally delivered, functioned substantially at level 1 (although running programs rather than the requester might be terminated in accordance with job priority), was modified as a result of NRL experience to level 2, and may have the features of levels 3 and 4 added in the future. These latter capabilities have been proposed for the operating systems of most new computers.

The spectrum of strategies to be considered varies with the assumed level of mechanical capability. At level 1 the only decision required of the operating system is when to start programs. Two simple strategies seem worth investigating: to attempt to keep the number of jobs in core constant, and to keep the amount of core allocated constant, i.e., to load a new job whenever core usage falls below a certain level. At the higher capability levels various algorithms will have to be devised and investigated to determine when roll-out and roll-back should be performed, what should be rolled out, and how the output queue on bulk storage should be handled.

The object of these analyses will be to develop expressions for productivity as a function of problem statistics, hardware complement, operating system capabilities, and operating system strategies. From these expressions the optimum strategy and associated productivity can be found for a given set of problem statistics, hardware complement, and operating system capabilities. This suggests two further calculations: assuming a given cost relationship between different types of system modules, it should be possible to find the optimum module mix and the associated operating system strategy and productivity for a given combination of problem statistics and operating system capabilities; furthermore, for given problem statistics it should be possible to express the value of an increment in operating system capabilities either in terms of increased productivity or in terms of the reduction in hardware complement possible while maintaining constant productivity.

It should be recalled that the discussion thus far has pertained to the simple model of an infinite well of jobs of zero priority. A contrasting model would assume a problem arrival rate sufficiently low that the system would be capable of solving all of them. Under this assumption a reasonable goal for the operating system is to minimize total delay

costs. These costs are, in general, a function of the time delay between the arrival of a problem in the system and the completion of its solution.

The simplest delay cost function would be linear in time and identical for each job; hence the simple sum of all delays would be the parameter to be minimized. It is not known whether more complex delay cost functions could be considered without resorting to simulation. It should be observed that because of the nonobjective standards a system user is likely to apply to his work, more complex functions could not be equitably applied to a job-shop problem mix, although realistic nonlinearities could be devised for certain numerical control applications.

Under the new assumption regarding the nature of the problem mix and the resultant new measure of productivity, we must now reexamine our assumptions concerning problem-mix statistics and the operating system capabilities and strategies.

The "simplest model" can probably be used unchanged, but the "complex model" would require some modifications. The principle that values of statistically distributed parameters are unveiled as late as possible was required to allow the measure of effectiveness to be meaningful, where the measure of effectiveness was applicable to the old assumption about problem mix. Under the new assumptions a better model might be --

Complex Model: Jobs are characterized by n , m , x , and y as before. The system is characterized by M , n_0 , and r as before. However, the values of n and m are declared in advance for each job. Furthermore, x and y , while random variables, are drawn from distributions declared in advance for each job.

The range of operating system capabilities listed earlier for the other problem-mix and associated productivity measure could be used unchanged, but more complex operating system strategies would have to be devised and evaluated to take advantage of the information available about the resource requirements of each job.

Still another variation would be to assume given statistics of jobs and their arrival rate and to find the minimum cost hardware complement that yields a given total delay cost at various levels of operating system capabilities. As in the first model, the value of various levels of operating system capability could be expressed in terms of possible reductions in hardware complement that maintain the given total delay cost.

Throughout the investigation attention will be directed to the determination of the sensitivity of solutions to the form and parameters of the distribution of statistical variables. Practical application of the work envisaged would depend on the identification of parameters which can be reasonably well estimated for real-life problem mixes and which determine optimum strategies and hardware complements.

CONCLUSIONS

It is concluded that the multiprogrammed multiprocessor has an important future for both conventional computing loads and for time-shared applications. The hardware problems in the construction of such multiprocessing systems have been solved, and the mechanical software problems of building workable operating systems have been solved. Additional problems in the area of memory allocation and protection are on the brink of solution. There is, however, a broad, relatively uninvestigated area in the optimization of hardware complements and operating system strategies that deserves analytic attention.

The principle problems include: job-loading strategies, job-suspension, -termination, and -roll-out strategies, the value of roll-out schemes and bulk-memory-buffered i/o, and the optimum hardware mix for a given problem mix.

While the analytic difficulties appear formidable, and models yielding useful results are expected to be more restrictive than desired, investigation of these and related problems appears overdue and potentially fruitful.

