ABSTRACT

JANUS: A REALTIME TIMESHARING COMPUTER SYSTEM
FOR USE IN NUCLEAR PHYSICS EXPERIMENTS

By

John Oscar Kopf

A computer program called JANUS has been developed for use on a Scientific Data Systems Sigma Seven computer. JANUS is a computer operating system, designed to permit many different users to share the resources of the computer, such that each user is apparently in sole control of the machine. These resources include the time available for operation, the program and data storage available, and communication links with the world external to the computer.

A comparison of the means and mechanisms of resource management provided by various computer operating systems, including JANUS, is presented. Descriptions of inadequacies, both in hardware and in operating systems, are given, with suggestions on possible improvements in future implementations. In those cases where it has been possible to measure various parameters under JANUS operation, the measurement and a comment on its significance is provided. Reference manuals for JANUS and various control monitor tasks are appended, as well as thoughts on the possible implementation of other desireable processes.

A novel method has been developed to handle realtime processes. The computer may be used to simultaneously control devices, acquire data, and perform analysis and computation. Any process may be started or stopped at random, irrespective of the other usage of the machine. The flexibility introduced into the use of the computer, compared with conventional realtime systems, is impressive, since, if necessary, all

of the resources of the computer may be directed toward any goal, using a single operating system, without the overhead normally associated with such systems.

This is accomplished by providing within the resident monitor only those primitive functions dealing with resources common to all usage. Higher level functions, such as Input/Output, are provided by independent timeshared tasks. These tasks, with the features normally associated with conventional monitors, provide those functions necessary and sufficient to the operation of a specific set of problems.

JANUS: A REALTIME TIMESHARING COMPUTER SYSTEM FOR USE IN NUCLEAR PHYSICS EXPERIMENTS

By

John Oscar Kopf

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Physics

1968

Dedication:

For my wife Peggy

Who minded the kids.

ACKNOWLEDGMENTS

I wish to acknowledge my indebtedness to all the people who helped make JANUS, and this thesis, possible.

First and foremost, I am grateful to my advisor, Professor Aaron Galonsky, for his patience and encouragement, and for allowing me a free hand in the design and implementation of JANUS.

Secondly, I acknowledge the great assistance of Mr. Phillip Plauger, whose familiarity with other computers and computer operating systems, and whose willingness to engage in many all-night bull sessions, helped immeasurably in the design of JANUS, to say nothing of the fact that the original idea to write the timesharing system which became JANUS was his.

Third, I wish to thank Richard Au, Douglas Bayer, Carolee and William Merritt, Francis Schiffer, and Laurence Wilbur for their programming assistance in coding tasks, libraries, symbionts, and other parts of JANUS, as well as their aid in finding and correcting errors in JANUS. In leaving, I feel that JANUS is in good hands, and will continue to grow in use.

Fourth, I acknowledge the assistance of Mr. Lester Hanson and Mr. Donald Freedman, who, as on-site engineering representatives of SDS, were instrumental in quickly verifying, redesigning, and changing those design errors which are always present in a new computer. Without these changes, the Sigma 7 would be incapable of running JANUS. I

also wish to cite SDS, which delivered an operating system with the computer, without which it would have been impossible to even begin to start JANUS.

Fifth, I wish to thank Mr. Robert Belgard, who drafted most of the figures presented, and Mrs. Susan West, who typed the thesis.

I also wish to thank all of the other people at the M.S.U. Cyclotron Laboratory, too numerous to name, who in one way or another helped with JANUS. My thanks especially to those people whose constant insistance that I justify JANUS, thereby forced me to re-evaluate the design of JANUS at all stages, and caused me to formulate a flexible and self-consistent system.

Finally, I wish to acknowledge the financial assistance contributed by the National Science Foundation and by Michigan State University.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Physicists have used digital computers for as long as computers have existed. Indeed, long before the first computer was built physicists were suggesting that computing machines would be useful for generating astronomical and mathematical tables. More recently, with the advent of quantum mechanics, people such as Hartree called for the development of machines to perform computations for the calculation of wavefunctions and energy levels of atomic structures.

As soon as electronic digital computers became available, they were set the task of performing physical calculations. The usage of computers took great strides with the successive introduction of assemblers (which freed the programmer from the nuisance of bookkeeping relevant to the computer but not to the operation he wished to perform), and compilers (which permitted the programmer to forget all details of a specific computer, but instead to write programs usable on all computers which had a similar compiler available). The earliest languages (FORTRAN, ALGOL) were developed as aids to computation. Their success lead to the development of larger and faster computers. This in turn permitted the development of more powerful programs, which led to concepts of batch processing.

Batch processing was a logical development of the observation that, over reasonably long periods of time, the computer averages as much time spent reading cards and printing as it does computing. Since the bulk of the operations involved were read, print and punch, the computer

could obviously do more computation if these operations could be done faster. However, there are limitations to how fast a device may be operated. Further, these operations could be performed by a tiny computer. Thus multiple computer systems were developed, where a small computer copied cards onto magnetic tape which was later read by the large computer at much higher speed. The large computer then would write the output on other magnetic tapes to be printed later by the small computer. This provided improved usage of the large computer, and the value of the additional computation more than offset the cost of the additional small computer. However, since the computer now performed all operations automatically, it was no longer possible for the programmer to know just when his problem was being processed, and to interact with it. Furthermore, a fairly long time delay was required between submission of a problem and the return of the results.

People soon discovered that the small computers were useful in their own right, and that for many simple problems, results could be returned faster than with the large batch processing systems, since almost no computation was required. Thus a continuing development of small computers paralleled the development of the large computer systems. With the rise in computer technology, it became possible to build special purpose computers, usually consisting of a hard-wired program and a memory. In nuclear physics, these were best typified by multichannel and later multiparameter analyzers. While extremely useful, the allowed sequences of operations were built into the machine and were relatively inflexible, being limited to specific configurations which could be changed only with great difficulty if at all. It was usually necessary to do complex operations external to the analyzer. Further, it was

not possible to manipulate the data once taken, but only to dump it onto a secondary storage medium.

As computer technology improved, small computers became more powerful and less expensive. By the early 1960's, it became both feasible and desirable to attempt to interface a small digital computer to a nuclear physics experiment [1]. This was successful, and proved to be much more flexible than a hard-wired analyzer. Similar systems began to spring up in many places. As programs were written and used, it became apparent that the main value of a computer attached to an experiment lay not in its flexibility as an analyzer, but rather in its use in an interactive mode with the experimenter. For the first time it became possible for the experimenter to provide flexible and elaborate checks on the experiment, such that the computer could inform the experimenter of questionable operation or malfunction, or provide, on demand, a list of parameters which would aid him in determining the status of the experiment. Further, it became possible to analyze data as soon as it was collected, and compare the experimental results with theory. Parameters could be varied in both the theoretical calculations and the experiment, allowing more accurate measurements of the quantities of interest. There was a definite possibility that an experiment would proceed faster, since it might be possible at an early state to determine that the effect of varying one parameter was negligible, and could be ignored. Data analysis could be completed with the experiment, and questionable data could be retaken if necessary while the experimental configuration was still operative.

There was, however, one serious drawback to this system. The computer was dedicated exclusively to the use of only one person at a time.

The period involved might cover days or weeks, during which time no other person could use the computer. Since the experimentalists normally discovered a few days after their experiment that they would like to vary another parameter with respect to the data they had collected, since they already knew how to operate one computer, and since they knew that they could analyze their data under an interactive system much faster than by sending it to a computation center for batch processing, computer usage became saturated. A struggle invariably developed between the person who was using the computer and those who wished to use it for data analysis, data reduction, simple computation, and development of new programs to make use of the computer. Each time a new program was developed and added to the library of useful programs, saturation increased. Furthermore, it was apparent that the computer was not being used at full efficiency, since programs rarely used all of the resources of the computer, and for long periods various resources could be seen standing idle. Two or more people who required non-overlapping resource subsets could easily share the computer, if a mechanism suitable for sharing were provided. Then "A" could analize his data using a graphic display and teletype, "B" could be reading cards and printing out the results of a computation based thereupon, while "C" could copy a magnetic tape.

Such a mechanism exists, namely timesharing, based on the observation that if in an interactive mode of operation, a computer is normally idle while waiting for a person to respond, then during this time it could easily be responding to each of several users, without appreciable degradation of response to any one. As a result, each user would feel that the computer is devoted exclusively to his use. The computer could still be providing a batch processing facility in the background of its

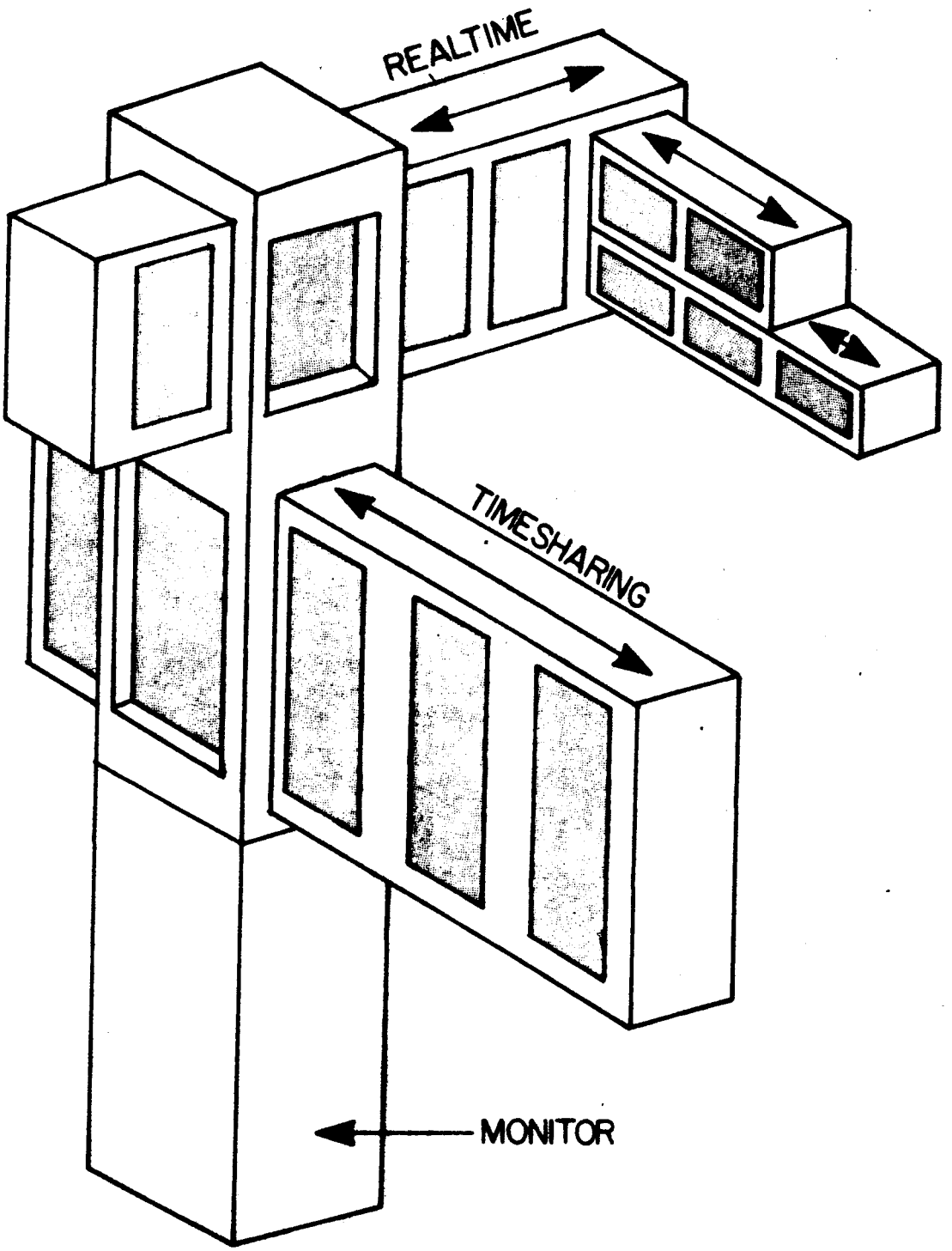
operation, or any other processing which was not time dependent. This background usage would be degraded by the interactive usage, but would still be keeping the computer busy and productive.

Timesharing schemes fall generally into three categories. These may be classified as follows:

1. Systems where all users are running independently, but where each is performing identical operations of computation, control, and input. Such a system would use the same program for all users, each differing only in the unique storage area he was using. In this scheme, the program is reentrant, such that it always assumes one or more pointers to the current area of storage it is manipulating. This scheme is effective where each terminal that may interact with the computer is identical in its capabilities and operation. Added flexibility may be provided by allowing the individual user to use his own programs, executing them from his storage area, to manipulate his data. However, any interaction between the user and his program must be handled by the main resident program, or monitor. Further, all allowed functions must be built into the monitor, and adding or changing a function is a non-trivial programming problem. A common example of such a timesharing system is that used by airlines for ticket reservations. Such a scheme is relatively easy to produce, since there is a finite set of operations allowed and desired. Its greatest deficiency lies in its lack of flexibility.

2. A second scheme is that of foreground-background usage, shown in Figure 1. Here an area of the memory is set aside for one or more foreground programs, which interrupt the program operating in the background as necessary to perform a specific set of functions and return

Figure 1. Memory allocation under a foreground-background scheme of timesharing. The vertical column represents core memory. The monitor is used to provide all control functions. The area of core devoted to background timesharing is successively occupied by a number of processes, all limited to the size of the background area. The realtime foreground area may be occupied by any one of a set of processes, (or divided so that it may be used by more than one process), but any realtime process generally locks out all other realtime processes.



CONVENTIONAL
TIMESHARING
WITH REALTIME

Figure 1.

control to the background when done. Usually there is a method of checkpointing the current background program, that is, saving it on an external storage medium, replacing it with an extension of the foreground program capable of performing certain complex operations, and when there is no longer any need for this, restoring the background program and continuing its operation from the point it was checkpointed. However, there is normally elaborate checking involved to insure that the foreground and background programs do not interact, as there would be great dissatisfaction on the part of the users if it was necessary, for example, to sort output because the foreground and background punched alternate cards or printed alternate lines. The big advantage of the foreground scheme is that of fast response to events, thus permitting evaluation of each event on its own merits. The disadvantage lies in the difficulty of changing the foreground. In a situation where the foreground is used to monitor and control a process, such as the operation of a manufacturing complex (eg. oil refinery) or complex machine (eg. accelerator), where parameters may be varied but where the foreground program is rarely changed, this is no real disadvantage. However, in a situation where multiple foreground operations may be in operation simultaneously, starting and stopping asynchronously with each other, severe problems occur with respect to keeping track of free memory and making efficient use of the memory.

