BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

THE HOSPITAL COMPUTER PROJECT
TIME-SHARING EXECUTIVE SYSTEM

Report Number 1673
April 1968

CAMBRIDGE NEW YORK CHICAGO LOS ANGELES

THE HOSPITAL COMPUTER PROJECT
TIME-SHARING EXECUTIVE SYSTEM

Medical Information Technology Department
Bolt Beranek and Newman Incorporated
Cambridge, Massachusetts 02138

Report Number 1673
April 1968

The work described in this document received support through a contract, PH43-62-850, from the Institute of General Medical Sciences, National Institutes of Health, and through a grant from the American Hospital Association.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The technical developments described in this report reflect mainly the contributions made by Bolt Beranek and Newman Inc. (BBN) to a joint research effort performed in collaboration with The Massachusetts General Hospital (MGH) during the period 1962-1968.

The Hospital Computer Project was initiated by Jordan J. Baruch, who organized and directed a staff of about 30 persons to perform BBN's part of the effort. Continuing direction at BBN during the past two years has been provided by Paul A. Castleman and Frank E. Heart. The Project's Time-Sharing Executive System was designed and implemented by Sheldon Boilen, Steven R. Weiss, Charles R. Morgan, Bernard P. Cosell, Jonathan G. Cole, and Andrew P. Munster . This report was written by Alexander A. McKenzie.

That part of the effort contributed by the Hospital was supervised first by E. Michael White, Assistant Director of the MGH , and subsequently by G. Octo Barnett, M.D., Director of the Laboratory of Computer Science of MGH. This essential hospital-centered contribution to the combined research effort was enabled by the backing and encouragement given by The Massachusetts General Hospital through the office of its director, John H. Knowles, M.D.

INTRODUCTION

The Hospital Computer System is viewed in several different ways by the various groups of people who use it. To the hospital staff, it is an information-handling system which interacts with them in a question and answer form of English dialogue. Application programmers at Bolt Beranek and Newman Inc. (BBN) view the system as a time-shared computer with a programming language and debugging facilities. The BBN systems programmer sees a collection of hardware elements (central processor, drums, tapes, Teletypes, etc.) which must be converted into a time-shared computer facility.

This report describes the time-shared computer facility which the systems programmers have created. The report is directed primarily toward those readers who are investigating the concepts necessary for construction of a time-sharing system. With this point of view in mind, machine-dependent descriptions have been avoided when they are not essential; hence, this is not a maintenance manual for the BBN system. On the other hand, it is assumed that the reader is familiar with the terminology and techniques of programming and time-sharing in general.

A scale of over-all complexity can be constructed for the classification of present time-sharing systems. At one end of this scale are systems such as the SABRE airline reservation system; users of SABRE interact with a small set of rigidly defined programs which perform prespecified functions of information storage and retrieval in a massive but simple data base. Users of this type of system are actually sharing an application function. At the other end of this scale are systems like MIT's Project MAC

in which the users actually share the computer hardware. These systems are capable of handling a wide variety of tasks, many large independent data bases, automatic assignment of almost unlimited memory to any task requiring it, and perhaps a multiplicity of central processor units as well.

Although the Hospital Computer System is not as large as Project MAC, it is conceptually much closer to that end of the scale than it is to systems like SABRE. It includes both special-purpose functions designed for use by hospital personnel and general-purpose functions which are available to modify and extend special-purpose functions. "Automatic" memory assignment includes only one segment of 4096 (4K) 18-bit words. There is only one language, a macro-assembly language, available to applications programmers. The data structure is oriented toward the particular storage and retrieval problems of the hospital. On the other hand, many users may be performing unrelated tasks at the same time. They can access one of several data bases; these data bases may be privately owned and confidential or they may be public and accessible by many users simultaneously. Thus, while the hospital computer system is directed toward a single goal, servicing a hospital, it permits a wide variety of different tasks to be performed simultaneously. Systems programmers, applications programmers, and hospital personnel are generally all working with the system at any given time.

The time-sharing operating system (Executive System) described in this report is the third hospital time-sharing operating system developed at BBN. It was first put into operation in May 1966 and has been in service operation since December of that year. In its first 15 months of operation, the system was scheduled

for full use by The Massachusetts General Hospital an average of 100 hours per week and had an over-all up-time average of 97 percent.

This report is divided into five sections. The first describes the hardware environment of the Executive System. No attempt has been made in Section I to specify the machine completely; only those elements that were designed specifically for time-sharing operation have been treated at length. The other sections describe the software routines that comprise the Executive System.

The second section describes the multiprogramming software, the "Swapper," in great detail. It is felt that the detail contained in this Section will enable others to learn from the problems encountered during this project.

The third section describes the method of linkage from user programs to user-oriented subroutines in Executive core memory. This linkage is accomplished through a routine, activated by privileged-instruction trap logic, which dispatches the privileged instruction to the appropriate subroutine — and therefore is called the "Dispatcher."

The fourth and fifth sections describe the various system I-O facilities. The fourth section covers the bulk-storage I-O routines, collectively known as the "I-O Processor." These routines must deal with the problem of interleaving many users' requests for a single device. The fifth section describes the I-O routines for devices such as the Teletypes and paper-tape reader/punch which are used by only one user at a time. Magnetic-tape I-O, however, is described in the fourth section because of the close

interrelationship between the magnetic-tape and bulk-storage-drum hardware.

I. THE HARDWARE ENVIRONMENT

The computer chosen as the basis for the Hospital Computer System was a PDP-1, manufactured by the Digital Equipment Corporation. The PDP-1 is a 5-microsecond, 18-bit computer. Each instruction occupies one word, allowing 12 bits of address information. The basic memory module consists of 4096 (4K) words. In order to meet the demands of time-sharing, this computer has been extensively modified. The cost of the modifications can reasonably be used as a measure of their scope; central-processor modifications alone cost as much as the purchase price of the unmodified PDP-1. The total cost of the system hardware is about one million dollars.

In addition to the central processor unit (CPU), the system includes two other processorlike devices, the program-swapping drum and the Data Channel. Both of these devices have the ability to make direct memory accesses without the use of CPU circuitry. Thus, three independent processes may be occurring simultaneously; this multiprocessing capability greatly increases the efficiency of the time-sharing system.

The program-swapping drum is divided into thirty-two 4K-word fields. The drum is capable of exchanging 4K of core storage for 4K of drum storage in 35 milliseconds. That is, during one revolution of the drum, a 4K core-memory module can be written onto one drum field and another drum field can simultaneously be written into the same memory module. This type of swapping technique has been more or less central to most time-sharing systems which permit a variety of tasks with limited memory.

The Data Channel is a high-speed I-O device used for transfers to and from bulk storage. Once it has been activated by an I-O command from the central processor, this device has the ability to make direct memory references for data transfers *or for additional I-O commands*. This latter capability allows the Data Channel to perform logically-complex I-O operations under program control without interrupting the CPU. Attached to the Data Channel is a Fastrand bulk-storage drum which provides approximately 60-million 6-bit characters of storage for random-access file-handling (and overflow from the swapping drum) at a 1.1-megacycle-per-second bit-transfer rate. In addition, two magnetic-tape units are attached to the Data Channel; they provide long-term bulk storage and a certain amount of system backup. The bulk drum, the tape units, and the controllers for these devices were acquired from the Univac Division of Sperry Rand Corporation; they also designed and built the Data Channel to BBN specifications.

As previously described, the PDP-1 computer is supplied with 12-bit addressing and a 4K memory module. A standard option allows special "extended addressing" (utilizing indirect addressing) to obtain 16-bit addresses; thus, the computer can address a maximum of 64K. Normally, all memory references are made through one memory buffer register and one memory address register. The swapping drum, because of its high transfer rate, requires exclusive use of these registers during its read/write time of 35 milliseconds, making multiprocessing impossible during each program swap. This situation is intolerable not only because of decreased system capacity but also because of the possible loss of incoming information owing to the inability of the memory to receive it.

To meet the problem of the swapping drum's monopolizing the system, an independent memory scheme was established to divide the *potential* 64K of memory into four 16K *banks*, each composed of four 4K *modules*. The actual memory configuration on the Hospital Computer System is one bank of 16K words (4 modules) for the Executive System and two banks each containing 4K words (1 module) for the user programs. Each of the banks has its own independent memory buffer register and memory address register. Logically, memory is treated very much like a piece of I-O gear; that is, a processor makes a request to memory for a piece of data or transmits a piece of data to memory for storage. The memory control also has priority logic to permit more than one processor to make access to the same memory bank, in the order of the immediacy of the processor's requirements. The swapping drum has the highest priority and therefore can interrupt either of the other two processors (although in actual operation the swapping drum should never need to access a bank being used by another processor). The Data Channel has priority over the central processor.