BIBLIOGRAPHY

Referenced in Text

1. Anderson, J.P., "The Burroughs D825," *Datamation* 10:30-34 (Apr. 1964)
2. Anderson, J.P., et al., "D825 - A Multiple-Computer System for Command and Control," *Proc. AFIPS 1962 FJCC*, pp. 86-96
3. Aoki, M., Estring, G., and Mandell, R., "Analysis of Computing-Load Assignment in a Multi-Processor Computer," *Proc. AFIPS 1963 FJCC*, pp. 147-160
4. Chapin, G., "Organizing and Programming a Shipboard Real-Time Computer System," *Proc. AFIPS 1963 FJCC*, pp. 127-138
5. Clippinger, R., "Multiprogramming on the Honeywell 800/1800," *Information Processing 1962, Proc. Intern. Federation Inform. Process., Amsterdam:North-Holland*, pp. 571-572, 1963
6. Coleur, J., (GE), private communication, 1964
7. Conway, M.E., "A Multiprocessor System Design," *Proc. AFIPS 1963 FJCC*, pp. 139-146
8. Corbato, F.J., et al., "The Compatible Time-Sharing System; A Programmer's Guide," Cambridge, Mass.:MIT Press, 1963
9. Delgavis, I., and Davison, G., "Storage Requirements for a Data Exchange," *IBM Systems J.* 3:2-13 (1964)
10. Dow, J., "Programming a Duplex Computer System," *CACM* 4:507-513 (Nov. 1961)
11. Fife, D., and Rosenberg, R., "Queueing in a Memory-Shared Computer," *Proc. 19th ACM Natl. Conf.*, Aug. 1964
12. General Electric Company, "GE-635 Comprehensive Operating Supervisor," GE Manual CPB-1002, July 1964
13. Gosden, J.A., "The Operations Control Center Multi-Computer Operating System," *Proc. 19th ACM Natl. Conf.*, 1964
14. Gregory, J., and McReynolds, R., "The SOLOMON Computer," *IEEE Trans. Electron. Computers* EC-12:774-781 (Dec. 1963)
15. Holt, A.W., "Program Organization and Record Keeping for Dynamic Storage Allocation," *Information Processing 1962, Proc. Intern. Federation Inform. Process., Amsterdam:North-Holland*, pp. 539-544, 1963
16. Keister, W., Ketchledge, R.W., and Vaughan, H.E., "No. 1 ESS: System Organization and Objectives," *BSTJ* 43:1831-1844, Sept. 1964

17. Kleinrock, L., "R64-34," IEEE Trans. Electron. Computers EC-13:320 (June 1964)
18. Lyons, R.E., and Vanderkulk, W., "The Use of Triple-Modular Redundancy to Improve Computer Reliability," IBM J. Res. Develop. 6:200-209 (Apr. 1962)
19. Mace, M., (IBM), private communication, 1964
20. McGovern, P.J., "GE Compatibles - 600," Computers and Automation 13:26-29 (Aug. 1964)
21. McKenney, J.L., "Simultaneous Processing of Jobs on an Electronic Computer," Management Sci. 8:344-354 (Apr. 1962)
22. Mills, M.R., "Operational Experience of Time Sharing and Parallel Processing," Computer J. 6:28-36 (Apr. 1963)
23. Perkins, R., and McGee, W.C., "Programmed Control of Multi-Computer Systems," Information Processing 1962, Proc. Intern. Federation Inform. Process., Amsterdam:North-Holland, pp. 545-549, 1963
24. Porter, R.E., "The RW-400 - A New Polymorphic Data System," Datamation 6:8-14 (Jan.-Feb. 1960)
25. Ryle, B.L., "Multiple Programming Data Processing," CACM 4:99-101 (Feb. 1961)
26. Schwartz, J., Coffman, E., and Weissman, C., "A General Purpose Time-Sharing System," Proc. AFIPS 1964 FJCC, pp. 397-441
27. Shaw, J.C., "Joss: A Designer's View of an Experimental On-Line Computing System," Proc. AFIPS 1964 FJCC, pp. 455-464
28. Slotnick, D., Borck, W., and McReynolds, R., "The Solomon Computer," Proc. AFIPS 1962 FJCC, pp. 97-107
29. Sperry Rand Corporation, private communication, 1964
30. Spratt, S., (SCD), private communication, 1963
31. Stone, I., (CDC), private communication, 1964
32. Takács, L., "Introduction to the Theory of Queues," New York:Oxford University Press, 1962
33. Thompson, R., and Wilkinson, J., "The D825 Automatic Operating and Scheduling Program," Proc. AFIPS 1963 SJCC, pp. 139-146
34. Thornton, J.E., "Considerations in Computer Design," Computers and Automation 12:22-26, 59 (Nov. 1963); 12:22, 24-25 (Dec. 1963); 13:22-26 (Jan. 1964)
35. Weingarten, A., "Storage Requirements for a Message Switching Computer," IEEE Trans. Commun. Systems CS-12:191-195 (June 1964)
36. Wilkinson, J., (Burroughs Corp.), private communication, 1961
37. Zurcher, F., (Burroughs Corp.), private communication, 1964