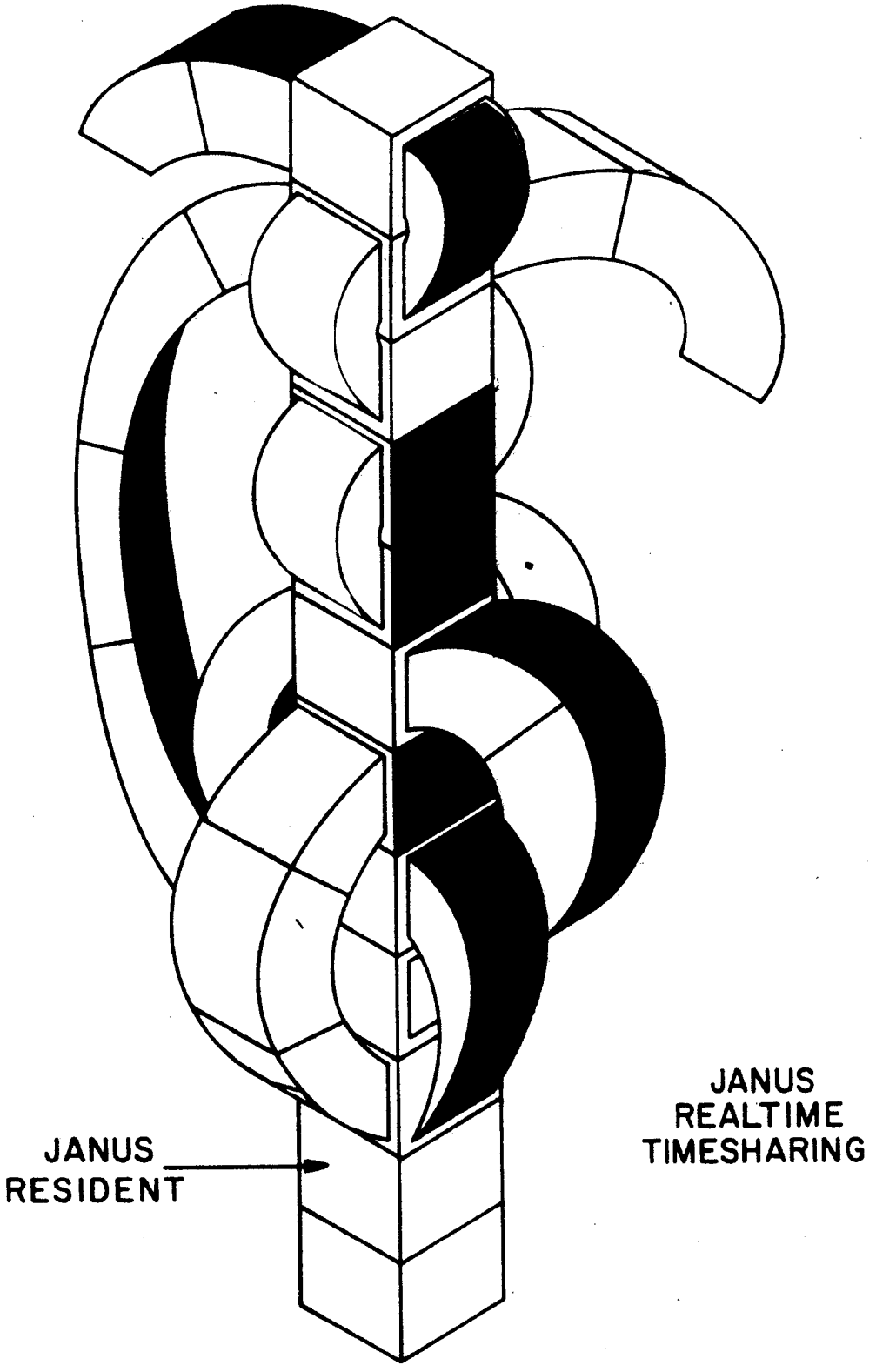
3. In the third scheme each user is performing operations completely independent of all other usage of the machine. This scheme, while being capable of the greatest degree of flexibility, is normally found to be so difficult to implement that restrictions are placed upon all usage. For example, no user is permitted to change the state of the

machine himself, and must request all state changes from the resident monitor. This monitor must on each request, determine if the request is valid, if the operation is permitted to the user, and perform other bookkeeping functions before actually going ahead and performing the operation. A typical operating system could easily require 20,000-40,000 words of memory at all times just for the resident programs. In addition, response time may be increased drastically such that, while still adequate for response to people, the response time is orders of magnitude slower than would be possible in a foreground system. This would seriously limit the usefulness of the system in an environment where events could occur thousands of times per second, such as in a nuclear physics laboratory.

This thesis describes a new scheme of realtime timesharing, which, while permitting the flexibility of scheme 3 above, also permits the response time associated with foreground programs, without many of the disadvantages of either scheme. It has the further advantage that the requirements of a resident monitor are kept to a minimum, since the task associated with each user performs all of his monitor functions, including all communication with the external world (INPUT/OUTPUT or I/O). This is shown in Figure 2. This is of great advantage in nuclear physics experiments, where an I/O operation might require a buffer of thousands of words, wasteful to make resident unless used frequently enough to justify it. (The acquiring of a multichannel or multiparameter spectrum can be thought of as such an I/O operation, where the storage allocated to the spectrum is in effect a single buffer.)

The operating system described is called JANUS, for the Roman god "...of all going out and coming in,...also the god of entrance into a new division of time" [2], thus the god of timesharing.

Figure 2. Memory allocation under JANUS. As in Figure 1, the column represents core memory, with JANUS at the bottom. Two tasks are shown, with the independent task address spaces twining through memory. Not all pages of a task need be in core at one time. The darkened pages of core are dedicated to realtime processes specific to the task which has the page.



STYLIZED REPRESENTATION OF
MEMORY USAGE UNDER JANUS

Figure 2.

2. THE USE OF CORE MEMORY

In order to provide a perspective for the discussion of JANUS, I will first describe how various other timesharing systems operate. Consider first the problem of sharing the core memory of the computer. How can more than one user make use of the core memory without the possibility that an error can interfere with another user? (Subsequent figures are keyed to Figure 3.)

The simplest scheme is to have only one user in core at a time, and all available core is his to use. This scheme is that used in Project MAC of the Massachusetts Institute of Technology on an IBM 7090 computer (Figure 4). It is also used in the Sigma 7 timesharing system developed by the Bubble Chamber Group at Brookhaven National Laboratory [3]. The users program is brought into core from an external storage medium (swapped), and started. If the program did not inform the resident executive program that it wished to exit early, then at the end of a fixed time increment execution is stopped, the current status is saved, and the program is swapped out to the external storage medium, freeing the core for the next user. This process continues for each user, until eventually the first user is swapped back into core, and his execution is continued. While this scheme has the advantage of simplicity, the amount of time spent on nonproductive bookkeeping (overhead) is high, as the computer is idle while swapping occurs. To provide a reasonable response time to each user, the interval specified (timeslice or time

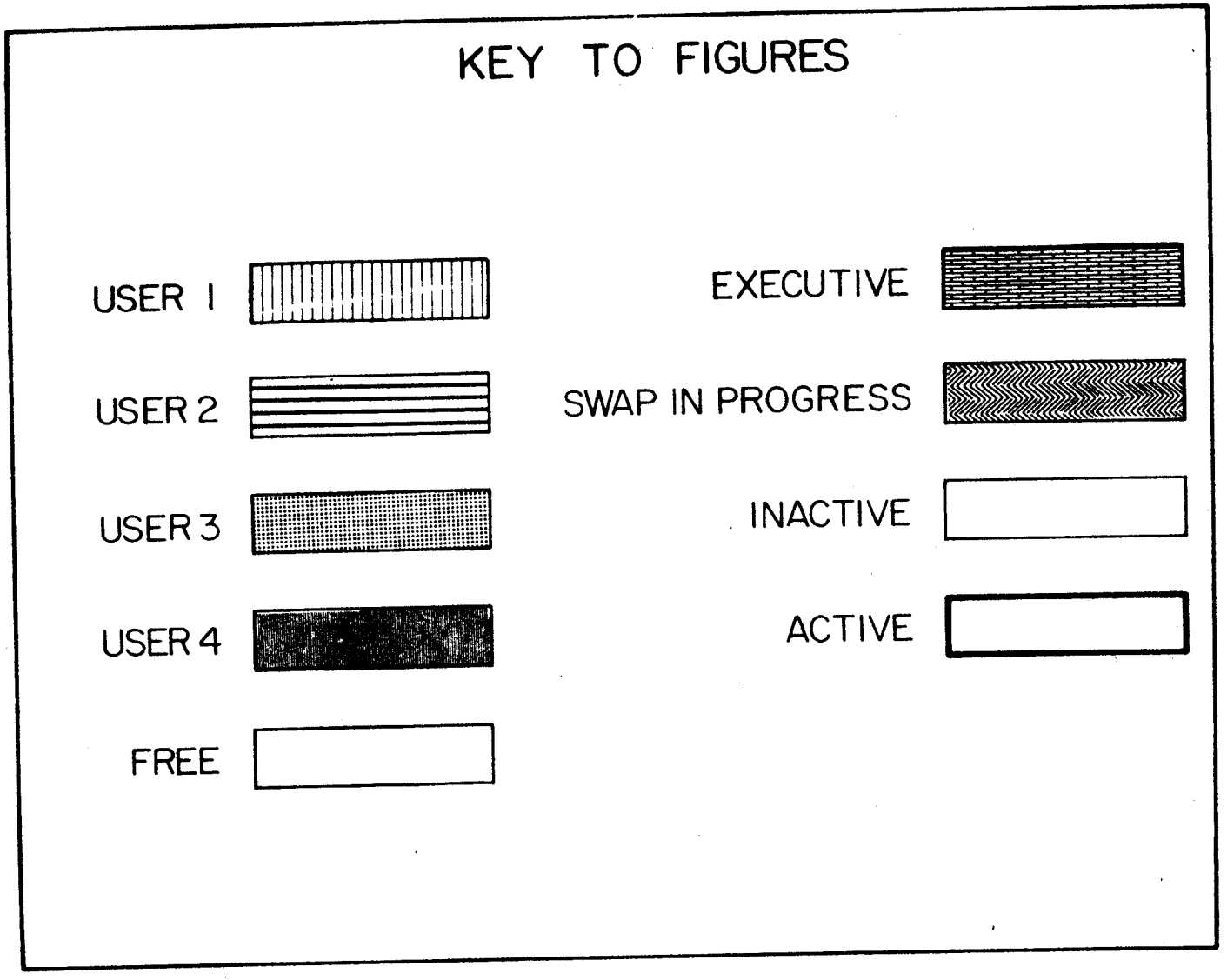


Figure 3. Key to subsequent figures. (Note the use of the word active, which in this context refers to a task executing a timeslice.)

Figure 4. Memory allocation--project MAC. User execution alternates with swapping (the process where the program is transferred to or from memory).

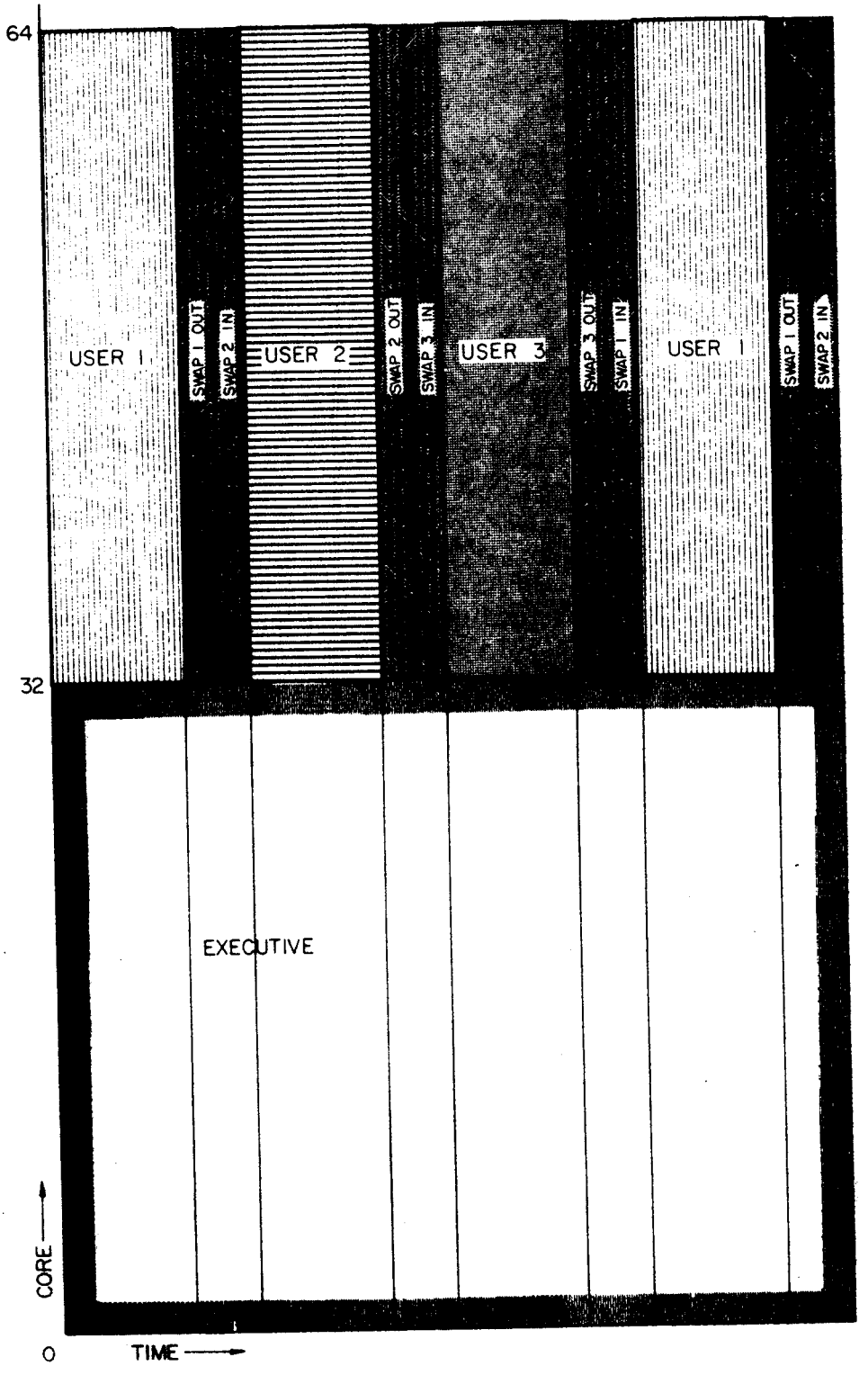

64
32
0
CORE
TIME
USER 1
SWAP 1 OUT
SWAP 2 IN
USER 2
SWAP 2 OUT
SWAP 3 IN
USER 3
SWAP 3 OUT
SWAP 1 IN
USER 1
SWAP 1 OUT
SWAP 2 IN
EXECUTIVE

Figure 4.

quantum) must be short--normally fractions of a second to each user. An example of the problems involved for such a scheme is demonstrated in the Brookhaven system, where the average time required to replace one block of 8192 words with another is 56 milliseconds. With the 1 second timeslice used, this provides 5.6% overhead, but with 6 active terminals as much as 5 seconds may pass before the computer can respond to the user. Three second response time is normally considered a reasonable upper limit. To provide this response time, a time slice of .5 seconds would have to be used, and overhead would increase to 78 milliseconds. Most terminal usage consists of the computer reading in a typed record, examining it, possibly commenting upon an error, and requesting new input, a process which normally takes much less than 1 second. In this case, the overhead would increase to a large value. Inter-user protection need not be considered, however, since they cannot get at each other.

By constraining each user to a separate part of the core available, such that more than one user may fit into the core memory, advantage may be taken of the fact that most computers suitable for timesharing are capable of asynchronous I/O operations, such that I/O may coexist with program execution. Thus one user may be executing while another is swapping in or out. The swapping I/O overhead is negligible as long as the timeslice is greater than or equal to the swapping time. However, a new problem arises--that of relocation. To make efficient use of the memory, a program should be capable of executing correctly wherever it may be located. Unfortunately, programs tend to reference absolute addresses.