With this independent memory scheme, one user program may run in one of the 4K banks while another user program is being swapped into the other 4K bank. Hence, swapping and computation may occur simultaneously. Similarly, a user program in one of these banks may be doing a bulk-storage I-O operation (via the Data Channel) while a second program is being swapped into the other 4K bank (see Fig. 1). In any case, however, the central processor always has access to the Executive memory bank and the interrupt-handling routines which reside there. This implies that by allocating some of Executive core memory to small buffers for the Teletype communications lines and other non Data Channel

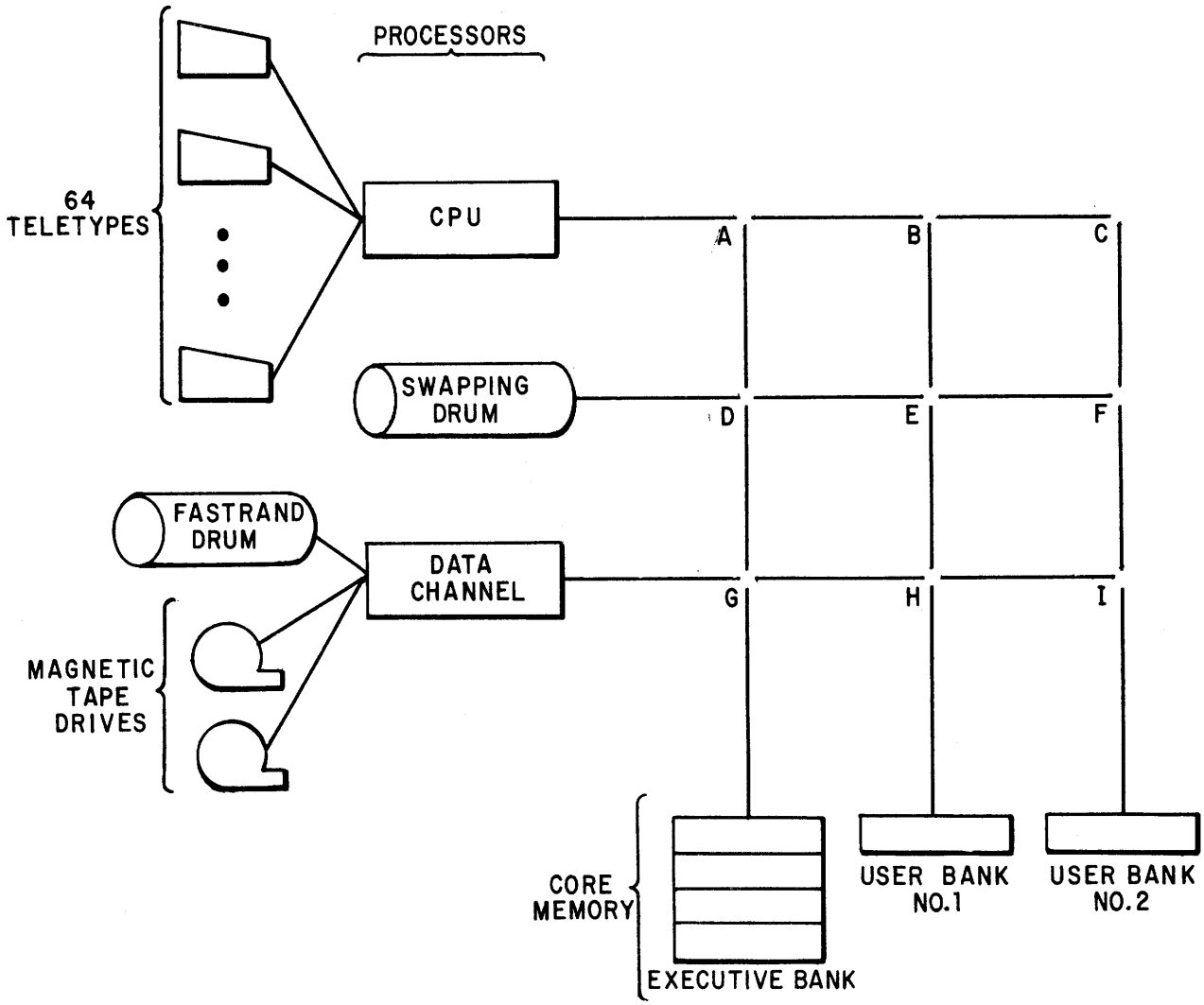

FIGURE 1. Interconnection of processors and memory banks.

Each of the points on the Figure, labeled A through I, is a possible processor/memory connection. For example, at some time connections might be made at points B and F; in this case, the program in User Memory Bank No. 1 is running while another program is being swapped into User Memory Bank No. 2. At another time, connections might be made at points A, E, and I; this is the case where the routines in Executive Memory are handling interrupts, the program in User Memory Bank No. 2 is performing a bulk-storage I-O operation via the Data Channel, and another program is being swapped into User Memory Bank No. 1.

I-O devices (paper-tape reader and punch, line printer, etc.) the system can handle these devices at maximum transmission rates without special hardware buffering.

Because it could not be known in advance in which bank the user would be running, it was necessary to add four 2-bit rename registers, which are essentially relocation registers or page registers. These rename registers work in the following manner. If the computer makes a memory reference to bank 0, then the contents of rename register 0 are substituted for the bank address and, hence, the memory reference will go to the physical bank specified by the contents of rename register 0. This mode of operation eliminates the need for multiple-level indirect addressing and gives all the relocation facilities that are needed at present.

Another important set of modifications to the original PDP-1 are the interrelated hardware components which provide for an interrupt (or sequence-break) system, privileged-instruction trap, and memory protection. The Hospital Computer System includes a 16-level interrupt system with a wired-in priority schedule. When an interrupt is received, the hardware determines if any higher-priority interrupt is already being serviced and, if not, honors the new interrupt. If a higher-priority interrupt is being serviced, the new interrupt is stored; it is honored as soon as it becomes the highest-priority interrupt requesting service. Interrupts may be initiated by such mechanical activities as the positioning of the paper-tape reader or a completion pulse from a paper-tape punch or console typewriter, by electrical activities such as the ticks of the 32-millisecond clock and the 1-minute clock, by signals from other processorlike devices such as the data channel or the swapping drum, by the instruction-trap logic,

or by interrupt commands coded in the Executive.

When an interrupt is honored, the live registers (e.g., the accumulator) associated with the interrupted program are automatically saved in Executive memory at locations unique to the priority level at which the interrupt occurred. Thus, even if an interrupt-handling routine is itself interrupted, there is no loss of information. Since these registers are stored in memory, they can be altered by the Executive routines if desired. At the conclusion of processing an interrupt, the Executive routine (which was started by the interrupt) executes a "debreak" instruction, signaling the interrupt hardware that lower-priority interrupts may be honored.

The assignment of priorities carries with it no connotation of *importance* but rather the connotation of *immediacy*, as determined by the characteristics of the device producing the interrupt. For example, if the Data Channel has been looking for a particular word on the rotating drum and suddenly finds it, its signal to the central processor for attention must be handled immediately lest a complete rotation of the drum be required before the information can actually be transferred. On the other hand, there is no immediacy in the start of the search for a word and, hence, initiation of such a search can take place on a low-priority interrupt level. (Although the priority organization may not allow an interrupt to be honored when it is received, the request will be saved and honored when it becomes the highest-priority interrupt which is requesting service.) Based on this concept of immediacy, the activities assigned to the various priority levels are listed in descending order of priority in Table I. It should be mentioned here that all Executive routines are

started by interrupts received by the priority-interrupt hardware.

Linked to the interrupt system are the hardware components which protect the computer system from user-issued "privileged" in-

TABLE I. Interrupt priorities.

Priority (Octal)	Source of Interrupt	Report Section Describing Routine
Ø	unused (*highest priority*)	
1	high-speed data channel	I-O Processor
2	paper-tape reader	Teletype, etc.
3	line printer	Teletype, etc.
4	I-O controller commands	I-O Processor
5	program-swapping drum	Swapper
6	terminal scanner	Teletype, etc.
7	one-second clock	Dispatcher
1Ø	one-minute clock	Dispatcher
11	unused	
12	paper-tape punch	Teletype, etc.
13	unused	
14	console typewriter	Teletype, etc.
15	I-O processor program	I-O Processor
16	privileged-instruction trap	Dispatcher
17	32-millisecond clock	Swapper

structions or memory-bound violations. Naturally, users cannot be permitted to execute instructions which would halt the machine or directly affect the I-O operations; similarly, a user cannot be permitted to transfer data into any memory bank other than the one in which he is currently running. On the other hand, the routines in the Executive memory bank must be able to use I-O instructions and must have access to all memory locations. Since the interrupts initiate only Executive routines, it was convenient to have the interrupt system "enable" and "disable" the privileged-instruction and memory-protection hardware. When an interrupt is honored, the protection hardware is turned off, and all instructions and all memory references are legal. As soon as debreaking occurs, the protection hardware is turned on and the machine runs in "user mode."