Unreferenced in Text

38. Amdahl, G.M., "New Concepts in Computing System Design," Proc. IRE 50:1073-1077 (May 1962)
39. Bauer, W.F., "Why Multi-Computers?," Datamation 8:51-55 (Sept. 1962)
40. Boilen, S., et al., "A Time-Sharing Debugging System for a Small Computer," Proc. AFIPS 1963 SJCC, pp. 51-57
41. Critchlow, A., "Generalized Multiprocessing and Multiprogramming Systems," Proc. AFIPS 1963 FJCC, pp. 107-126
42. Critchlow, A.J., "Multiprogramming and Multiprocessing," IEEE Spectrum 1:192-198 (Mar. 1964)
43. Curtin, W.A., "Multiple Computer Systems," Advan. in Computers 4:245-303, New York:Academic Press, 1963
44. Dennis, J., "Rev. No. 5698," Computing Reviews 5:162 (May 1964)
45. Estrin, G., et al., "Parallel Processing in a Restructurable Computer System," IEEE Trans. Electron. Computers EC-12:747-755 (Dec. 1963)
46. Flores, I., "Derivation of a Waiting-Time Factor for a Multiple-Bank Memory," JACM 11:265-282 (July 1964)
47. Howarth, D., Jones, P., and Wyld, M., "The Atlas Scheduling System," Proc. AFIPS 1963 SJCC, pp. 59-67
48. Lehman, M., Netter, Z., and Eshed, R., "SABRAC, A Time-Sharing Low-Cost Computer," CACM 6:427-429 (Aug. 1963)
49. Lonergan, W., and King, P., "Design of the B5000 System," Datamation 7:28-32 (May 1961)
50. Marcotty, M., Longstaff, F., and Williams, A., "Time Sharing on the Ferranti-Packard FP6000 Computer System," Proc. 1963 SJCC, pp. 29-40
51. Miller, W.F., and Aschenbrenner, R., "The GUS Multicomputer System," IEEE Trans. Electron. Computers EC-12:671-676 (Dec. 1963)
52. Penny, J.P., and Pearcey, T., "Use of Multiprogramming in the Design of a Low Cost Digital Computer," CACM 5:473-476 (Sept. 1962)
53. Rosenberg, A.M., "Computer-Usage Accounting for Generalized Time-Sharing Systems," CACM 7:304-308 (May 1964)
54. Teoste, R., "Design of a Repairable Redundant Computer," IEEE Trans. Electron. Computers EC-11:643-649 (Oct. 1962)
55. Weil, J.W., "A Heuristic for Page Turning in a Multiprogrammed Computer," CACM 5:480-481 (Sept. 1962)

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Naval Research Laboratory Washington, D.C. 20390		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP	
3. REPORT TITLE MULTIPROCESSOR OPERATING SYSTEMS			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) An interim report on a continuing problem.			
5. AUTHOR(S) (First name, middle initial, last name) Bruce Wald			
6. REPORT DATE April 11, 1967		7a. TOTAL NO. OF PAGES 28	7b. NO. OF REFS 55
8a. CONTRACT OR GRANT NO. NRL Problem R06-07		9a. ORIGINATOR'S REPORT NUMBER(S) NRL Report 6531	
b. PROJECT NO. Project RF-001-08-41-4552		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
c.			
d.			
10. DISTRIBUTION STATEMENT Distribution of this document is unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Department of the Navy (Office of Naval Research), Washington, D.C. 20360	
13. ABSTRACT <p>The history and present status (1965) of multiprocessing, multiprogramming, and timesharing are reviewed. It is concluded that, despite their diverse histories, these techniques are destined to be intertwined. Although the mechanical problems in operating systems that exploit these techniques have largely been solved and the difficult memory allocation problem is on the brink of solution, the important question of optimum operating system strategy in initiating, suspending, and terminating jobs is largely unexplored. Suggestions are made concerning models which might be suitable for both analytic and Monte-Carlo approaches to the optimization of operating system strategy and to the selection of optimum hardware mixes. An extensive bibliography is included.</p>			

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Multiprocessing Multiprogramming Timesharing Scheduling Operating systems Computer						