The simplist method of treating relocation is to ignore the problem. This approach was taken by Dartmouth College with a GE 265 [4] (and

more recently a GE 415) computer (Figure 5). Available core is divided into two areas, "high" and "low" core. Execution is alternated between high and low core..as low core is executing, high core is undergoing a swap. However, a program loaded into high core will not run in low core, and vice versa. A bad job mix can cause an excess of programs in one area or the other, with the result that either the low density area users get more computer time, or else the computer becomes inefficient, as time must be spent waiting for swap in the high density area. Additionally, any system library program which is available to all users must be kept in both a high and low version. Protection is provided by a bound register, which specifies the highest and lowest legal core references permitted.

In order to treat relocation adequately, so that a program may run in different areas of core without revision, special hardware must be used; if the relocation operations were performed by software the overhead would be tremendous.

There are three methods of automatic relocation used. Two of these are almost identical, with only a slight difference in emphasis.

The first of these methods uses a location register and relative addressing. Each address is relative to the referencing instruction. The actual reference is made by adding the location register to the address specified. The block of code will now operate anywhere in core automatically. This scheme has been most successfully applied to the SDS Sigma 2, which is not however used for timesharing.

A second scheme uses a base register. Addresses specified are relative to the beginning of the program, rather than to the address of the instruction; otherwise operation is identical to that outlined above.

Figure 5. Memory allocation--Dartmouth project. A new program is swapped into core memory while the current program is executing. However, an imbalance in the number of jobs assigned to high and low storage causes a delay at the end of the third timeslice, as there is nothing to do but wait between USER 3 and USER 1.

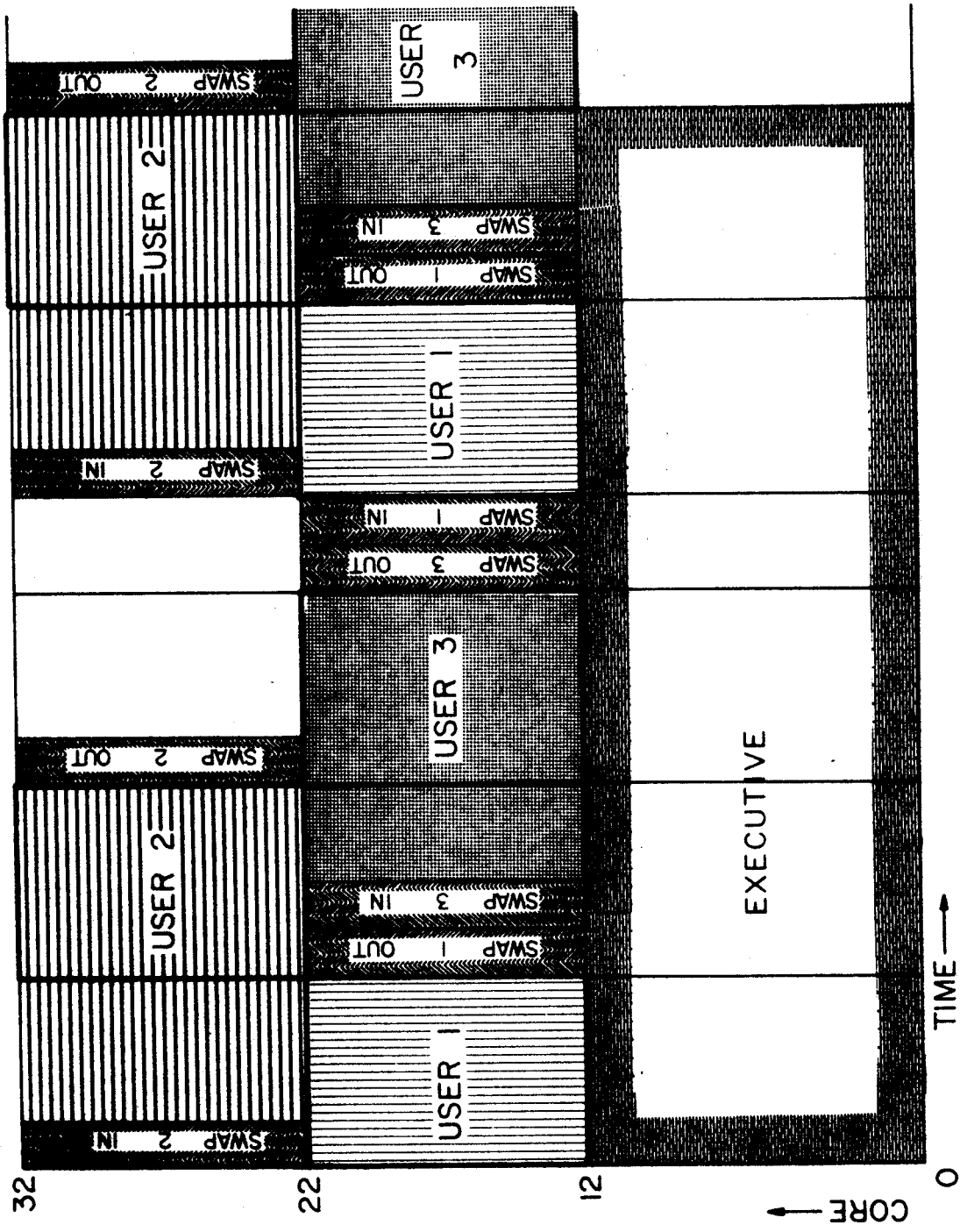

32
22
12
0
CORE
TIME
EXECUTIVE
USER 1
USER 3
USER 2
SWAP 1 OUT
SWAP 3 IN
SWAP 2 IN
USER 1
USER 3
USER 2
SWAP 1 IN
SWAP 3 OUT
SWAP 2 OUT
USER 1
USER 3
USER 2
SWAP 3 IN
SWAP 1 OUT
SWAP 2 IN

Figure 5.

This scheme is used on the IBM 360 computers [5, 6], and in the PDP-6 7) (and now in the PDP-10) computers (Figure 6). There are two advantages of this scheme over the Dartmouth scheme. First, successive users are placed where there is room in core, without the problems associated with high and low core areas. Secondly, programs may be of variable length, up to half the available space in extent. Short programs can coexist with longer programs. This scheme further introduces the concept of pages--a basic unit of core size. In the PDP-10 each program consists of an integral multiple of 1024 word pages. Protection is again provided by a bound register, the lower limit of which is also the base register.

The third scheme of auto-relocation involves a memory map. First developed by Project GENIE at the University of California, Berkeley, using an SDS 940 computer, it is also used by the IBM 360-67 [8, 9], and JANUS in an SDS Sigma 7. Figure 7 shows the use in an SDS 940 computer, which uses 2048 word pages. Note that pages which are modified (M) while a program is active are flagged to be written back (W). Since for unmodified pages there is a true copy on the external storage medium, these pages are preferentially chosen to be overwritten thereby cutting down the number of swap operations necessary. The penalty for reducing the number of swap operations is the necessity of searching through a table of content-associated core pages, to determine if a page of a program is currently in core. Programs execute in a virtual address space, connected to the real address space of the core memory through the map. Thus contiguous virtual pages need not be in contiguous real pages of core, but may instead by located wherever most desirable. Inter-user protection is afforded by a multilevel page protection system, used to

Figure 6. Memory allocation--PDP-6 computer. Swapping occurs concurrently with execution, but the efficiency is higher than in Figure 5, as no limitation is imposed by the use of specific areas. Note that a significant part of memory is rarely or never used.

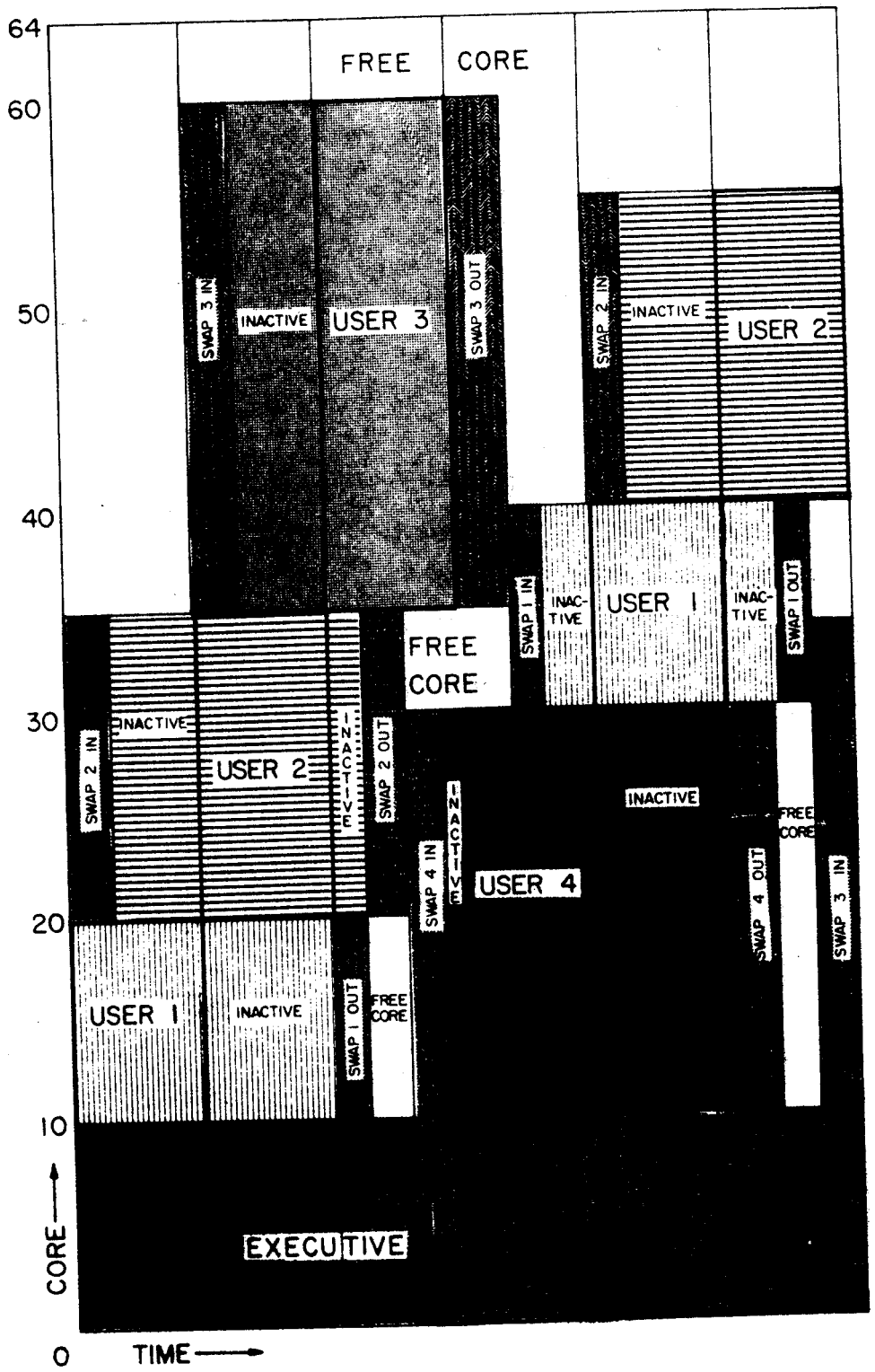

64
60
50
40
30
20
10
0
CORE
TIME
FREE
CORE
SWAP 3 IN
INACTIVE
USER 3
SWAP 3 OUT
SWAP 2 IN
INACTIVE
USER 2
SWAP 1 IN
INAC-
TIVE
USER 1
INAC-
TIVE
SWAP 1 OUT
FREE
CORE
SWAP 2 IN
INACTIVE
USER 2
INACTIVE
SWAP 2 OUT
SWAP 4 IN
INACTIVE
USER 4
INACTIVE
FREE
CORE
SWAP 4 OUT
SWAP 3 IN
USER 1
INACTIVE
SWAP 1 OUT
FREE
CORE
EXECUTIVE

Figure 6.

Figure 7. Memory allocation--project GENIE. Problems are initially brought into free core. When no more core is free (TASK 4), those pages are chosen which are not currently in use and are identical to the copy on the disk (unmodified), to be overwritten by new pages. The resultant fragmentation of jobs is offset by the use of a memory map. Not until the sixth timeslice is it necessary to write a page out to the disk (TASK 1, Page 5). This scheme is also used in JANUS. (The P numbers are pages within each user's address space. M signifies that the page is modified, W indicates that it must be written back onto the disk.)

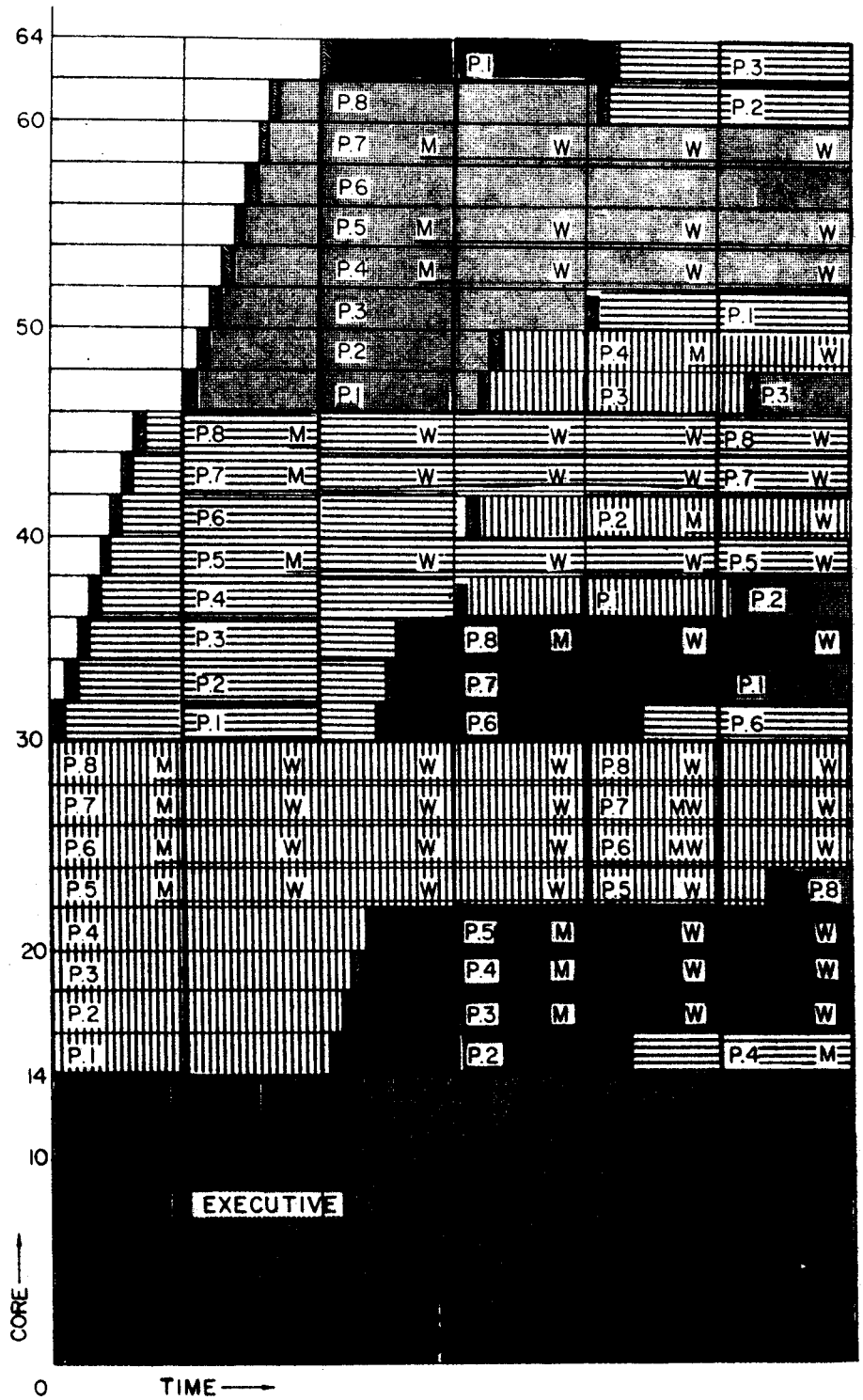

64
60
50
40
30
20
14
10
0
CORE
TIME
EXECUTIVE
P.1
P.2
P.3
P.4
P.5
P.6
P.7
P.8
M
W

Figure 7.

protect and monitor the usage of pages.