The privileged instructions, which a user cannot directly execute, include

I-O instructions,

instructions which halt the computer,

instructions which affect the interrupt system,

instructions which affect special registers, e.g., the rename registers, and

instructions which refer to protected memory locations.

Instructions may make references to any address in the memory module in which the instructions themselves are located, or to any address in memory bank Ø (the running user's bank) except for registers Ø-37 in memory bank Ø.

The protected registers in the user bank are used by the Executive software to store information about the

> status of the user, and so they must not be altered except by the Executive. Permitting references to the user bank from other banks allows reentrant Executive subroutines to service the user, under his control.

If the hardware detects the use of a privileged instruction in user mode, it stores the instruction in a special trap register and generates an interrupt. This interrupt initiates an Executive routine called the Dispatcher (see Section III) which interprets the contents of the trap register and responds appropriately. Many privileged instructions are defined by the system as subroutine calls, allowing a user program to link to Executive subroutines.

The final important hardware modification was the addition of character-manipulation instructions and of a special operating mode called "ring mode." One of the functions that an Executive routine must control is the transmission of characters to and from the remote Teletypes. This function is performed by the Teletype Service Routine (see Section V). Since this routine runs frequently, it is important that it be very fast. For this reason, instructions that address a single 6-bit byte were added. (These instructions, of course, are useful to any character-handling program.) Optionally, these instructions are self-incrementing, increasing the byte pointer by one each time the instruction is executed. Thus, four successive executions of the "deposit character and index" instruction which was initialized with the address of the first (left) byte of the word *X* would deposit characters in the left, middle, and right bytes of word *X* and the left byte of word *X*+1.

Ring mode was added to facilitate circular buffering. When oper-

ating in ring mode, the central processor suppresses the carry between the third and fourth (from rightmost) bits if a self-indexing instruction is being executed. Thus, the Teletype Service Routine can store incoming or outgoing characters in an 8-word circular buffer and store (or fetch) the next character by using only a self-incrementing byte-reference instruction.

II. THE SWAPPER

The primary goal of a time-sharing system, as opposed to other methods of computer utilization, is to provide each of a number of users with *immediate* access to data and to processing facilities. Each user is given the illusion that he has complete control over the functions of the computer system. Time-sharing systems achieve this illusion by dividing the total available computing time into small discrete time units (time quanta) and distributing these time quanta among the system's users according to some predetermined queueing algorithm. Because of the extremely high speeds of computers, each user still feels that his job is being performed at computer speed. In the case of an interactive system, in which the user program is delayed by some slow I-O device (e.g., Teletype) or by the time spent by the user in thinking about his problem, a given user program requests time quanta for computation relatively infrequently. Thus, a program engaged in computation is normally in competition for system resources not with *all* other users but with only a small percentage of the user population.

Some time-sharing systems utilize a "paging" concept which allows some portion of many user programs to be present in core memory at the same time. The system decides when a new portion (page) of a user program is required in core, allocates a space for the new page, and retrieves it from some bulk-storage device. The Hospital Computer System permits only one user program to be in core memory (for computation) at any one time; the space available is 4K words and the user may perform his own program segmentation. When the time quantum allocated to a user program has elapsed, or when the program becomes "hung" waiting for an external action

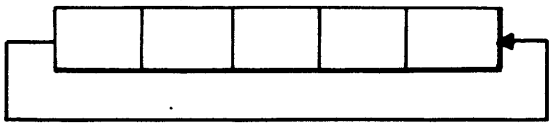

FIGURE 2. Round-robin queue.

(e.g., I-O operation), the entire 4K of user core memory is written out on the swapping drum and some other user program is read into core. This operation is known as "swapping" and it is controlled by the Executive routine called the Swapper. The Swapper administrates the queueing algorithm, making all decisions about when to perform a swap and which user program should be permitted to run next.

One simple queueing algorithm is the "round-robin" method. This algorithm uses a circular queue of users: the user at the top of the queue is given one time quantum and is then placed at the bottom of the queue (see Fig. 2).

The round-robin algorithm, however, fails to account for several problems which occur in actual practice. It does not allow for the fact that before a quantum of time has passed the running user may be "hung" waiting for something to happen. It also ignores the problem of a user losing incoming data because the interval between his time quanta is too long (i.e., there are too many users).

Another disadvantage of the round-robin method is that it makes no attempt to match the amount of time allocated to a user to the past history of his requirements. This is an important consideration, both from the point of view of satisfying the greatest number of users and from the point of view of minimizing the number of swaps performed. A reasonable assumption is that an actual

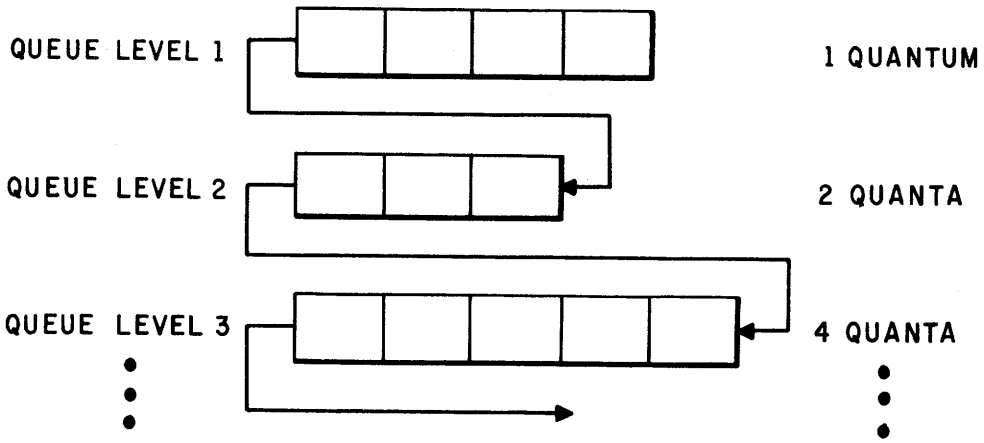

FIGURE 3. Multilevel queue.

user's tolerance to system delay in answering a question is something like an exponential function of the actual computing time needed to answer the question—i.e., that the user prefers a system that delays answering complicated questions in return for quick response to simple questions. Thus, a queueing algorithm which was cognizant of the user programs' past history would be able to favor the small computer-time requests (whose users were less tolerant of slow response) at the expense of penalizing those user programs which demanded a great deal of computer processing time.

One answer to some of the objections to the round-robin method is a *multilevel queue*, which has been implemented on this system. In this system, there are 12 queues into which a user can be placed. A user in the first (highest) queue is given one quantum of computing time; the time allotted to a user in any other queue is twice the time allotted to the user in the next higher queue. Thus, a user in queue n will receive 2^{n-1} quanta of time. If a program in queue n uses all the time allowed in that queue, then the program is placed in queue $n+1$ (see Fig. 3). On the other

hand, all programs in queue 1 are allocated their time before any other program is given time; similarly, programs in queue 2 are all allocated time quanta before programs in queues 3-12, and so on. Programs which spend a large amount of time computing move lower and lower in the queue structure; at each downward jump, they receive more time quanta, but wait longer to get it. On the other hand, demands for interaction with the "outside world" cause the program to be placed in a high queue; for example, if a user's Teletype input buffer is almost full, his program will be placed in queue 1.

In addition to considering the queue position of a program, the Swapper also considers two other parameters, *location* and *status*. Program status is a device for temporarily disregarding the queue structure; classification by location allows the Swapper to determine which programs may be accessed without delay.

There are four possible locations for programs in the system: in one of the two user cores, on one of the 32 swapping-drum fields, on the swapping area of the Fastrand drum, and "in limbo." The in-limbo location applies to programs which have been requested, either by a Teletype user or by another program, but have not yet been started. (These programs do not need to be swapped into core but will be read in by a special Executive "startup" routine.)

Program location is used to determine whether programs are "accessible" or "inaccessible." Programs stored on the swapping drum are accessible if the swapping drum is not being used. Programs on the Fastrand are accessible when the Fastrand is not already being used for swapping. In-limbo programs are accessible when

(1) there is an empty user core, or

(2) the swapping drum is not busy and has a free field, or

(3) the Fastrand is not being used for a swap and has a free field.

Programs which are already in core are not considered to be either accessible or inaccessible.

Programs are separated into three distinct groups on the basis of status. Programs which need computing (central processor) time are in *runnable* status. For these programs, the queues are sufficient to determine allocation of time. Some programs, however, are waiting for the completion of a noncomputational event; the event may be input or output to a Teletype or other peripheral device, use of the I-O Processor (IOP), the passage of a predetermined amount of time, etc. These programs are in *hung* status but will return to runnable status as soon as the necessary event is completed. The third status, called *wanted*, includes programs which require special attention because

(1) the user has typed the function BREAK at his Teletype, demanding suspension of his program, or

(2) the program is wanted for examination by the Executive debugging system, or

(3) the program has been hung, waiting for its turn to use the IOP and the IOP is now ready for it.

Finally, on the basis of queue level and location, the Swapper

performs an ordering of the urgency-of-service requirements of the programs in the system, finding a "best" (most-urgent requirements) and a "worst" (least-urgent requirements) program. The ordering is determined according to the following rules.

1. Programs in wanted status are better than programs in runnable status; programs in runnable status are better than programs in hung status.

2. Programs which are runnable are ordered according to their queue: queue 1 is best; queue 12 is worst.

3. Programs within a given queue are ordered according to the number of time quanta that they have already received in that queue: the more time a program has already received, the better it is. This rule approximates a "first-in first-out" philosophy, but allows for the fact that several programs within a single queue may each have received some computing time in that queue. (As an example, suppose that program A, in queue 5, computes for two time quanta and then becomes hung. Program B, also in queue 5, then computes for three time quanta and becomes hung. An ordering at this time would make program B better than program A.)

The job of the Swapper can now be defined as finding the best accessible program, and if this program is *better* than the worst program in core, swapping them. As used here the word *swap* means reading one program into memory from a drum storage device and writing another program from the same area of memory onto that device. These two operations may be performed simultaneously when the swapping drum is used. The term *one-direction swap* is defined to mean either one of these operations, but not both. If

there are no accessible programs, or if the worst program in core is as good as the best accessible program, then

(1) if the best program in core is not hung, it will be run;

(2) if the best program in core is hung, the swapper will wait for a change in the status or location of some program. Whenever an interrupt routine recognizes a change in a user program's queue level, status, or location, it notifies the Swapper, which then begins a new evaluation.

If the Swapper has given control to a program in core, its job is obviously completed. If, on the other hand, the Swapper initiates a swap, the Swapper routine is immediately restarted from the beginning for another evaluation. In this case, the set of accessible programs is smaller (because of the set of programs made inaccessible by the initiation of the swap) so that a series of evaluations will always result in either running a program located in a user core or waiting for a change in some program's status or location (e.g., the completion of a swap).

In addition to the organizational philosophy described above, a number of special techniques were adopted to improve the performance of the system. Some are merely coding tricks used to save core space or processing time; others are modifications of the philosophy to handle specific problems. The rest of this Section describes the four most important of these techniques.

A. Queue Counter

The rules for finding a "best" program are, first, to find all

the programs in the highest occupied queue and, then, to pick the one which has already been given the most time. Each program has a queue counter organized in such a way as to make both these checks possible in one operation. For a program in queue n, the counter C_n is defined by the equation

$$C_n = \text{queue number} + \text{number of unused quanta} - 1.$$

The queue number Q_n of a program in queue n is defined as

$$Q_n = 2^{n-1}.$$

Recall that a program in queue n will receive 2^{n-1} time quanta. Therefore, the number of unused quanta A_n in queue n is

$$2^{n-1} \geq A_n \geq 1.$$

By simple operations, this inequality can be converted to the inequality

$$2^n - 1 \geq Q_n + A_n - 1 \geq 2^{n-1}$$

or

$$Q_{n+1} - 1 \geq C_n \geq Q_n .$$

There are two programming advantages to this scheme. First, a running program can be charged for a time quantum merely by subtracting one from its queue counter. Then the program has just changed queue if

$$[(C_n) \text{ AND } (C_n - 1)] = 0 .$$

(This test must be made to determine whether the Swapper should

search for a new best program.)

The second advantage to this queue-counter scheme is that the best runnable program is the runnable program with the lowest queue counter. Similarly, the worst runnable program is the one with the highest queue counter. Up to this point, the queues have been described as though they were separate lists. This is not necessary, and in fact the allocation of sufficient space to store 12 different queue lists would be quite inefficient. Both space and search time can be saved by assigning each program a fixed location in a queue-counter table. The Swapper then searches the table for the lowest (or highest) queue counter when it is looking for the best (or worst) user.

The time quantum used in this system is 32 milliseconds, the approximate amount of time required for a swap on the swapping drum. A running program, of course, may be interrupted by some outside event, which causes it to be swapped out after only a fraction of this time has passed. Therefore, each program contains a clock register which gives the number of milliseconds that it has used which have not been charged to its queue counter. This register is maintained by the Swapper, and the program's queue counter is decremented only when this clock register shows that a full quantum has been used.

B. Priority

If two programs which contain equal and large amounts of computing time are started at the same time, they will each fall to lower queues at about the same rate and can be expected to com-

plete their calculations at about the same time. This is not always desirable, since certain hospital services should be completed faster than program-development computations. For example, the compilation of a laboratory report should take precedence over a systems programmer's assembly. Accordingly, the Swapper recognizes a priority system (not to be confused with the interrupt priority system) which allows certain programs to fall to lower queues more slowly than normal. Programs with a priority of zero (or one) are handled exactly as described above. If an application program, by direction of the system administrator, is assigned a priority P ($2 \leq P \leq 31$), then its queue counter will be decremented by 1 for every P quanta of running time. Thus, for a priority of 14, 14 time quanta must be given to the program before its queue counter is decremented by 1. It is also possible for a program to be assigned a negative priority; in this case, the program's queue counter is decremented by P for every single time quantum which it receives.

Any priority system can be abused, and the system described here is no exception. The user, however, must use the Executive system to change his program's priority, and the Executive could have been written to require special passwords in order to allow priority change. Instead, it was decided that the highest priority used by a program should be included in the system statistics that are kept on every user run. These statistics are available to the system administrator, and with them he can minimize the misuse of priority.

C. Fastrand Swapping

Ordinarily, active nonrunning programs are stored on the swapping

drum (or "little drum"). This little drum is small and fast; it can store thirty-two 4K programs and complete either a one-direction or two-direction swap in 35 milliseconds. In order to allow for more than 32 active programs, a portion of the Fastrand (or "big drum") has been allocated to the Swapper; this portion has the capacity to store 49 additional programs. The big drum, however, is much slower. The amount of time required for a one-direction swap can be computed as follows.

45 milliseconds	— average head positioning
34 milliseconds	— average rotational latency
87 milliseconds	— data transfer
166 milliseconds	— average one-direction swap

A two-direction swap requires an average of 287 milliseconds since the heads do not need to be repositioned. Even in the rare case where latency time is zero, a one-direction big-drum swap takes almost three times as long as a little-drum swap.

Big-drum swapping is actually performed by the I-O Processor. The parameters of the I-O operation are determined by the Swapper, but the program to be swapped out must actually execute the instructions which link it to the I-O Processor. (These instructions are located in Executive core, and the instruction counter of the program is set to their location by the Swapper.) In order to allow the program to execute these instructions, its queue counter is temporarily set to the best possible value by the Swapper; the queue counter is reset to its correct value after the swap is performed.