Under a mapped paging scheme, the usage of each page may be closely monitored--closely enough, in fact, to permit demand paging. If the executive system can be informed whenever a page is referenced, and if the user can be locked out of some of his own pages, there is no longer any need of the whole program being in core. Those pages currently being used can be brought in, and if a valid reference is made to a page which is not present (demanded), the current timeslice can be stopped, and conditions set up such that the demanded page will be available during the next timeslice for the program. Further, if a page is not referenced for some period of time, it may be safe to assume that it will not be referenced again for a while, and eased out of core memory in order to make room for pages in use.

JANUS uses a mapped memory usage scheme, but with an important advantage over that specified above. Much of the executive is unique to the task, rather than resident and common to all tasks. As a result, it is entirely up to each task if demand paging is to be used. Furthermore, any task's monitor may dedicate one or more pages, making that area resident until undedicated (Figure 8). These portions may be connected to interrupts, permitting realtime operations asynchronous to timesharing. These pages form resident islands, and timeshared usage maps around them. All the advantages of foreground usage result, without the rigidity inherent in conventional foreground-background systems. The added ability to solve problems which are actually larger than physical core, without requiring special techniques of the programmer, such as overlays, is a boon.

Figure 8. Gross memory allocation under JANUS, showing realtime processes. Using a large time scale, all short-term details of memory usage are omitted. Shown instead are areas dedicated for realtime processes. The blank area is available for swapping. The activation of four tasks is shown, each of which immediately initiates a realtime data input process. These tasks, in order of appearance, might be MOIRAE, JBCM, DATA TAKER, and JFCM. Note the interleaving of lineprinter output of tasks 1 and 2. The variable size scope display is a measure of the amount of realtime buffer area required as a display is expanded to show relationships instead of detail.

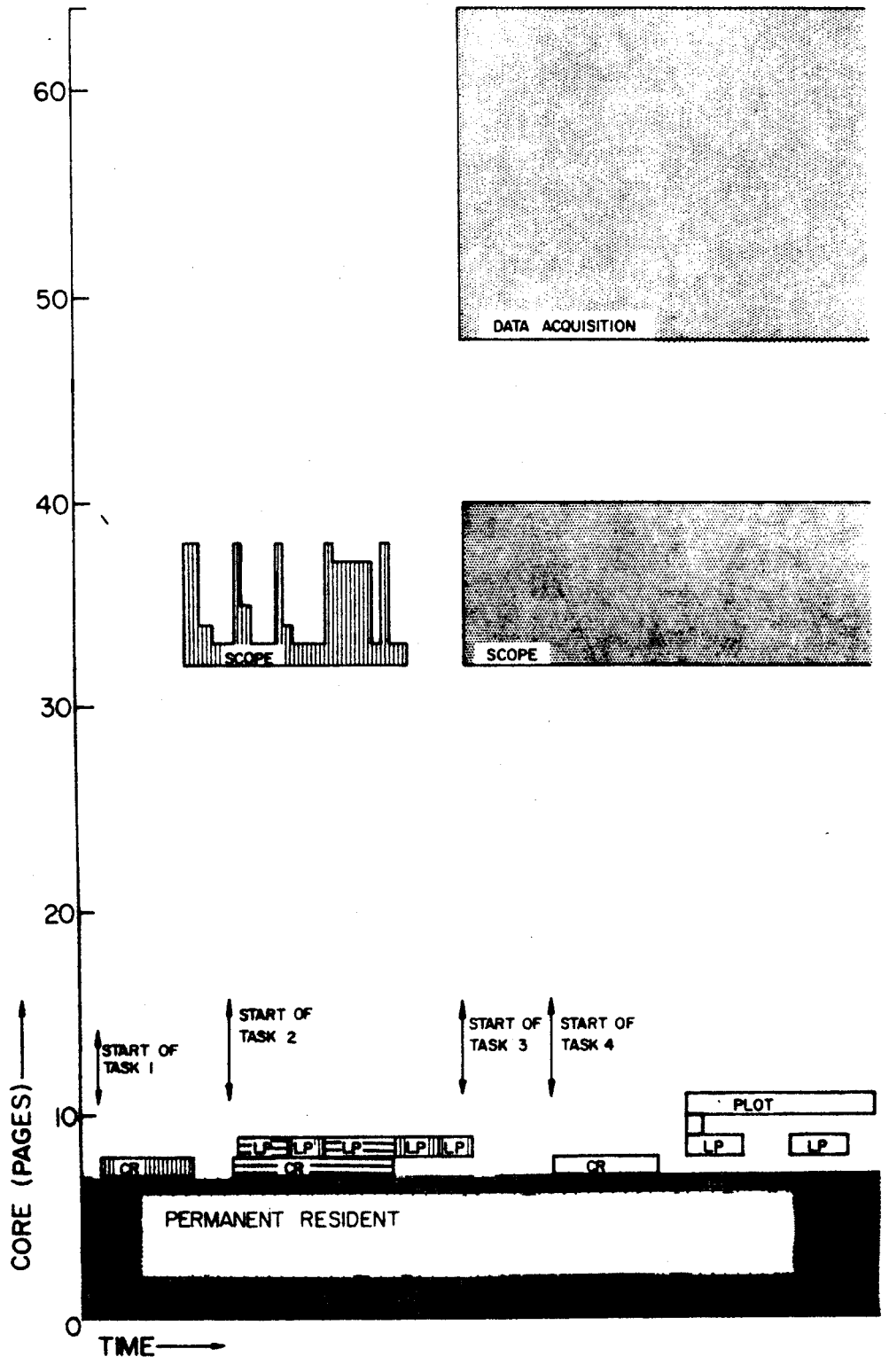

60
50
40
30
20
10
0
CORE (PAGES)
TIME
DATA ACQUISITION
SCOPE
SCOPE
START OF TASK 1
START OF TASK 2
START OF TASK 3
START OF TASK 4
PLOT
LP
LP
LP
LP
LP
LP
LP
CR
CR
CR
PERMANENT RESIDENT

Figure 8.

3. BULK STORAGE

Almost all time sharing schemes require, in addition to the core memory, additional bulk storage to keep programs, libraries, and data. In general, this storage is addressable, in that a specific block of storage may be located without searching all the storage medium. With certain exceptions, magnetic tape does not fall into this category. Instead, magnetic tape is a serial or "stream" storage medium, where records relative to the current record may be referenced. As such, it is useful primarily as an archival storage medium, where data stored thereon is not capable of change without either destroying all succeeding records or requiring a copy operation to move the data from a source tape to a destination tape, making changes as necessary in the new copy. This use is adequate for storing data, and for some functions such as holding lineprinter output. It is inadequate for working bulk storage in a timesharing environment where response time is critical.

In this environment, addressable storage is required. Commonly used storage takes many forms, some of which are indicated in Figure 9. Shown also is the typical access time and range of storage capability for that form of storage, as well as cost. Cost and storage are in terms of bytes, a byte containing 8 bits of data. It is readily apparent that as access time decreases the cost increases. This factor of cost/byte is what normally sets an upper limit to the practical storage capability for a particular form of storage.

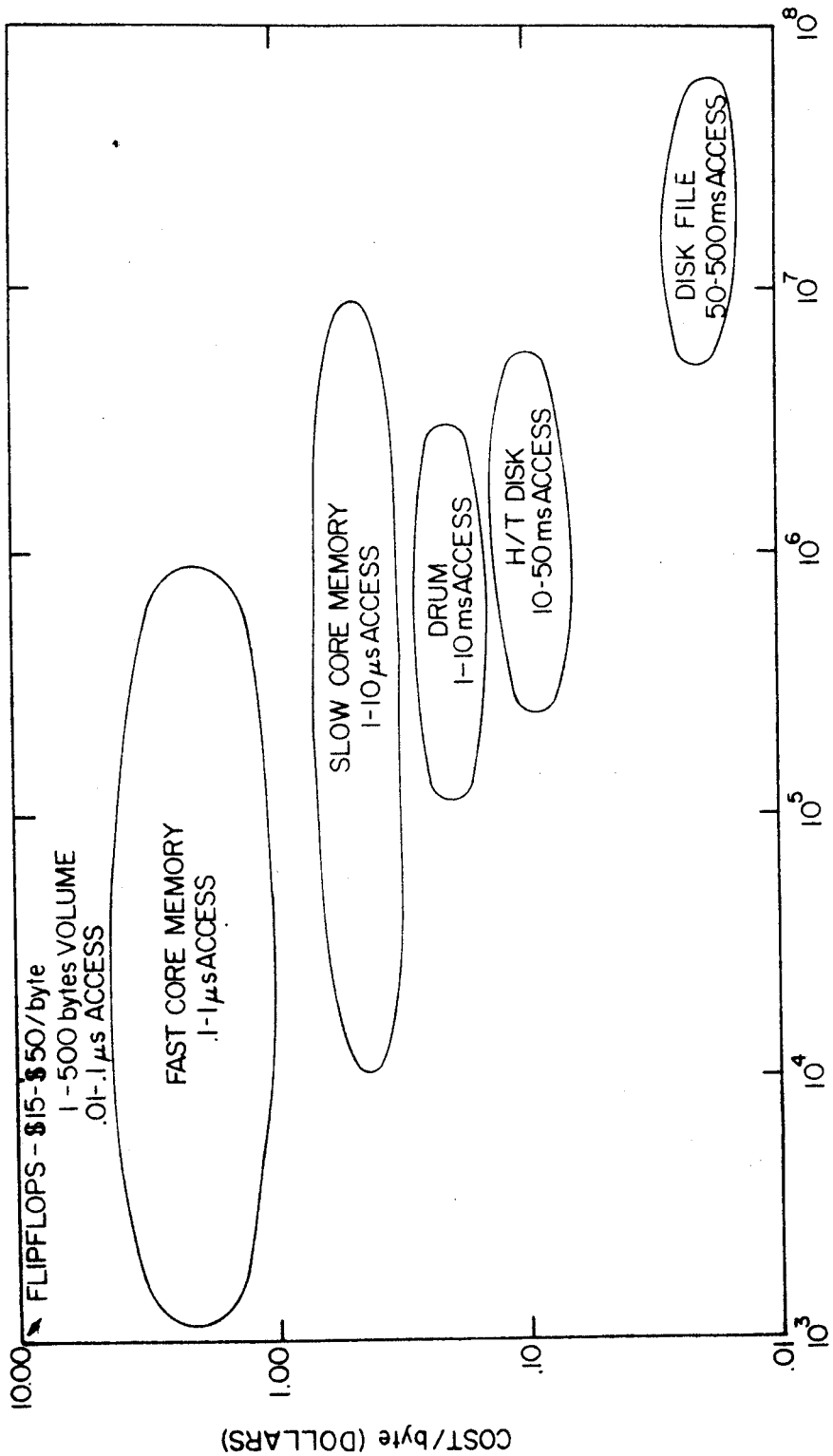


Figure 9. Characteristics of various bulk storage media.

Timesharing systems require an immense amount of bulk storage, normally approaching infinity as closely as possible. In addition, it is normally desired that access time be as low as possible. To be practical in terms of cost, it becomes necessary to build a bulk storage structure, using a set of storage forms of differing characteristics. Thus, one uses a fast, low volume medium as well as a slow, large volume medium.

This storage structure may take three possible forms, of which the linear form is rarely used in comparison with hierarchical and hybrid structures, at least for timesharing usage.

The most easily understood structure, however, is the linear structure, shown in Figure 10. Here the storage is an extension of the core memory, but suffers from the difficulty and inconvenience of executing programs directly from the storage. Its advantage lies in the fact that there is a unique address associated with every piece of storage, both core and bulk. Operation consists of copying blocks of data into core memory, manipulating them, and then replacing them.

The opposite extreme is the hierarchical or pyramid structure (Figure 11). In this scheme, unused sets of data are kept at the lowest (large capacity, slow accessibility) level of storage. If referenced, the set of data is brought into core, and, if necessary, later written back. The system executive does automatic accounting of usage--if a data set is used frequently enough, such that time spent accessing a level of storage becomes significant because of frequent references, that data set is copied through core to the next higher level of storage. Depending on the algorithm used, the original block of storage may or may not be freed (depending on whether or not the storage is referenced

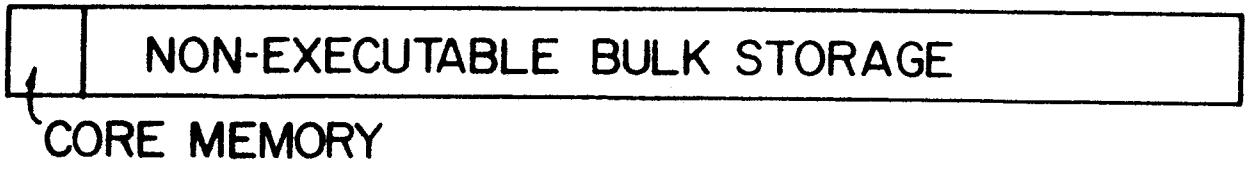


Figure 10. Linear bulk storage structure. (Used by CDC 6600 (extended core storage) to hold programs and non-executable I/O buffers.)

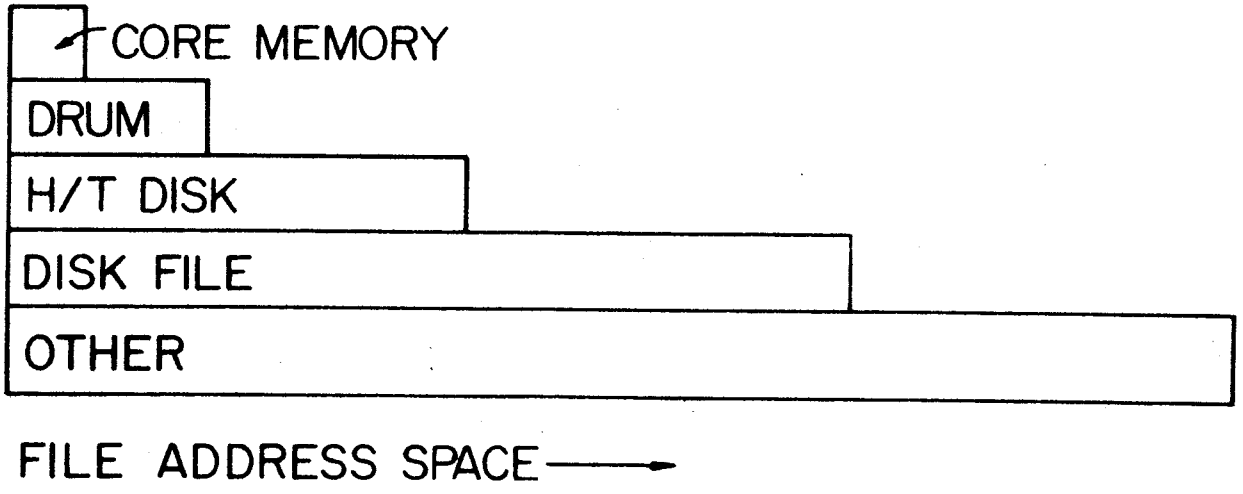


Figure 11. Hierarchical storage structure. Access time decreases with height. Each storage medium has an independent address space.

by a unique name, or by its areal name on the lowest level). As the use of this procedure unaided would tend to fill higher levels of storage, a mechanism must be provided to purge a level of some data sets, whose usage frequency does not warrant such a high level of storage, to a lower level. This may be done periodically, as well as upon demand. Efficient use is made of bulk storage, but there are two disadvantages to such a scheme. For ease of description, I will assume a data set consists of a group of pages, and consider the use of a single page.