Because of the length of time required for a big-drum swap, the

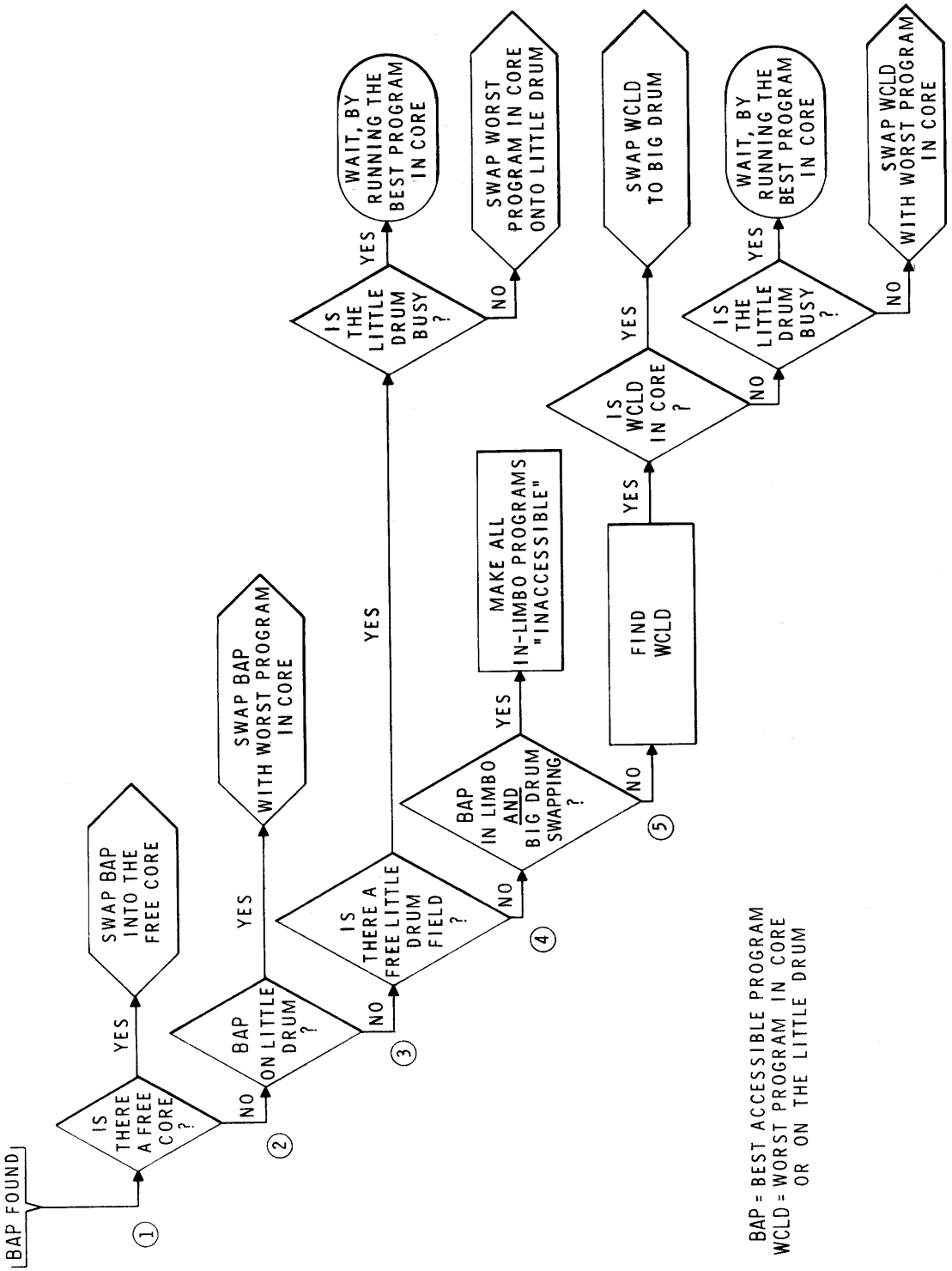

FIGURE 4. Swapper flow diagram.

Swapper makes several tests to ensure that programs are not unnecessarily placed on the big drum and that, when a program is swapped onto the big drum, it is the worst in the system. The simplified explanation of these test is given below in the order that the tests are made; these steps are also keyed to the accompanying block diagram (Fig. 4). Note that in each case other than running a user program the Swapper is then restarted to perform another evaluation.

1. If there is an unused core, the best accessible program (BAP) is swapped in and nothing is swapped out.

2. If the BAP is on the little drum, then the worst program in core is swapped with the BAP.

3. The BAP is now known to be in limbo or on the big drum; the Swapper will look for an unused field on the little drum. If there is an unused field, the worst user in core will be swapped onto that field before bringing in the BAP. This procedure may require the Swapper to wait for the little drum to complete some other operation, but, even if this is so, the program will be swapped out more rapidly than if it were swapped onto the big drum (and it can be retrieved much more rapidly).

4. At this point, the little drum is known to have no unused fields, so the only way to create space in core for the BAP is to swap some program onto the big drum. If the BAP is in limbo and the Swapper has already assigned some program to be swapped on the big drum, then all programs in limbo will be considered inaccessible and the Swapper will be restarted to find a new BAP. To understand the reason for this procedure, recall how a big-drum swap is initiated. The program to be swapped out of core is started in Executive core and executes instructions to link it to the I-O Processor. In order for this program to be started, it must be the BAP, and it won't be the BAP if the programs in limbo are ac-

cessible. On the other hand, it must be started (in order to free a core for the program in limbo). Therefore, failure to make the programs in limbo inaccessible would result in an endless loop within the Swapper. A similar problem does not arise for programs on the big drum, since they are automatically inaccessible if the big drum is needed for a previously authorized swap.

5. If the big drum is not already engaged in a swapping operation, or if the BAP is already on a big-drum swapping field, the Swapper searches for the most suitable program to swap onto the big drum. Although the worst program in core could be swapped out immediately, it is probably in core because it is better than most of the programs stored on the little drum; therefore, it will probably soon be the best program in the system and should not be placed on the big drum. Consequently, the Swapper examines the status and queue level of programs both on the little drum and in core in order to find the worst program in core or on the little drum (WCLD). An important feature of this search is that it is initialized to start at a different point on the little drum each time that it is performed; therefore, any completely inactive program on the little drum will be moved to the big drum in a maximum of 31 big-drum swaps. It should also be noted that programs hung because they need access to the big drum for I-O operations are not considered when searching for the WCLD, in order to prevent them from appearing twice in the I-O Processor's queue structure (see Section IV).

Now the location of the WCLD is determined. If this program is in core, the Swapper alters its control registers so that the next time that it is started it will link to the I-O Processor for big-drum swapping. On the other hand, if it is on the little drum and the little drum is free, then it is swapped with the worst program in core in preparation for big-drum swapping. If the little drum is busy, then the Swapper waits for the little drum to complete its current operation by running the best user in core.

D. System Overload

Occasionally, the system will be requested to create space for a new program when the system is already operating at maximum capacity. If the request comes from a user Teletype, the system responds by merely typing CALL LATER PLEASE at that Teletype. However, the request may come from some already running program which cannot be delayed until space is available for the requested program. In order to avoid loss of information in this situation, descriptions of all requested programs are stored in a special list on the big drum. Each time that a program is added to this list, an in-limbo entry is placed in the Swapper's tables; the in-limbo program is actually a routine in Executive core which starts the first program in the list and creates a new in-limbo entry if there are further programs in the list. If running the in-limbo program would exceed system capacity, the system alerts the machine operators and "throws away" the in-limbo program (by deleting its entry in the Swapper's tables in core) but retains the list on the big drum. At any future time, the programs in the list may be started either by the machine operator or by the addition of a new program to the list.

III. THE DISPATCHER

As pointed out in Section I, there is a class of privileged instructions which user programs must not be allowed to perform because of their possible detrimental effect on the system; these include input-output instructions and instructions that halt the machine. If a user program attempts to execute a privileged instruction, the instruction is trapped by computer hardware and stored in the trap register, and an interrupt is initiated. The user programs, of course, must be allowed to perform I-O operations or halt by some mechanism, so the routine started by the privileged-instruction interrupt was given the function of examining the trap register and dispatching to the portion of the Executive which can perform the desired operation for the user. This routine is therefore known as the Dispatcher. Since the operations which the earliest version of the Dispatcher performed for the user were primarily Input-Output Transfer operations, the privileged instructions which were specified to cause such a linkage were known as IOT's.

The present Hospital Computer System uses a greatly expanded IOT concept. From the point of view of the user programs, IOT's are the same as machine-language instructions. In general, they are more complex than machine commands, may require somewhat longer calling sequences, and perform those operations normally associated with closed subroutines. Most of the IOT's fall into one of the three following categories:

1. operations which the user programs must not be allowed to perform for themselves —— this category includes all I-O operations and references to the Executive memory bank;

2. operations which are common to many user programs, especially programs which implement hospital applications —— this category includes routines for storage and retrieval of individual items in tree-structured hospital records, conversion of numbers in date or decimal format to internal code, and floating-point arithmetic;

3. linkage to two system "interpreter" routines, called Job Hunter and Syntax Verifier, which are used by all hospital-applications programs. Job Hunter is a questionnaire administrator that provides standard methods for formatting questions, correcting erroneous answers, and branching around questions made unnecessary by previous answers. Syntax Verifier provides a standard method of specifying the syntax of possible user input and determining which of a number of possible user inputs (including invalid input) was actually given.

A block of user core memory (registers 40_8-72_8) is reserved for temporary storage for the IOT routines. (These registers are not protected by hardware, as are registers 0_8-37_8, but protection is unnecessary since user-written code is not executed during the execution of an IOT.) With all temporary storage used by IOT's allocated to the user memory block, it is clear that the IOT's can be written as reenterable subroutines; thus a user program may be swapped out during the execution of an IOT. In actual practice, IOT's which have high execution speeds are executed out of user mode on interrupt level 16 (i.e., before debreaking), thereby preventing initiation of the Swapper which is on interrupt level 17 and, therefore, of lower priority. Slower IOT's, in particular Job Hunter and Syntax Verifier, are executed in user mode (i.e., after debreaking) so that swapping of programs using these IOT's is very likely to occur.

The execution of an IOT is performed according to the following steps.

A. The privileged instruction (IOT) in the user program is encountered by the central processor and trapped by the hardware. An interrupt is generated and the hardware registers associated with the user program, including the program counter (which specifies the next instruction to be executed in the user program), are stored in Executive memory.

B. The Dispatcher, which is started by the interrupt, reads the trap register and decodes the IOT by means of a table lookup on seven bits of the trap register. This lookup yields the address of the Executive routine required, and a branch is made to that address.

C. Each routine decodes an additional six bits of the trap register to determine which of various options may have been requested for the basic IOT. If the IOT has a high execution speed, it is executed and the routine debreaks to the user program. In cases where the IOT will be executed in user mode, the user registers stored by the interrupt are transferred from Executive memory to the user-memory area reserved for IOT's and the debreak address is changed to the address (in Executive memory) of the appropriate IOT routine. Debreaking then occurs, starting the IOT routine in user mode instead of starting the user program.

Notice that if a user program which is using a slow IOT is swapped out, the location, saved by the Swapper, at which the program will be restarted later is a location in the IOT routine in Executive memory. Also, the eventual point of return to the user program is stored in, and therefore swapped out with, the user memory block. Any other program which is swapped in between time periods allotted to this user program may use any IOT since no user-program information has been saved in the Executive memory. Finally, the slow IOT routines may themselves use fast IOT's (for example, Job Hunter uses Teletype IOT's) without any loss of information;

in fact, the slow IOT routines return to the user program by means of a special "return" IOT that moves the user registers from user memory back to the special interrupt locations and then debreaks.

IOT's communicate with the user programs in several ways. Many IOT routines transfer information through two special hardware registers accessible to all programs, the "Accumulator" and the "I-O register." In addition, some IOT's cause information to be transferred to user-program buffer areas; these areas are frequently defined by a prespecified block of pointers immediately following the privileged instruction in the user program. Other IOT's indicate the success or failure of an attempted operation by returning to the user program at one of a series of prespecified points, normally the first, second, third, or fourth location following the privileged instruction. The IOT to "reserve the paper-tape punch," for example, will return to the first location after the privileged instruction if the punch is already reserved by some other program, or to the second location if the punch has been successfully reserved.

A few IOT's are designed to delay execution of the user program for a given length of time or to restart a user program at a specific time. These time-dependent functions require the IOT routines involved to have some knowledge of the passage of time in the real world. For this reason, there are two clocks which interact with these routines, a one-second clock and a one-minute clock. The one-second clock is used to measure delay times and the one-minute clock allows the Executive to keep track of the time of day.

Some privileged instructions, of course, will be errors in the

user programs. Some of these will appear to the Dispatcher to be valid IOT's but execution of them in the controlled environment of the Executive will at worst damage the user program. Others will be recognized as errors; in this case, the Dispatcher will transfer the special interrupt registers associated with the user program to user memory (in the IOT area) and write the user program on the Fastrand drum for later examination by the programmer. The Dispatcher then terminates the program by branching to the normal "halt" IOT routine.

IV. THE I-O PROCESSOR

The I-O Processor (IOP) is that portion of the Executive system that handles the manipulation of data on the Fastrand drum and on magnetic tape. The Fastrand system and the magnetic-tape system are logically independent from the user's point of view; their interaction within the IOP is dealt with briefly at the end of this Section.

The magnetic-tape equipment, which is discussed at the end of this Section, includes two tape drives and a tape controller capable of spacing the tape forward or backward by record, file, or reel and of transferring information between the Data Channel and either tape drive.

The Fastrand drum provides storage for approximately 19.7 million words. A set of movable heads can be positioned over any one of 96 tracks, each of which is subdivided into 4,096 50-word sectors. Each sector, in addition to a storage capacity of 50 data words, contains a tag word which can be used for "linking" information. A sector is the smallest addressable unit of drum storage. The drum hardware also includes a drum controller to control movement of the boom, on which the read/write heads are mounted, and to transfer information between the Data Channel and the drum.

Conceptually, the drum is divided into thirds, each having 32 track positions. The allocation of various types of data to specific thirds is by convention as follows.

Third	Use
Ø	Active Patient Record Files
1	Program Library and Programmer's Files
2	Research Files

Each third is divided into 128 quarter-tracks; each quarter-track is composed of 1024 sectors. A quarter-track can be in one of three conditions: available (i.e., empty and unowned), free, or held. Data such as permanent file structures, libraries, and permanent programmers' files are stored as free information. Free quarter-tracks may be written on (or expunged from) by any user program (provided that it presents the appropriate own-word). A held quarter-track is owned by one user program; only that program can write on, change, or expunge from it, although any program can read it. Held quarter-tracks are used for temporary storage——that is, for program segments, scratch areas, and temporary file storage. When a user program halts, its held quarter-tracks are returned to available status. If a user program expunges all information on a held or free quarter-track, that quarter-track is returned to available status.

The I-O Processor automatically assigns quarter-tracks to held or free use, depending on the type of "write" IOT which is executed. The I-O Processor keeps track of the locations of available sectors within each quarter-track. If a user program writes "nonaddressed," it leaves it to the I-O Processor to put the information into the next free sector or sectors; if it writes "addressed," it specifies the location of already existing information to be written over. If a user program writes "nonaddressed held," the I-O Processor puts the information into space on a quarter-

track held by that user program, assigning a quarter-track to held use if there is no more space available on the user program's currently held quarter-track(s).

There are two formats for data storage on the drum: the block and the item. A user program can write a fixed length of information, called a block, which consists of precisely 50 words and is stored in one drum sector. An item is variable length and offers a more flexible means for storing information on the drum. Its format in core is as shown in Fig. 5, with the "word count" equal to the number of words in the item. On the drum, the formats are allocated as shown in Fig. 6.

To prevent certain time-sharing problems and to protect the drum from accidental changing (rewriting) or expunging, two mechanisms have been included: the rewrite number and the own-word. The rewrite number follows the word count in each item (as is illustrated in the preceeding diagram). This number is incremented by one each time an item is rewritten (written addressed). When rewriting or expunging an item, the I-O Processor compares the rewrite number of the item in core with the corresponding number on the drum. If they are the same (i.e., no one else has rewritten this item since the current user program read it), the rewrite number is incremented by one and the item is rewritten. If the rewrite numbers are not the same (i.e., someone else has rewritten this item since the current user program read it), no writing takes place, and a "rewrite error" message is transmitted back to the user program. Thus the rewrite-number concept permits simultaneous updating of a file by two or more user programs, while preventing the overlapped updating of a particular item in the file.

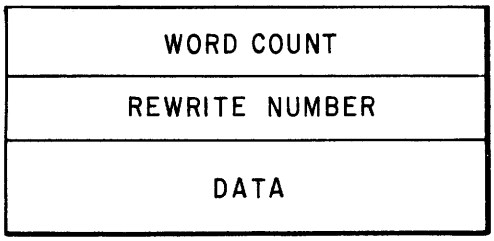

FIGURE 5. Item format in core memory.

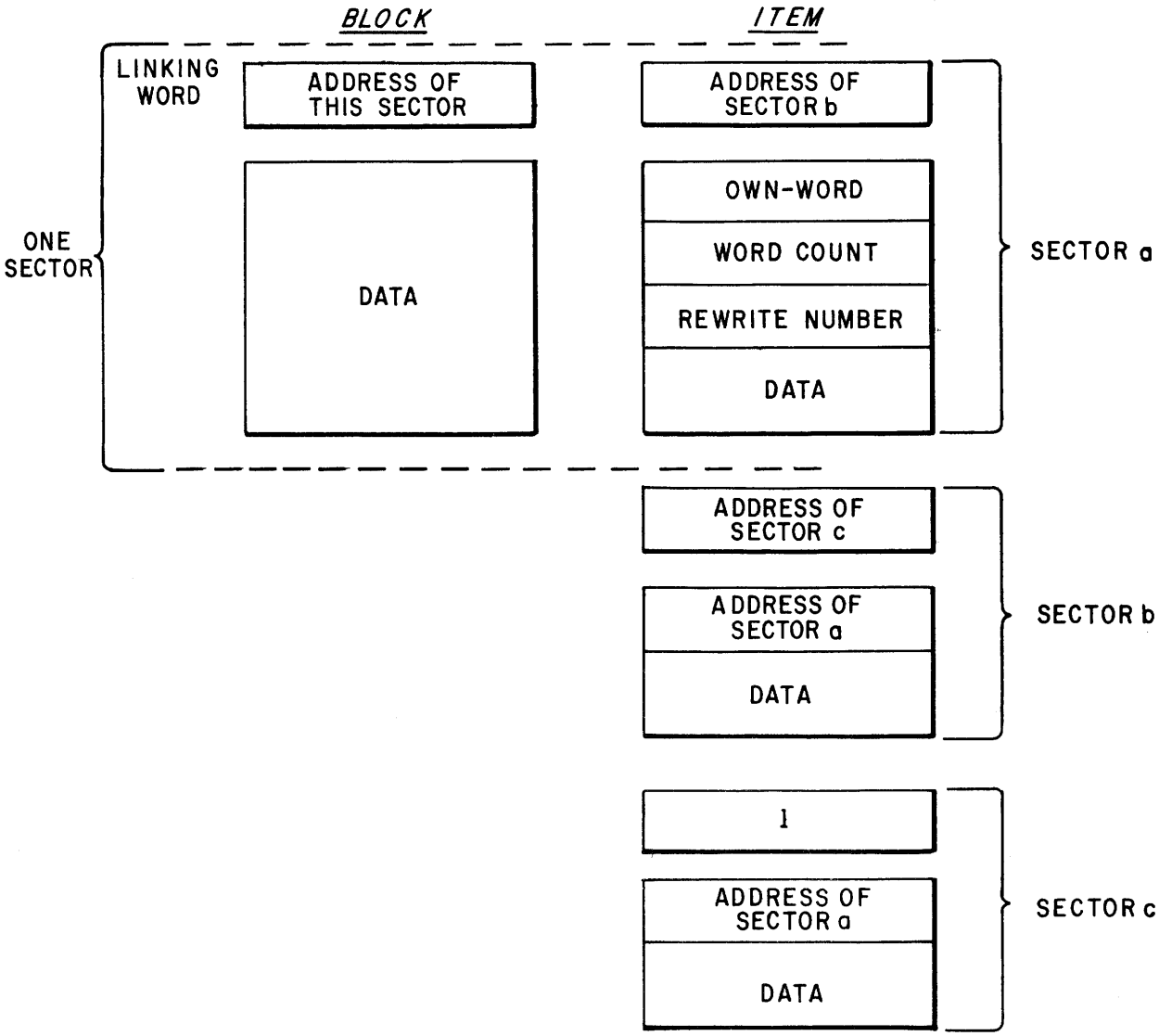

FIGURE 6. Block and item formats on Fastrand drum.