First, and most important, a page must be referenced by a unique name. This name will have associated with it indicators telling the current level and location within that level of the specific page, as well as some sort of usage indicator. A table is required to permit the association to be made. In addition, some indication must be provided for free pages on each level, as well as for free names. In order to provide speed of reference, at least part of this table must be resident. Since a minimum of one word/page is indicated, and since storage of 100,000,000 bytes (50,000 pages) may be available, it is readily apparent that an excessive amount of core storage is required for the table. There are alternatives, requiring only a list of the contents of the highest level of storage, but these are expensive also, as each reference requires searching the table to see if the reference is there--a time consuming operation.

Secondly, the overhead introduced in purging levels is non-trivial. A typical hierarchically organized system is the IBM 360 Time Sharing System (TSS) [8, 9]. Using a complex algorithm to improve efficiency, measured overhead is still 80-90% [7].

The third form of storage structure (Figure 12) is the hybrid

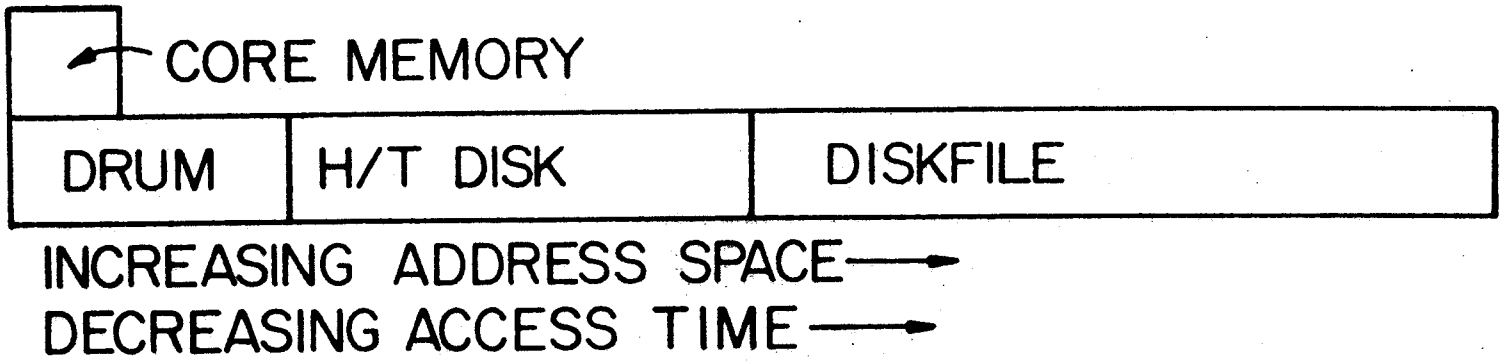


Figure 12. Hybrid storage structure.

structure. In this scheme, all names are absolute rather than relocatable, and no resident table is required, except for a list of unused pages for each storage medium, and an associative table of the pages contained in core memory. Pages are allocated according to expected use and duration of existence--program libraries in slow storage, active programs in fast storage. If a storage medium is full, space is allocated from a slower medium. A program is copied to fast working storage and renamed when it is brought active, and that area is freed when it exits. As a result, little overhead is required compared with the hierarchical structure, and the benefits of the linear structure apply, without the problems associated with nonexecutable storage.

JANUS uses a hybrid storage structure. As the MSU Sigma 7 configuration includes only a single bulk storage medium (a 1.5 megabyte H/T disk), it is a simple structure. However, only a relatively minor change in JANUS is required to implement one or more additional storage media.

4. SCHEDULING OF TIME

In any timesharing scheme, the timesharing is effected by dividing the time available into quanta, or timeslices, which are allocated to successive users. Scheduling involves two parameters--timeslice duration, and ordering of users.

Timeslice duration may be fixed or variable, and a mechanism is usually provided to terminate a timeslice early. A fixed timeslice has obvious meaning; each user gets the same quantum of time for his problem.

A variable timeslice is normally used in a timesharing system where the status of the machine, the system, and the previous history of a particular task's usage is available. On the basis of these parameters (and possibly others, which may be defined by the user), an "optimum" timeslice is calculated for each user each time. Thus, if the machine and system are lightly loaded, a longer duration is provided than if a heavy load exists such that many users must be serviced within a given period of time. Again, in a priority oriented system (see below), the duration is related (perhaps proportional) to the amount of time a user's problem has already taken. The rationale for this scheme is that, if a problem has already taken a certain period of time, a longer timeslice alloted to it will cut down on time spent for system overhead, and more productive work will be accomplished. For example, a problem which has already required N timeslices of duration T may be

given a duration of 2T for the next N timeslices. If still not done, the duration might rise to 4T for the next 2N timeslices. Since an extremely long calculation, such as frequently arises in scientific work, might require hours of computer time, a mechanism must be provided to permit the abortion of a long timeslice, in order to allow access to the machine by other users.

A further perturbing factor may be the admission of a user to specify his own timesharing, as in tasking under PL/1 [10]. In this case, a user may start subtasks to operate concurrently with the controlling task, and specify what portion of the time the computer is to spend on each. Then the duration allotted to the user must be divided into appropriate sub-quanta to permit each of his tasks its proper allowance of time.

The second phase of scheduling involves ordering the access of a user to the machine. The simplest ordering is to place all users into a "ring". In a simple ring, all users, both active and inactive, are in a circular structure. Control passes from user to user sequentially, skipping those who are inactive. Users may pass from active to inactive states and vice versa. The access time for a user then depends on his position in the ring relative to the currently active user. (A user is active if his program can proceed with computation, rather than wait, as for example for input.)

A more sophisticated approach is to have a ring of active users, and a list of inactive users. When a user is activated (goes from the inactive state to the active state), his task is inserted into the ring as the next task to run. This has the advantage that an activated user task has a short access time, and therefore fast response time. If

the task stays active for longer than a timeslice, he enters the normal ring sequence.

An extension of this scheme is the assignment of priorities to the tasks (the above is a two-level priority scheme). A multilevel priority system is generally an aristocratic system: all tasks at a given level are exhausted before proceeding to a lower level. If a level is being executed, and a task appears on a higher level, the lower level processing is discontinued. A task activated is normally entered into a high level.

Under priority scheduling, there may be a pyramid of rings (Figure 13). These rings leak--if a task stays active for long enough, it drops to a lower priority level. If it goes inactive, it drops to the lowest (inactive) level. As described above, lower priority tasks may be given a longer time quantum, to offset the fact that they may be entered less often.

In general, the parameters used to define priorities and timeslice duration are produced by an empirical fitting process, based upon some specific mix of possible jobs. In a terminal-oriented timesharing system, a .1 second quantum may be used for the highest priority, to provide fast terminal response time, and may expand to several seconds at a lower level, where long computations are performed.

JANUS uses a scheduling algorithm with a two-valued timeslice and a four-valued priority scheme. A task is allocated a .1 second timeslice, unless it is the only active task in the machine, in which case it is alloted .4 seconds. If a realtime process requires rapid response from its associated task, it can bring the task active, and even effect jobchange for the currently active task.

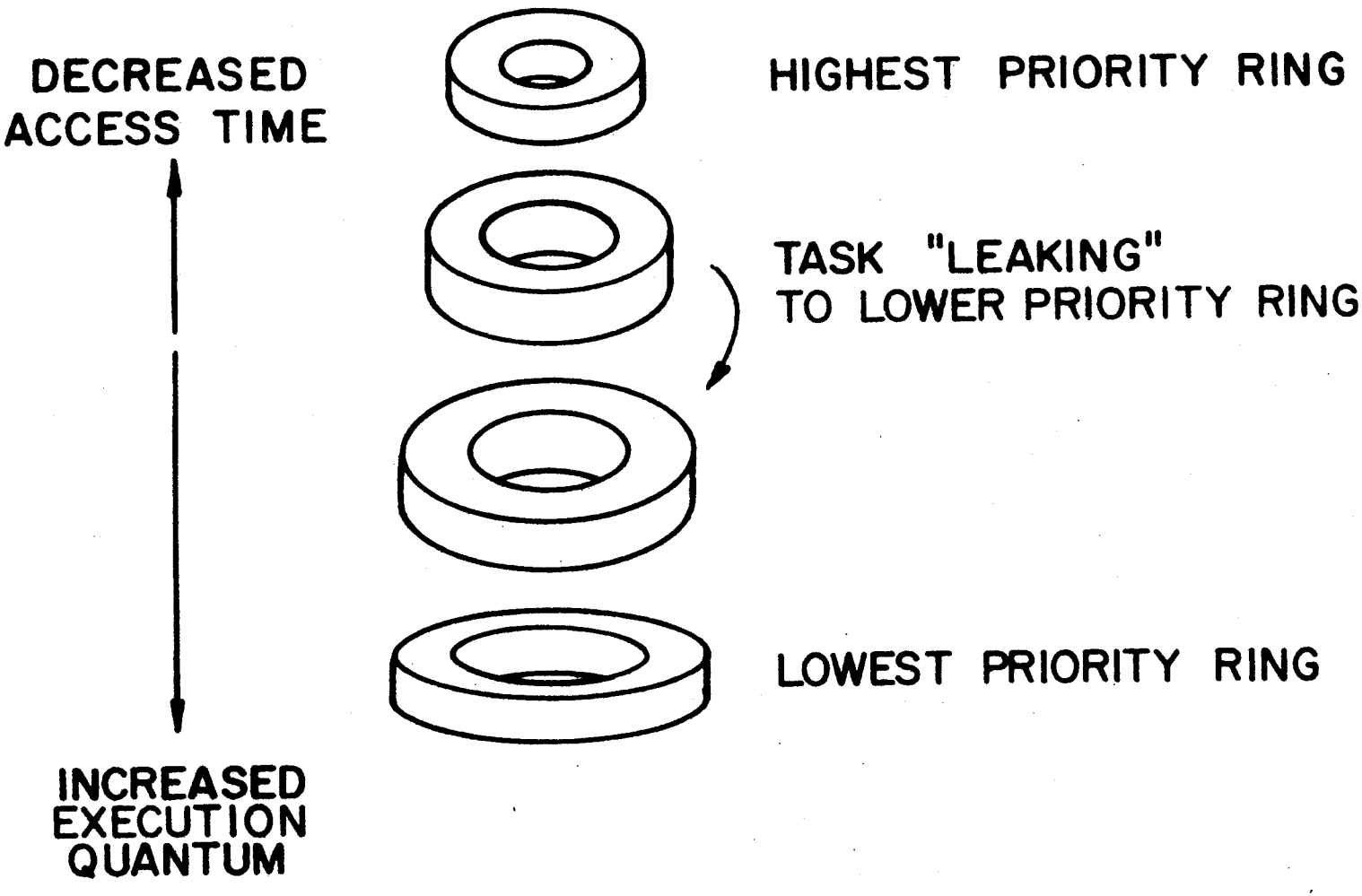


Figure 13. Priority scheduling. A task in a high priority ring will be accessed more rapidly than one on a lower ring.

The possible priorities are inactive, normal, hurry, and rush, in order of increasing level. The structure is that of a simple ring--an activated task remains in its normal sequence. However, a task may be placed in a higher priority level than normal. The use of a higher priority overrides the normal sequence of operation, and the first high priority task encountered is used.

The difference between hurry and rush priority is one of efficiency. Under rush priority, a task is made ready and started, even if another task is ready to proceed. Under hurry priority, if another task is ready to proceed, it does so, rather than have the computer wait while the high priority task is readied. A task found in either high priority state is reduced to normal priority on entry; thus high priority applies only to the first access. On return from a high priority task, execution will proceed with the next task in normal sequence from it, unless another high priority situation exists.

Since JANUS is a system geared to realtime operations, there is a second form of timesharing available. This is by means of interrupts (Figure 14). A realtime event may be defined as the occurrence of an event asynchronously with the operation of the computer. If the event is attached to a hardware interrupt, it is possible to rapidly switch the complete state of the computer, interrupting the current process, and transferring control to an interrupt routine. This routine may take the necessary action based upon the event, and then return control to the interrupted process. While the interrupt is active, it can have performed various operations, including the activation of the associated task. In general, all I/O operations are associated with interrupts, including data acquisition. Indeed, the basic JANUS functions of

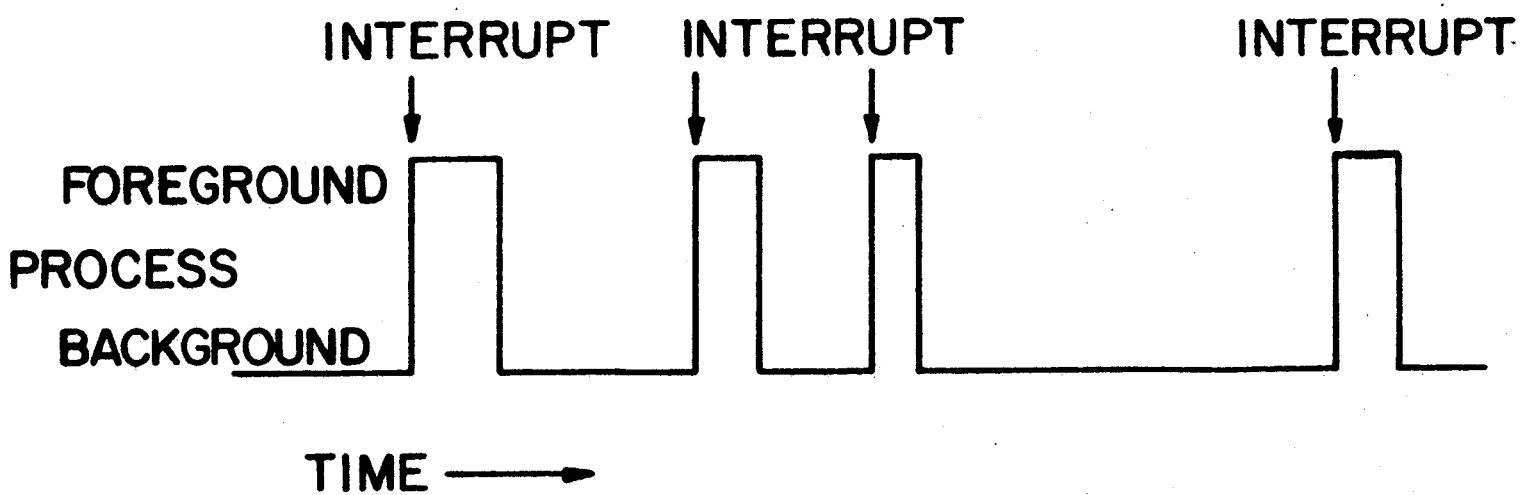


Figure 14. "Timesharing" by interrupts. Execution alternates between the background (lower) and foreground (upper) line, on the basis of interrupts. While a simple case is shown, interrupts need not all belong to the same foreground process.

swapping and jobchanging are associated with interrupts, although of lower priority than those used for data acquisition.

Realtime processes operate in the foreground of the computer--timeshared operations in the background. As a result, it is possible to perform a nuclear physics experiment at several levels simultaneously, such as realtime data acquisition, realtime I/O (e.g., plotting), realtime timeshared data analysis, and timeshared theoretical computation. Each of these can be treated as an independent process, and thus the processes necessary to a particular experiment may be selected and started. If it is discovered that an additional process is required during the course of an experiment, it may be selected and added to the set of active processes. Similarly, a process no longer required may be dropped.