The own-word is another protective device. When an item is written nonaddressed, the user program specifies an own-word to be permanently associated with that item on the drum. Whenever a user program attempts to rewrite or expunge an item, it must present the word that matches the item's own-word. This method of identification helps to protect files from accidental destruction by programs under development.

In order to allocate storage space as it is requested, the I-O Processor must maintain information about the status of each quarter-track and about the status of sectors within each quarter-track. One possible method is to maintain all free space as threaded lists, each unused sector or quarter-track pointing to the next. Using this method, however, makes it almost impossible for the I-O Processor to perform any optimization of storage allocation, since no Executive routine could have access to a list of all unused quarter-tracks at one boom position without performing a prohibitive number of drum reads. For this reason, information about the status of each sector or quarter-track is instead maintained in tables in Executive memory. One table contains one word for each of the 384 quarter-tracks on the drum. The quarter-tracks are specified as free, held, or unusable (i.e., physically damaged), and empty, full, or partially full. The I-O Processor uses this table in an attempt to minimize future boom movement when assigning additional quarter-tracks to a user program.

The first two sectors of each quarter-track on the Fastrand are allocated for use by the I-O Processor. These two sectors, comprising only 0.2% of the storage capacity of the quarter-track, are used as an availability table for the rest of the sectors,

one bit per sector specifying either "used" or "available." The two sectors are read into one of four tables in the Executive memory when the quarter-track on which they are located is being modified. The I-O Processor uses the table in memory to locate available sectors, changes the table as additional sectors are used, and rewrites the table on the Fastrand when necessary (as described below). Each of four sector-availability tables in Executive memory ("Table 1" through "Table 4") contains the availability information for one quarter-track. Table 1 is assigned to the quarter-track being used for nonaddressed writes on third Ø. Tables 2 and 3 are used in a like manner for thirds 1 and 2. Table 4 is used for held writes and held or free rewrites on all thirds of the drum. Tables 1, 2, and 3 need to be written out only when the corresponding quarter-tracks are completely filled. Because the contents of Table 4 relate to quarter-tracks designated by the user programs, Table 4 is written back onto the drum and reset with information about another quarter-track very frequently.

Any of the Fastrand operations which a user program may perform (via IOT's) reaches the I-O Processor as one of four basic operations—— read, rewrite, nonaddressed write, or held write. Each of these operations requires certain preconditions to be satisfied before the desired information transfer can occur; the preconditions can frequently be satisfied by the I-O Processor while the user program is being swapped into core in "IOP wants" status. The preconditions for each basic operation are as follows.

A. *Read*

1. Have the boom positioned on the track specified by the user program.

B. *Rewrite*

1. Have the boom positioned on the track specified by the user program.

2. Have the availability table for the quarter-track specified by the user program in Table 4.

C. *Nonaddressed write*

1. Have the boom positioned on the track being used for nonaddressed writes on the third specified by the user program.

2. Have a list of the addresses of the necessary number of free sectors in the quarter-track to be used; this list is generated by the I-O Processor from Table 1, 2, or 3.

D. *Held write*

1. Have the boom positioned on the track specified by the user program.

2. Have the availability table for the quarter-track specified by the user program in Table 4.

3. Have a list of the addresses of the necessary number of free sectors in the quarter-track specified by the user program; this list is generated by the I-O Processor from Table 4 and used for nonaddressed held writes.

The I-O Processor contains subroutines to perform the actions needed to fulfill each of the three possible preconditions, as well as routines which handle special cases, such as the assignment of a held quarter-track to a user. Note that one precondition—namely, having the correct availability table in Table 4—may require drum writing and reading operations to be performed; in any case, if the boom is moved, the current contents of Table

4 may have to be written out (if it has been altered) before the move is performed.

The I-O Processor interacts with three different priority interrupt levels, in addition to IOT interpretation by the Dispatcher on interrupt level 16. All data transfers take place via the Data Channel, which is connected to interrupt level 1. The Fastrand-drum controller and the magnetic-tape controller are connected to interrupt level 4, and various table-manipulation routines are started by interrupt level 15. Driving the I-O Processor through several interrupt levels instead of one makes the I-O Processor more complicated but allows its functions which are not time-dependent to be interrupted by other system functions which are time-dependent.

Perhaps the easiest way to see the interrelationship of the various interrupt levels is to follow a user through the I-O Processor. When the user program's Fastrand IOT is trapped, the Dispatcher branches to one of a number of I-O Processor routines which interpret the IOT and construct two control words; these specify which of the four basic operations the user requires and the address of the desired quarter-track, if applicable. The control words, together with a pointer to the user program's queue counter (in the Swapper area of Executive memory), are then entered in an I-O Processor queue of user programs waiting for the Fastrand. The user program's status is set to "IOP hung" and the Swapper is alerted to conduct an evaluation. If the Fastrand is not being used, the IOT routine generates an interrupt on level 4 (by means of a coded instruction) before debreaking (see second following paragraph); usually the Fastrand is already in use and debreaking on interrupt level 16 occurs immediately.

When the current Fastrand-user's data transfer is completed, the Data Channel generates an interrupt on interrupt level 1. The I-O Processor charges the user program's queue level counter for the time spent doing the data transfer, sets the user program's status to "runnable," sets its program counter to the appropriate return from the Fastrand IOT, and notifies the Swapper that a user program's status has changed. The interrupt level 1 routine then generates a level 4 interrupt and debreaks from level 1

The interrupt on level 4, which is generated by code in either a level 16 or a level 1 routine, activates a portion of the I-O Processor which determines whether there are any programs waiting for the Data Channel. If there are no programs waiting, debreaking occurs. If at least one program is waiting, the level 4 routine generates a level 15 interrupt and debreaks.

The interrupt on level 15 activates a queue-sorting routine in the I-O Processor; this routine examines the Fastrand queue and finds the best waiting user program, i.e., the user program with the lowest queue level counter (the queue level counter is described in Section II of this report). The control words for this user program are moved to an I-O Processor communication area, the program's status is set to "IOP wants," a level 17 interrupt is generated to notify the Swapper, and a level 4 interrupt is generated.

The level 4 interrupt, generated on level 15, starts a routine which begins to satisfy the preconditions implied by the user program's control words. Most of the actual work must be done by subroutines which link the level 4 routine to routines on other levels; for example, Table 4 must be written out and

refilled on level 1 because this operation requires data transfers, and the list of block addresses, if needed, is made up by a level 15 routine. The level 4 routine is primarily a control routine; the only Fastrand activity performed on level 4 is boom movement, in the case when Table 4 does not need to be written out *and* the Data Channel is performing a magnetic-tape data transfer. At the same time, the Swapper is bringing the user program with "IOP wants" status into core, and, when this operation is completed, the Swapper generates an interrupt on level 1 to notify the I-O Processor.

Finally, when the user program is in core and the level 4 routine has satisfied as many preconditions as possible, the level 1 routine stops debreaking to lower interrupt levels and assumes control. Any remaining preconditions are satisfied (if a block address list must still be generated, it is generated by level 15 routines which return to level 1) and the data transfer is started. The routine on level 1 then debreaks and the cycle is complete.

The preceding discussion has taken very little account of the fact that two magnetic-tape drives are also controlled by the I-O Processor and must transfer information by means of the Data Channel. Available to user programs are IOT's which space magnetic tape forward and backward by record, forward by file, rewind tape, and reserve and release tape drives, as well as IOT's which read and write one record at a time. Reserving and releasing tape drives, of course, do not require the use of the I-O Processor, and, with the exception of reading and writing, all other magnetic-tape operations can be performed by the interrupt level 4 routines. The flow through the I-O Processor of user programs

waiting to use the magnetic tape is identical to the flow of those using the Fastrand drum. A queue of tape users is maintained by the level 15 and level 16 routines. Unlike boom movement, however, tape movement without information transfer is always performed on level 4.

It would, of course, be impermissible for either the tape users or the drum users to monopolize the Data Channel, and the order of priorities necessary to prevent this is administered by the level 4 routines as follows.

A. The "best" tape user ready for level 1 has priority over all other Data Channel users.

B. The "best" drum user ready for level 4 has priority of access to the level 4 routines.

Thus, even if there are user programs waiting for both the tapes and the drum, the drum users will be made ready for use of the Data Channel while the tape system is using it, and vice versa. The load on the tape system is normally quite light, and this alternation of Data Channel use has proved to be a satisfactory solution, although heavier tape use might necessitate a more complex scheme.

V. TELETYPE SERVICE ROUTINE AND OTHER "SLOW" I-O

In addition to the I-O Processor, the Executive contains a number of routines which drive "hard-copy" I-O devices. All of these devices are so slow that the user programs communicating with them must not be allowed to monopolize the user memory bank while the devices are sending or receiving messages. Devices which fall into this category, with their associated interrupt levels, are the following:

Device	Interrupt Level
Paper-tape reader	2
Line printer	3
Paper-tape punch	12
Console typewriter	14
Teletype-terminal scanner	6

Interrupts are generated each time that the device is ready to accept another character (output devices) or each time that a character is received at the computer (input devices). Input and output on these devices are controlled by two asynchronous processes. Information is moved between the user memory bank and buffers in Executive memory by means of IOT's activated by the user program. Information is transferred between the Executive memory buffers and the actual I-O device by the "service routines" activated by interrupts.

The Teletype service routine drives 64 devices rather than one, and each of the Teletypes is an interactive device at a remote location. With the exception of these special problems, however, all the I-O service routines are essentially the same.

Circular buffers for each of the slow I-O devices are maintained in Executive core. Buffer space is allocated according to the speeds of the devices and the number of bits required to specify a character; Table II summarizes the buffer allocation. Each I-O service routine has two functions:

1. sending I-O commands to the device when appropriate; and
2. notifying the Swapper that the user program should be run each time that the buffer is almost full (for input) or almost empty (for output).

Suppose, for example, that a user program has computed a set of values which it wishes to punch on paper tape. The user program first requests connection to the paper-tape punch by means of a two-return IOT; the first return is used if the punch is held by another program, otherwise the punch is assigned to the requesting user. When connection to the punch has been successfully completed, the user program sends characters (or 3-character binary words)

TABLE II. Buffer allocation for slow I-O devices.

Device	Data Rate (characters/sec)	Buffer Length (words)	Buffer Capacity (characters)
Paper-tape reader	400	128	256
Line printer	300	150	450
Paper-tape punch	63	32	32
Console typewriter	12	32	96
Teletype (1 of 64)	10	8	24

to the punch buffer one at a time by means of another IOT. This IOT includes special checks for buffer-empty and buffer-full conditions. If the buffer is empty when a character is sent to it, the IOT causes an interrupt on the paper-tape punch level which "awakens" the punch service routine. If the transmission of a character from the user program to the buffer causes the buffer to be filled, the IOT places the user program in "punch-hung" status and notifies the Swapper that an evaluation of user program status should be made.

The punch service routine, upon receipt of an interrupt, locates the next character in its buffer and transmits it to the punch. The routine then checks to see if the buffer is "almost empty," i.e., within a few characters of being empty. If the buffer is almost empty, the routine places the punch user in one of the highest queues, cancels the "punch-hung" status, and notifies the Swapper that an evaluation should be made. If the buffer is completely emptied, the routine merely debreaks; it will be restarted by the character-transfer IOT when the buffer-empty condition is discovered.

Finally, when the user program has completed transmission of its message, another IOT is used to relinquish control of the punch. Even if the user program omits this step, the "halt" IOT causes all devices held by the program which is halting to be released.

Input is handled in much the same way as output. For example, to read a message from paper tape the user program would first obtain control of the paper-tape reader and then request a character. This request would cause the reader to be activated by an interrupt and the user program placed in "reader-hung" status.

Reading would proceed, one character at a time, until the buffer became "almost full," at which time the reader service routine would place the reader user in one of the highest queues, remove the "reader-hung" status, and notify the Swapper that an evaluation should be made. If the reader buffer became completely full, the reader service routine would stop the reader until it was reactivated by an IOT.

As previously mentioned, the Teletypes pose several additional problems, related to the fact that they are both input and output devices and are used interactively. All Teletypes are interfaced with the computer through a terminal line scanner (or line concentrator). This scanner not only generates an interrupt when a character is transmitted but also indicates on which of the 64 Teletype lines the transmission took place. Thus, when activated by an interrupt, the first action of the Teletype service routine is to determine the line number of the line causing the interrupt.

A separate buffer is assigned to each line. When a buffer is almost full or almost empty, a table is consulted to determine which user program is associated with that buffer.

Programs in this system normally deal with 6-bit characters while the Teletypes use ASCII code (7 information bits, one parity bit). Rather than have each program perform the appropriate character translation, the Teletype service routine translates each character, as it is transmitted, by means of a table look-up. The 63 most common characters are coded as 6-bit characters; the 64th possible character is used as a "warning character." An additional 64 less commonly used characters are coded in

12 bits, the first 6 of which constitute the warning character. Thus, the user programs may use all ASCII characters, but alphanumeric information and common symbols require only six bits of storage.

Some additional problems of Teletype use require a brief description of the actual hardware involved. The connection between the computer and any single Teletype is a loop of wire, beginning and ending in the computer room. This loop can be thought of as directional, passing through various pieces of hardware in the order described below. The equipment is capable of sending or receiving 10 characters per second; each character consists of 11 bits. A bit is represented by the presence or absence of a current through the wire for 1/110 second. Presence of a current represents a one; absence of current represents a zero. The equipment arrayed along the transmission wire includes the following.

A. Computer Room (send)

1. A continuous source of one's (that is, a current source).
2. A computer send station——equipment capable of temporarily breaking the current flow to send zero's under control of the computer.

B. Teletype

1. A Teletype send station, operating under control of the keyboard, functionally identical to the computer send station.
2. A Teletype receive station, which interprets the signals on the line and delivers the characters that they represent to the Teletype printing and carriage control mechanisms.

C. Computer Room (receive)

A computer receiver which interprets the signals on the line and delivers the characters to the scanner. The scanner sends an interrupt to the computer when a complete character has been received.

Of the 11 bits per character, only eight are used for the ASCII code. These are preceded by an "activation" bit (always zero) and followed by two "deactivation" bits (always one's). The activation bit notifies the two receivers that a character is arriving; thus, an inactive Teletype will cause no "character received" interrupts since no activation bits will be present. The deactivation bits are needed to synchronize the Teletype receiver with the computer.

Note that a character transmitted by the computer send station will be received both at the Teletype and at the computer receive station, and will cause a "character received" interrupt in the same way as a character originating at the Teletype send station. This is called the "echo" character and can be used, if desired, to verify the correct operation of the transmission line. The arrival of the echo character also indicates that the line is free for transmission of another character by the Teletype service routine.

Each Teletype is equipped with a BREAK key (ASCII, "NULL"), which works differently from the other Teletype keys. Depressing the BREAK key breaks the transmission wire; that is, it sends a continuous stream of zero bits as long as it is depressed. The use of the BREAK key is intended to signal to the Executive that the user wishes to terminate current operations and start some-

thing else. Notice, however, that before the system can do anything for the user it must determine if he has closed the line (taken his finger off the BREAK key) so that other transmission is possible. Note that the user may break during computer-to-Teletype operation, as well as at other times.

The Teletype service routine checks all received characters (including echo characters) for the presence of a break, which is received as eight zero bits. As soon as a break is detected, some other character ("λ") is sent to the Teletype. If the next echo character received is "λ," then the line has been closed; otherwise, the process is repeated. When the line has been closed, the user program is notified that a break has been received.

The character chosen for "λ" is of particular importance. It should be a nonprinting character, if possible, in order to eliminate possible user confusion. More important, recall that the deactivation bits exist for the purpose of synchronizing the Teletype with the computer. Since the deactivation bits are not transmitted during a break, it is possible (in fact, probable) that the Teletype will lose synchronization, so "λ" should contain as many one bits as possible to reestablish synchronization. The character chosen, therefore, is "RUBOUT," which is nonprinting and consists of 8 one bits.

It is common for a Teletype to be unplugged from the transmission line, thus generating a "permanent" break. This situation causes the reception of 10 meaningless characters each second, each of which must be processed as a separate interrupt. To avoid wasting time on lines which are unplugged, the Teletype service routines count the number of consecutive break characters received

on each line. After 64 consecutive break characters (6.4 seconds), the line is considered unplugged and additional interrupts are ignored, except to check for the character "ALT MODE." If a Teletype is eventually plugged into the line again, the user may regain service by typing the "ALT MODE" character.