5. DESIGN GOALS OF JANUS

JANUS was first designed to optimize realtime processes, specifically data collection, control, and interactive processes relating to experimental nuclear physics. Only after it was decided that these were the most important functions of the computer for our application, was an attempt made to determine how to proceed [11]. An analysis of projected usage indicated that any realtime process had associated with it a large body of associated data and program which was not used continuously, and which was of low priority. Also indicated was the fact that, while a realtime process was active, large portions of the computer resources would be standing idle. Furthermore, the capability of operating multiple independent realtime processes simultaneously was desired.

It rapidly became apparent that a timesharing system was desired, with a capacility of running realtime processes in the foreground, while the associated programs (and independent ones also) would run in the background. Efficient use of core memory indicated the need for a small resident monitor, as well as memory mapping. Tasks were defined, and their requirements and capabilities were delineated.

The result was the JANUS philosophy:

1. The highest priority use of the computer is to service interrupts, and external data collection associated interrupts should be of highest priority. There should be no critical timing requirements on any sequence of code, requiring the inhibiting of interrupts.

2. Any hardware in a given installation should be available to a user if he needs it.

3. The only restriction on any user should be that all other users are protected from him, as he is from them.

4. No user should be aware of any other use of the machine, except when a delay is required because of a request for a currently unavailable resource.

5. It is the responsibility of the task associated with a user to insure proper operation, as JANUS is to impose no arbitrary restrictions by checking on the task. JANUS should be invoked only for control functions.

6. What a user does not know about the system should not hurt him, as long as he follows reasonable conventions as to usage.

7. A user need not have any knowledge of anything he doesn't use, to the extent that it does not exist in the machine he is using.

These goals have been met in JANUS.

6. UNIQUE FEATURES OF JANUS

The system is based on the observation that a piece of a program need only be accessible while it is being referenced. In a timesharing system, programs succeed each other at small intervals of time. The only piece of any such program which need be in core at all times is that piece which may be called upon asynchronously to the normal sequence of timesharing. It is readily seen that this description is exactly that of realtime operations, which can be extended to include all forms of I/O. If a mechanism is provided for a task to make a part of itself resident for the duration of a realtime requirement, then it becomes feasible to timeshare monitor systems, since, in general, monitors provide primarily for realtime I/O functions and any operations or usage which it is not desirable to allow to the user directly.

Further, if a mechanism exists to determine the usage of a block of storage including its accessibility, then the only blocks which must be available each time the task is active are those blocks to which the mechanism does not apply. Other blocks need not be accessible, or even in core, if a mechanism is provided for fetching them as necessary.

Again, if a mechanism exists for automatic relocation, such that a given piece of program can be operated successfully from different parts of the real computer memory, then these mechanisms can be used as a memory expander, such that programs can operate anywhere in the address space (chapter 9) of the computer whether or not that address space

corresponds in full to existing memory.

Furthermore, each task may operate in a completely independent address space, or several tasks may intersect in one or more blocks, which need not, however, correspond to the same piece of address space in each of the intersecting tasks.

In order to describe how these mechanisms are implemented in JANUS, it will first be necessary to describe unique features of the target machine, a SDS Sigma 7 computer. Certain features of JANUS have been influenced greatly by the operations allowed by the hardware. Some of the operations necessary to JANUS are provided quickly and easily by the hardware, showing the power and utility of such hardware, while others equally important must be implemented in a less than straight-forward manner, and would benefit greatly from the existance of hardware suited to the application.

7. THE TARGET COMPUTER

The SDS Sigma 7 computer [12] is a relatively new third generation computer of which MSU acquired the first sold. A first generation computer is generally typified as consisting of some form of memory and a processor, which can acquire specific data words (instructions) from the memory and based upon these instructions manipulate other data words. This generation usually used vacuum tubes. Second generation computers are typified by the use of transistors, and include such features as index registers, I/O channels (a device which shares the memory of the computer, and which, on command, can perform asynchronous operations for the computer, such as transmitting a whole string of data elements, rather than just one), and interrupts, allowing multiple use of the computer. The third generation computer is distinguished by the use of integrated circuits, and features such as privileged instructions, paged memory, memory mapping, flexible memory protection, "scratch pad" registers, and I/O processors. These are discussed in turn below.

Privileged instructions: the computer may be operated in either slave (computational) or master (control) modes. In slave mode, all instructions relating to the internal use of the computer are permitted, but certain control instructions are illegal, and memory usage is controlled. Conversely, in master mode, both internal and privileged control instructions are permitted, including those which change the status of the machine, but the core monitoring capability is lost.

A paged memory means that a natural unit of memory exists, such that specific conditions may be applied to one page and not another. In the Sigma 7, the size of a page is 512 words. These pages are indistinguishable in their usage except for the real memory sequence they are in, with special usage for the first page in this sequence (page zero).

Memory mapping is special purpose hardware to provide automatic relocation by pages. Associated with this is a table, the map, which specifies the usage to provide. Under this scheme, each page of memory is automatically, and with insignificant cost in time, mapped into a specific real page of memory. There is no requirement of identity, and thus the address space the computer is operating under need have no correspondance with space in the real memory, except in special cases. As a result, a task may be loaded by pages in such a way that the most efficient use is made of the existing memory, independent of the address space requirements of the task. There may be a one-to-one correspondance in regions of the map, as an alternate device to the pointers described in timesharing scheme 1. in the introduction.

Memory protection is that ability to specify the usage of certain pages of memory under specific conditions. For example, in the Sigma 7 operating in mapped slave mode, four modes of usage may be specified for each page, corresponding to the degree of access allowed. These access protects are: 0. Complete access allowed, 1. Write prohibited, 2. Write and execution prohibited, 3. All reference prohibited. While intended primarily as an aid in debugging and to provide security, the access protect may also be used to monitor the usage of memory.

Memory is divided into fast and slow portions. The fast portion, although strictly a storage medium, may be treated as a large set of

registers with identical capabilities. These "scratchpad" registers are normally treated as specific locations in memory, and thus all register-register operations are a natural extension of normal memory. Indeed, execution of the program may proceed from the registers. The net effect is that of a two address computer, where one address space is a small subset of the other.

There is usually one or more I/O processors (IOP). These are essentially the small computer, described in the introduction as the batch processing system, built directly into the large computer, an IOP is more powerful than a channel, in that it can provide simple operations, such as collecting data sequentially from several places in memory and combining them into a single record, or transmitting multiple records between the memory and a device.

These features describe the Sigma 7, but there are others which are extremely powerful, although not unique to third generation computers. The most important such feature is the inclusion of hard-wired subroutines. (It is not generally realized that all floating point operations fall into this category.) Since the operation is hard-wired, it may proceed relatively fast. Hard-wired subroutines are usually expensive, and are normally recognized by being an optional feature on the computer. The simpler ones may, however, be standard. The map and protection described above fall into the category of a hard-wired subroutine.

Stacks: it is frequently desirable in a program to be able to save information temporarily in a push-down list or stack, wherein the last item entered is the first item removed. The major use of such a feature is to provide dynamic allocation of storage. If all temporary modifiable storage is used in a stack, less storage is required than if

all storage were entirely static, its use is optimized, and recursive routines (those which may call themselves) become easy to write.

Conversion: the Sigma 7 has instructions for conversion between any two number systems, provided that the two are related by a weighted-number system.

Byte string: four standard and one optional instructions permit the manipulation of strings of bytes (character), with operations such as move, compare, translate, and search. These permit powerful text editing facilities.

Floating point: optional. Including single and double precision, this is useful whenever accuracy is unimportant, or the range or numbers involved is unknown. Used in scientific calculation, where complex operations are performed.

Decimal: optional. All operations are guaranteed good to 31 decimal digits. Numbers are carried around as binary coded decimal quantities. Used primarily for business applications, where the operations are simple and quantity of operations is the criterion. (It must be remembered that all operations on a single number must include the time spent to translate the number to and from a character string, and this may be appreciable.)

There is also an instruction (interpret) which is relatively powerful in juggling tables of non-numeric data, provided that they are of a specific format.

In addition, the Sigma 7 provides variable data bases, such as byte (8-bit), halfword (2-byte), word (4-byte), and doubleword (8-byte), and independent instructions to manipulate these data bases. In conjunction with these, a base addressing scheme is provided, such that any

indexing operation is automatically at the resolution of the data base.

There is also a set of instructions, called immediate instructions, which reference the registers only, using the 20 bit address field of the instruction as a signed operand.

In general, the Sigma 7 is a well designed computer, with a powerful instruction set, designed for the convenience of the programmer, rather than the engineer. However, it lacks a few features which would be very useful.

Specifically, I feel that there are two sets of instructions missing. These may be classified as "logical immediate" and "queuing". Logical immediate instructions would be an addition to the immediate instruction group described above, to include AND immediate, OR immediate, and EXCLUSIVE-OR immediate. The addition would be of great value for non-numeric operations, as currently it is necessary to provide a mask in core, even if needed only once. The queuing group would be harder to implement, since it would need to be a hard-wired subroutine. A complement to stacks, queues are cyclic stacks, of first-in-first-out nature, rather than first-in-last-out. However, it would seem no more difficult to implement queues than stacks.

Additional instructions which might be of use are list processing instructions, where a list and count are provided, and the list is scanned until some condition is satisfied.

As far as the Sigma 7 hardware is concerned, there is one major failing. This can be traced to the expectation that the computer would be used with a centralized monitor. As a result, there are a number of items of machine status which cannot be read directly, but must be represented by an image in core memory. These include the map, access

protect, and interrupt status. In a decentralized system, as under JANUS, it is not convenient to keep a centralized record of all interrupt status. As a result, while optional power failure interrupts are available for the Sigma 7, allowing the status of the machine to be saved if power is lost, it is impossible to save the status of the interrupts.

A special instruction is provided to load the map and protection. How much more useful it would be if the map and protect registers were inside the address space of the computer, subject to normal protection. Then normal addressing instructions could be used to modify these registers, they could still be protected against a malevolent user, but they would be readable.

8. STRUCTURE OF JANUS

JANUS is the name of both the system as a whole, and the resident part specifically. As a whole, JANUS consists of a base (resident) and a ring of tasks which fluctuate in size. JANUS operates on several levels concurrently (Figure 15). At the lowest level is the set of tasks, one of which is always current (active) or next. If active, the task is executing, performing its set of operations for a given period of time (time slice). While it is active, it may call upon JANUS resident to provide or save information or perform a specific operation, either explicitly or implicitly. At the end of the task's timeslice, the task is placed on a higher (interrupt) level, that of jobchanging. After performing any operations unique to the task at slice end, the task returns to the resident Jobchanger. The Jobchanger performs standard slice end operations, and then determines if a new task is ready to proceed. If so, the computer is set up to execute the new task, and then control is transferred back to the lower task level. However, just before this control tranfer, the Jobchanger determines what task is to be next after the current task. It may decide to start the Swapper, a routine on a higher level than either the Jobchanger or task. The Swapper will asynchronously interrupt the Jobchanger and active task as necessary in order to bring into memory parts of the next task. Thus there is a good chance that a new task will be ready to proceed when the current task's time is up. In addition, the resident portion of JANUS is

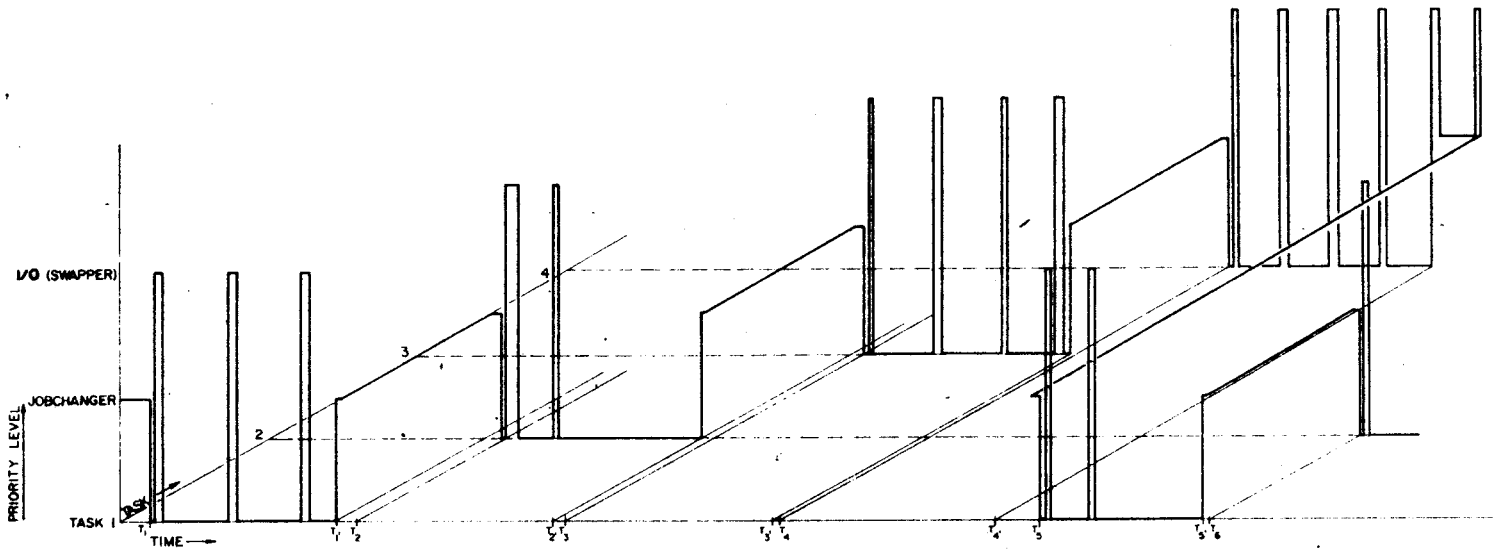


Figure 15. Typical world line of JANUS operation. The path of operation (heavy line) proceeds through a task 1 from T_j to $T_{j'}$, interrupted by the Swapper. At each T', it passes through the Jobchanger, which performs operations (interruptable by the Swapper, as between $T_{4'}$ and T_5), and returns to the next task in the ring. Each interrupt by the Swapper is used to initiate a new RAD operation, or finish an old operation. These RAD operations are used to ready the next task.

asynchronously handling a number of specialized interrupts, which may be fanned out to specific routines. During the timeslice of a task, the task may request in a specific manner that a portion of itself be dedicated and be attached to a realtime process (Figure 16). This resident routine may then also operate asynchronously to the timesharing, and at a higher level. The dedicated portion may at any time signal the task that a specific condition has occurred. Even if the task has put itself on wait (inactive) status, this signal is sufficient to cause the task to reenter the ring as an active task and take the proper action with respect to the condition.

A task may, at any time, provide JANUS with the name of a subtask to start (Figure 17) or delete from the ring of tasks. Any task may signal a subtask, or a parent task, and is responsible for destroying all subtasks before exiting itself.

Thus, at the task level, JANUS does nothing for the task, in the sense of performing a high level operation. Instead, it performs a bookkeeping operating to keep track of various system resources, and allocates these to tasks on request if they are available. These bookkeeping operations may make higher functions possible, such as by connecting an interrupt routine to the I/O interrupt, but JANUS will neither do the I/O operation itself, nor check the legality of the request. The assumption is always made that any task which is performing such an operation is doing its own internal bookkeeping, and has had the specific device assigned to it before it proceeds to attach itself to that device through the interrupt.

At a higher level, JANUS performs those functions common to all. For example, if an I/O interrupt occurs, JANUS will asknowledge the

Figure 16. JANUS form of control structure. Operations are divided into three parts; mapped slave, mapped master, and unmapped realtime. Paths of communication between the three parts are shown by arrows.

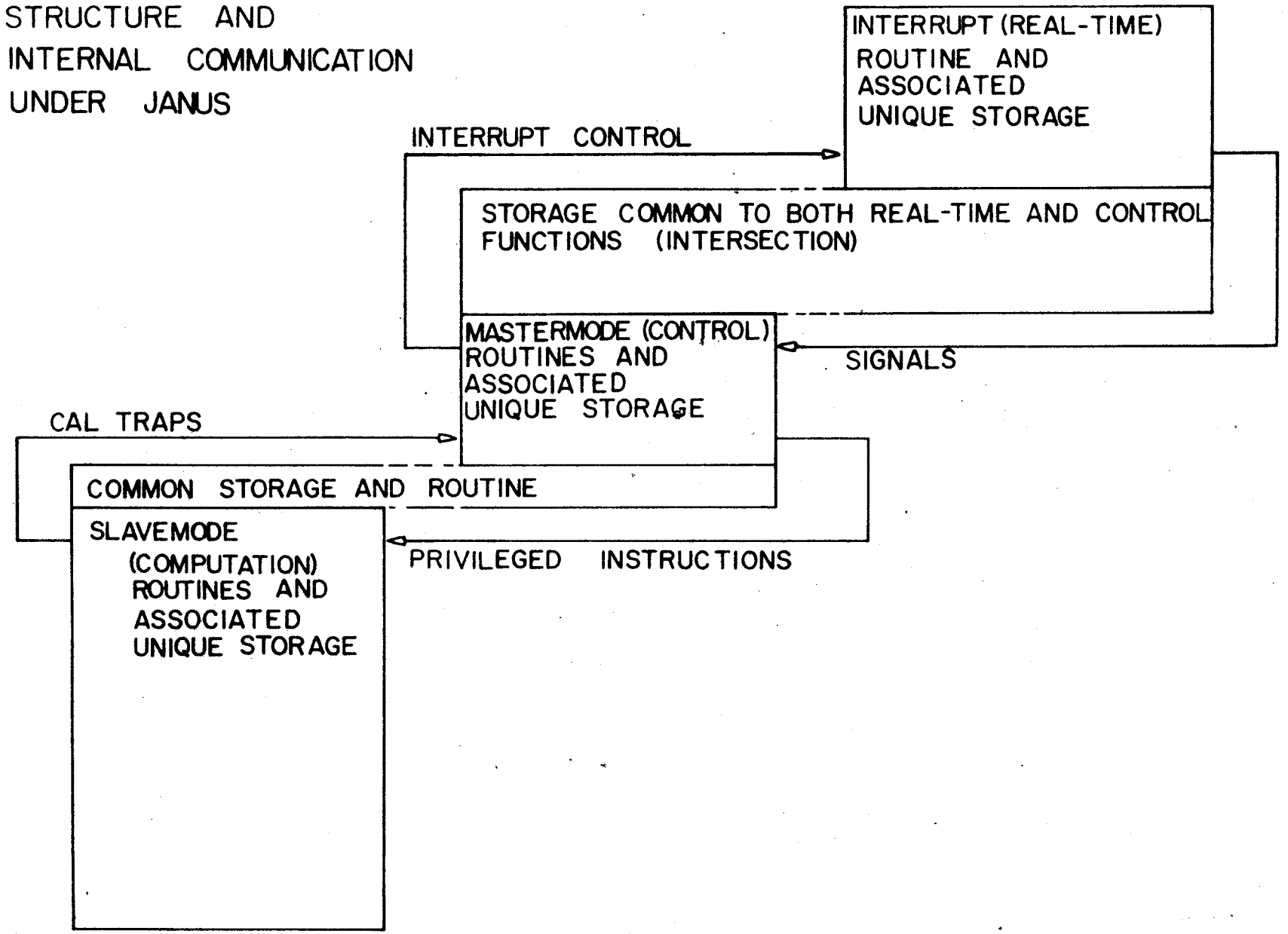

STRUCTURE AND
INTERNAL COMMUNICATION
UNDER JANUS
INTERRUPT (REAL-TIME)
ROUTINE AND
ASSOCIATED
UNIQUE STORAGE
INTERRUPT CONTROL
STORAGE COMMON TO BOTH REAL-TIME AND CONTROL
FUNCTIONS (INTERSECTION)
MASTERMODE (CONTROL)
ROUTINES AND
ASSOCIATED
UNIQUE STORAGE
SIGNALS
CAL TRAPS
COMMON STORAGE AND ROUTINE
SLAVEMODE
(COMPUTATION)
ROUTINES AND
ASSOCIATED
UNIQUE STORAGE
PRIVILEGED INSTRUCTIONS

Figure 16.

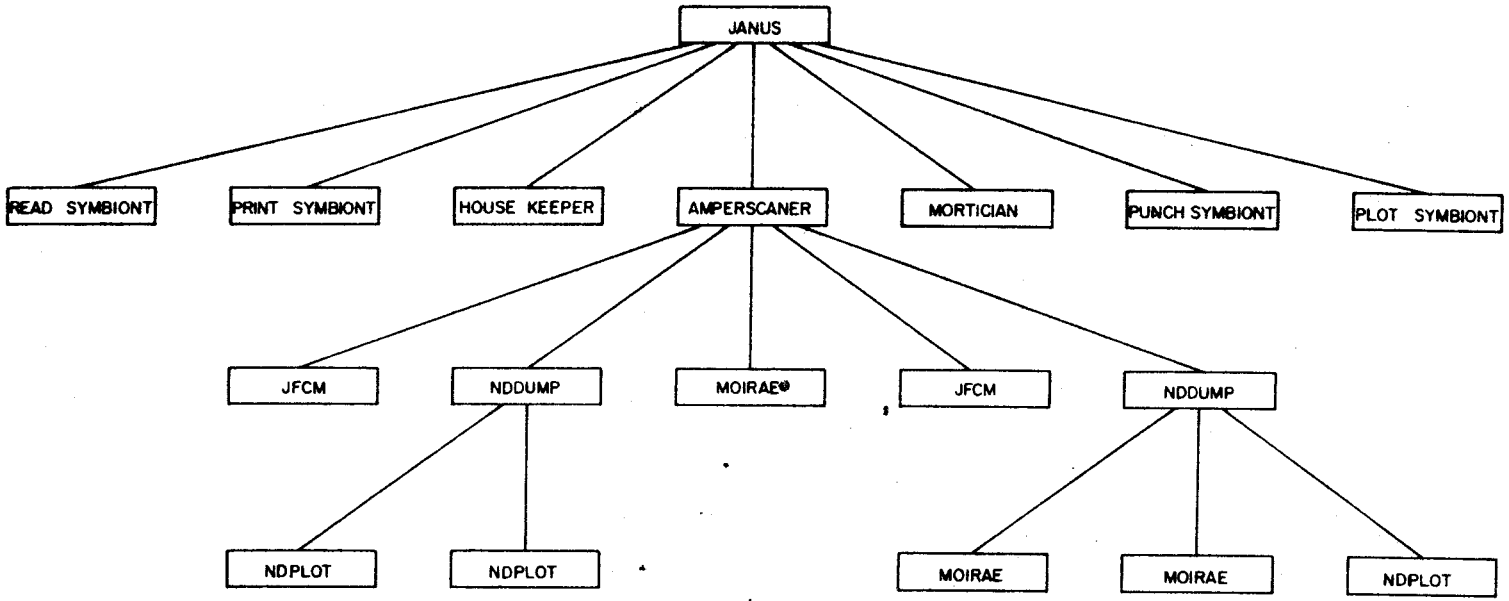


Figure 17. Tree structure of tasks. The first rank of tasks below JANUS are the system tasks. One of these (the Amperscaner), can start subtasks, which may start subtasks of their own. These subtasks may be identical copies of tasks which may be started by the Amperscaner, or they may be unique.

interrupt in order to discover which device caused it, and will search a table of active devices for this device. When the device is found, associated with it will be an address of a routine in dedicated memory, to which control is transferred in order that that routine can service the interrupt. This operation is necessary since several tasks may be doing independent I/O operations, and the computer has only one I/O interrupt, to which all such processes are connected.

One of the resources which JANUS keeps track of is space on the Rapid Access Disk (RAD). The disk is divided into a collection of diskpages, each of which is the same size as a page of memory. The hardware of the Sigma 7 treats the slower core memory as an extension of the fast scratchpad memory: JANUS treats the disk as a slow extension of core memory. Indeed, a task is an ordered set of diskpages (diskfile). Just as a set of operands from core memory may be in the registers during a computation, so also a set of diskpages may be in core memory during the execution of a task. A simple task, which is limited to only a few pages, may be entirely in core memory each time it is active, while a large task, one using the full address space of the machine, would do its own demand paging, such that a page was not in core until referenced and when no longer used would rapidly retreat back to the disk.

JANUS treats resources in two categories, common and unique. A unique resource is one which is asked for by name rather than by generic type. An I/O device is a unique resource, while a diskpage is common, since all diskpages are interchangeable.

Memory pages are also interchangeable, with one important exception, that of pages which may be dedicated. Because JANUS uses the map to relocate around dedicated "islands" in core, the map is a resource shared

by all tasks at the task level, and all interrupt routines must run outside the map. All unmapped code and storage must be loaded into a specific page of real memory, since there is no automatic relocation. Hence, certain pages of a task must be flagged as absolute pages, in that they contain unmapped storage. In this case only does memory become unique, although any page may have a unique set of attributes.

While all unique resources must be kept track of in resident tables, this is not true of many common resources. The best example of this is, again, diskpages. While the disk consists of hundreds of diskpages, the load of usage is not normally critical. A list of twenty diskpages is nominally sufficient to support at least one cycle around the ring, especially since diskpages are allocated and freed with equal frequency. JANUS keeps a resident stack, of twice the nominal size, half full, permitting transfers in either direction.

All other free diskpages are kept in a list in a system task, the Housekeeper. The Housekeeper's function is just as its name implies, to tidy up the resident. Any time the resident portion needs a specific function which it is not profitable to keep resident, it calls upon the Housekeeper task, using the standard JANUS mechanism of timesharing, such as signals. Without this feature JANUS would have a resident portion twice as large as it does. The Housekeeper and other system tasks will be described at length in Appendix B, as well as the specific JANUS mechanisms.

9. ADDRESS SPACES

Vital to an understanding of JANUS is a knowledge of the addressing scheme used (Figure 18). Each location in the computer is unique in that it has a fixed address associated with it. The contents of the location are referenced by a reference to the address. In any normal form of program generation, either via assembler or compiler, it is possible to assign a name to a location. This symbolic name has a value associated with it, which nominally is identical to the address of the location. However, in JANUS, or indeed any mapped system, there is no longer any correlation between the address and a memory location, except within page boundaries. Since there is no longer any requirement on identity between address and location, except when a reference may be made, many restrictions are lifted.

The most important of these is uniqueness. It is no longer necessary that different programs use different parts of the available address space. Instead, each uses a unique version of the same address space. Each task normally will execute in an address space orthogonal to all other tasks (exceptions will be noted below). Thus tasks A, B, and C may reference symbolic names X, Y, and Z, respectively, each of which has an address of 10,000, but each of which is a unique location, containing a unique quantity.

Let us consider what information may be required to assemble a task to use under JANUS. In normal generation of code, the contents of

Figure 18. Examples of address space usage, including files. The vertical columns are independent task address spaces, resting on the JANUS block common to all tasks. Two tasks are shown, with TASK 2 being a subtask of TASK 1. The two tasks have one page in common (TCP2) which appears in different parts of the two task address spaces. The two tasks share a driven stream file, which is also referenced from different parts of the two address spaces. It differs from the TCP2 usage, however, in that the file driver (TASK 1) may be several pages ahead of the file receiver (TASK 2). The files are a collection of diskpages, each of which may be used as the same address space page. Only one page of a file is actually within the task address space at any given time, however. Files may be linked internally, or may be linked through a table residing within the task, as is shown in the keyed file.

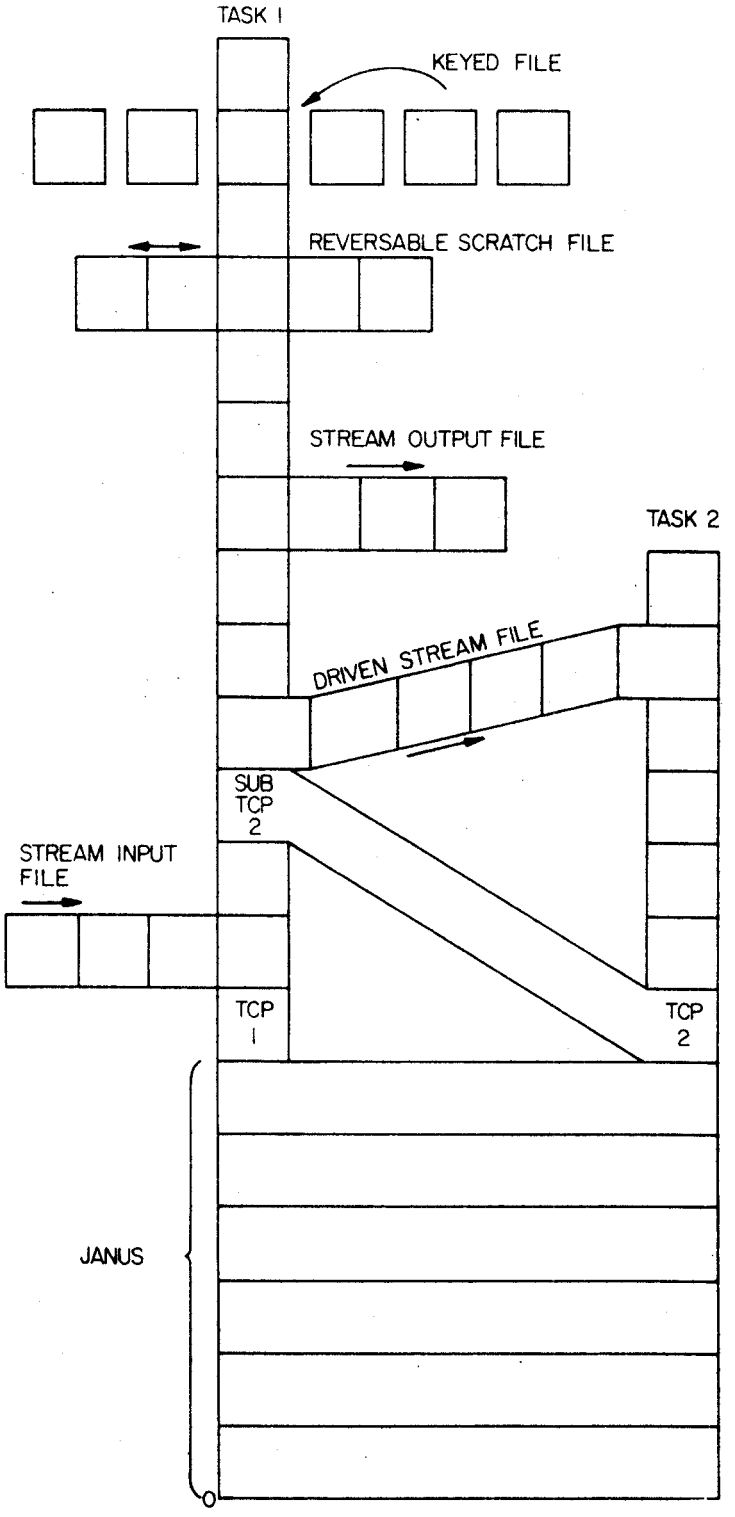

TASK 1
KEYED FILE
REVERSABLE SCRATCH FILE
STREAM OUTPUT FILE
TASK 2
DRIVEN STREAM FILE
SUB TCP 2
STREAM INPUT FILE
TCP 1
TCP 2
JANUS

Figure 18.

successive locations are generated, and addresses are incremented. It is therefore necessary to have a location counter (LC). (This is the basis of all assemblers.) It is conceivable that, under certain circumstances, it is desirable to generate code to be loaded in one place, but capable of being moved to a specified place for execution. This would, for example, be of use in overlay programming. We may thus readily convince ourselves of the utility of both a load location counter (LLC) and an execution location counter (ELC). Many assemblers have only a LC, some have both LLC and ELC.

Under JANUS, each task is normally in the same address space as all other tasks, implying an overlay structure. This is, however, no problem, since each task is usually the result of a separate load. A problem does exist, however, with respect to the unmapped address space used by interrupt routines. All unmapped storage is in the same address space, independent of the spaces of the controlling tasks. In addition, in a task it is necessary to have intersections between the mapped and unmapped areas.

It may be seen that it would be desirable to have mapped and unmapped location counters (MLLC, MELC, ULLC, and UELC). Furthermore, it would be desirable in a decentralized system, such as JANUS, for the system loader to be capable of relocating unmapped code completely independently of the mapped storage, optimizing the usage of real core for dedicable pages.

As yet JANUS is not capable of these operations due to the lack of such a flexible assembler-loader. It is necessary to write tasks using the SYMBOL assembler provided by SDS. Since this has only an ELC ($) and LLC ($$), it is necessary to perform certain coding tricks to generate

mixed mapped and unmapped code. The most distasteful of these, for aesthetic reasons, is the necessity of allocating unmapped pages for interrupt routines before assembly, rather than at load time, making JANUS much less flexible than it would be, given a good assembler-loader. However, JANUS is still sufficiently useful to be adequate for many operations. In order to include both mapped and unmapped storage, as well as intersections, it becomes necessary to use one counter for a mapped location counter (MLC), the other for unmapped (ULC). Since a task has a unique address space, it is necessarily loaded as one block, without overlays. This suggests that the SDS SYMBOL's LLC be used as MLC; thus ELC becomes ULC. However, one must take care that, in setting ULC so that it will track the unmapped addresses correctly, the relocation of the task as a whole is included. The alternative is to define each unmapped symbol as the mapped location plus a suitable bias.

While all tasks are normally independent, there are exceptions. The JANUS concept permits tasks to communicate. This may be through the resident, common to all tasks, or it may be at a higher level of abstraction.

Higher level communication between tasks will generally mean that at least some address space is common to both tasks. Further, any task may start subtasks. In these cases, it will generally be true that the master and subtasks will be generated as one load. This load will then be studded with tasks, which need overlay each other at least in part (see description of task control page below). In this case, using SYMBOL, LLC must be the LC for the whole task, while the ELC is used for each task. As a result, unmapped storage has to be referenced

without modifying ELC or LLC, forcing one to use references of the form of LC+bias.

Since in this scheme the tasks may be almost completely independent, it may be seen that the concept described above, with respect to ELC and LLC, can be extended in an open-ended process to include a unique ELC and LLC for each address space which might be used by a task. The case of unmapped and mapped counters can then readily be extended to include UNMAPPED, $MAPPED_1$, $MAPPED_2$,..., $MAPPED_N$, as well as multiple computers using the same memory, even when they have different word size. As far as this is concerned, JANUS is in effect a two-computer system, where the unmapped computer is independent of the mapped computer, with different usage. It is possible to have not only an open-ended set of mapped LC's, but also an open-ended set of processor LC's. These would include the IOP. For flexibility, it should be left for the programmer to define the symbol he wishes to use for each LC.

10. JANUS AND BPM: A COMPARISON

Perhaps one of the best ways to evaluate a specific computer operating system is to compare it with another operating system. Let us therefore compare JANUS with one of the operating systems provided by SDS, the Batch Processing Monitor (BPM) [13]. This must be a qualified comparison: whereas JANUS is a realtime oriented system, with background capabilities, BPM is, as its name implies, primarily a background oriented system with some realtime capability. The SDS BPM has been selected because it is the most advanced system yet released by the manufacturer. It is as valid a comparison to make as any other, since there is no other computer system like JANUS.

Consider first that there are three basic means of acquiring an operating system. The easiest and fastest method is to use the manufacturer-supplied operating system with no changes not supported by the manufacturer. For most cases, this is the best approach, as the manufacturer will continue to improve the system for the customer. The limitation is that the user must live with the system supplied, and with any inefficiencies in its operation.

A second alternative is to modify an operating system supplied, to introduce nonstandard functions, or to counteract some inefficiencies in operation. This is the most attractive approach when nonstandard functions are desired, because one is building upon a working system, and the implementation is speeded. This approach turns into a dead-end

easily, because of a hidden fault. By introducing a nonstandard function into a system, the system itself becomes nonstandard, and will no longer be supported by the manufacturer. The customer is forced into one of two paths--either he is confined to an operating system which becomes more and more obsolete with time, as the manufacturer upgrades and improves the system or even replaces it with more powerful systems, or he must be prepared to repeat his work each time a newer version of the system appears. In either case, the consequence of the initial expediency becomes an unending source of annoyance at best. The solution is thus satisfactory only in a stable environment.

The third alternative is to build a completely independent system as in JANUS. In doing so, one cuts oneself off from all support by the manufacturer and sets himself an extended programming effort, but in return ends up with a system optimized for the intended usage.

As a manufacturer-supplied system, BPM must of necessity be extremely general in applicability, so that the first situation described will apply to most customers. As a result, there is a tendency toward a comprehensive inclusion of all possibly desired function. Two conditions result: the monitor is large, requiring extensive core and disk storage, and slow, requiring a large partion of time in determining which function is desired, and in doing that function.

In addition, SDS manufactures a second computer (the Sigma 5) which is identical to the Sigma 7, except that it lacks the map, access protection, and certain instructions (notably the byte immediate and convert instructions). As a result, all Sigma 7 software is downward compatible, able to run in the Sigma 5 also. No advantage is taken of the added power of the Sigma 7.

Because of these differences, it is difficult to compare specific features of JANUS and BPM. However, general comparisons in various classes of usage are possible.

Perhaps the most striking difference between JANUS and BPM lies in the contents of resident storage. JANUS contains a minimum of resident storage, devoted to the minimum number of primitive routines necessary to resource management. Conversely, BPM contains many high level functions, most of which are a convenience rather than a necessity. Indeed, many of these functions could be deleted, and instead provided by library routines. For example, both JANUS and BPM keep track of time of day in resident storage. Under JANUS, tasks are informed where to find the information if they ask for it. Under BPM, there is a special resident routine which may be called, and which, after extensive checking as to the form of the request, physically transfers the data to the area specified by the user. Again, BPM provides two files for compressed I/O (M:CI and M:CO, used for Compressed Input and Compressed Output, respectively). Use of these files causes automatic translation between compressed code (where text is compressed by replacing strings of blanks with a blank count, all bytes are compressed to 6 bits, and punched onto cards in binary), and BCD, through resident routines. Anyone desiring to use these functions could as easily call upon a library routine, with no loss in speed, and with an increase in available core if they were not used.

Under BPM, all I/O is done through files, each of which has associated with it a Device Control Block (DCB), which is 90 words long. Since there are 17 resident DCB's, 1.5k of resident core is dedicated to this storage, much of which is again superfluous. (The Basic Control

Monitor, an earlier SDS monitor, has 6 word DCB's.) Included are such parameters as the file name, tab settings, file keys, etc. Many of these parameters again have no real justification for inclusion in the resident monitor. If such quantities are used in JANUS, they are unique to a task, carried around with it, and not a system convention.

Let us now turn our attention to resource management. In this respect, JANUS and BPM are so divergent that little comparison can be made. While JANUS freely allocates resources to tasks upon request if at all possible, it does not set any restriction on usage. By contrast, BPM is extremely paternalistic, thereby constraining permitted operations. For example, BPM requires that each I/O operation reference the file associated DCB, which includes an account number (presumably for billing purposes), a password (presumably for file security), expiration date, and read and write account numbers (again presumably for security). Under JANUS, any task which references a file knows the name of the file, and tasks which have no need for the file do not bother with it. If the file is write protected (e.g., a library disk file) and is accessable to users, the task provides the necessary security. This is a much simpler (and faster) process.

Both JANUS and BPM have functions to allocate and free core memory. However, while JANUS automatically allocates free pages for a task during a timeslice and permits a task to get or free diskpages, the BPM both permits and requires these operations of the slave-mode user. (In JANUS tasks with demand paging, the function of demand allocation is easily provided by the demand paging algorithm, and if desired the function of freeing pages within the address space is easily implemented.)

Realtime or foreground processes are exactly what JANUS is

oriented and designed for. BPM also has a foreground facility, and monitor functions such as M:MASTER, which permits a user to enter master-mode if his is a foreground job. A foreground user has two options under BPM. He may use the hardware structure of the computer for speed, but act independently of the BPM. In this case, he is not permitted to use any of the BPM functions, such as I/O, from his interrupt routines. In order to do I/O from the interrupt routines, all interrupts must pass through BPM, so they can be monitored. The return from such an interrupt routine requires three RAD operations, and thus 80 milliseconds. This time would be excessive for most experimental applications.

One of the most important aspects of a realtime system is security--not in the conventional sense of privacy of data, but instead in terms of the freedom from the possibility of an individual introducing a program which destroys the system, and the realtime along with it. If this can happen, intentionally or not, no one performing a realtime process will trust another to use the computer in the background, especially if the process involves accumulating data over a long period of time. If this situation can occur, a multi-user system, no matter how elaborate, is useless.

Under JANUS, one invokes a specific task from a user library. (While it is possible for tasks to be loaded from card decks using a special task, this is understood to be strictly a debugging feature, and not for general usage.) No task is added to the library until it has been thoroughly debugged to the satisfaction of all concerned. While a system-destroying error could lurk in any task, it is a rare occurrence. No task can be told to destroy the system, nor will any

task permit the execution of a user program under its control which can cause the destruction of the system. Such operations are not allowed under the JANUS design philosophy.

By contrast, BPM is not at all safe. Programs can be written which will cause the BPM to overwrite itself in any number of ways, and thus destroy itself. A user can declare himself as a realtime process, enter master-mode, and do untold damage. It is even possible to provide a set of control cards which cause all system files to be irrevocably freed, thus destroying the system. Indeed, this is sometimes the only way that some jobs may be done, using limited resources. For example, in using BPM with a 1.5 Mbyte RAD, an operation as simple in appearance as assembling the FORTRAN compiler from a magnetic tape requires so much disk storage that the system must be destroyed to accomplish it (as an aside, this "simple" process necessitates the use of approximately 470 distinct cards, each of which must be correct and in the correct sequence).

As a final comparison, any time it is considered desirable, the JANUS JBCM/JFCM tasks could be upgraded, by the addition of suitable additional functions, into a JBPM, or JANUS Batch Processing Monitor, without, of course, the realtime or other destructive characteristics of BPM. In that case, JANUS could easily timeshare several JBPM's simultaneously, just as it now can do for the JBCM/JFCM.

11. MEASUREMENTS

Since computer logic signals have two values, it is possible to connect a logic signal to an electrical meter and directly measure the fraction of time the logic signal is in one state or the other. Calibration is relatively simple: with the logic signal in the "0" state, the meter movement is adjusted to read "0" exactly, in spite of the probable existence of small currents. If we now switch the logic signal to the "1" state, it is possible to adjust a variable resistance in series with the meter to cause the meter to read exactly full scale--if necessary, these operations may be iterated for a higher degree of accuracy. Furthermore, if a meter is chosen which has a high sensitivity (such that a low load upon the signal is effected) and a 0-100 scale (such as 0-100 microamperes) it is possible to read the average time that the logic signal spends in the "1" state directly in percent, and short term fluctuations are integrated out by the meter. These measurements are accurate to the accuracy of the calibrated meter, nominally five percent.

By a judicious choice of logic signals, it is possible to measure various operating conditions, and observe the effect of varying various parameters. This was done on the MSU Sigma 7.

As timeshared JANUS runs under the map, and realtime JANUS runs unmapped, the fraction of time spent in mapped mode is a measure of the relative weight given each mode (note that the Swapper is an unmapped realtime process). Consequently a meter was attached to the signal

MAP, a level set while the computer is in mapped mode.

Similarly, under JANUS, slave mode is used for all problem solving (production), master mode is used for all control (overhead). As a result, the signal NMASTER was used to monitor actual production operation.

JANUS uses the wait instruction only once--in the Jobchanger, under the condition that no job is ready to proceed. The associated signal HALT provides a measure of the completely nonproductive overhead.

A fourth signal, PRE1, while not actually useful for system parameters, does provide a measure of the efficiency of code. The signal is raised once and only once for a fixed duration, in the course of each instruction execution. By choosing an instruction with a well-defined time, such as branch (1 microsecond) and providing a timing loop, it is possible to calibrate the meter directly in instructions/second.

The most important factor, in a realtime oriented system, is the time required to service an interrupt. For other than clock interrupts, which are of lowest priority and which are frequently inhibited, the time required to service an interrupt is hardware, rather than software, limited. This time is 6 microseconds + the time required to complete the instruction interrupted, if there is no higher priority interrupt active: 6 microseconds + the time required to complete servicing all higher priority interrupts if any are active. While there are patholigical situations which do cause interrupt inhibition, these are demonstrably rare.

Similarly, 4 microseconds are required to exit from a realtime interrupt process. (Compare with the SDS Batch Processing Monitor [13], where an exit from a realtime process may require as much as 80

milliseconds.)

Actual metered measurements are more difficult to make, because of the rapid fluctuations the system undergoes with time. However, it is possible to provide quantitative numbers in certain relatively stable situations, notably those which are not I/O limited.

1. Single task active, requiring no swapping. An overhead of 2% has been measured, and is entirely due to timeslicing the single task to provide an entry for an asynchronously activated task.

2. Multiple tasks active, requiring no swapping. Times are 8%, or just four times those measured in case 1, attributed entirely to the fact that the time quantum is four times longer for the single task case.

3. One task active, performing demand paging for pages not in core. Times vary from 5% overhead for well behaved programs to as much as 50% for some cases, notably processors (such as the JBCM/JFCM Loader) which are required to physically move multipage blocks of storage around within the task's address space.

4. Two tasks active, swapping required. Times vary from 5% to 20% overhead, the difference between case 3 and case 4 being attributable to the high probability that execution of one task is proceeding concurrently with the swapping for the other.

5. Three or more tasks, all requiring swapping. In general, overhead approaches 100% in this case. This problem and a possible solution will be treated in more detail in Chapter 11.

6. Realtime scope display. These scope displays are unbuffered, and must be refreshed periodically by the computer. They may be implemented by computing the display points from a small data base (requiring

little core but much computer time), by having a large data base initialized in such a form that it may be written directly out to the scope (requiring more core but less computer time), or a mixture of the two. These displays are normally refreshed 20 times a second to avoid screen flicker disturbing to viewers. Two cases are of interest, one involving a static display (initialized storage), the other a dynamic display (computed display).

A. Static display. The data analysis code MOIRAE, displaying 4096 points, plus three dynamic vectors and several characters--60% of the computer time is used to generate the display.

B. Dynamic display. The game code SPACEWAR, displaying 100 static and 500-1000 dynamically generated points (position computed from orbit equations as a function of real elapsed time)--10-30% overhead.

It is readily seen from these examples that multiple displays would generally tie up the computer entirely, and the need for self-buffered displays is indicated (such as storage scopes).

12. CONCLUSIONS

JANUS is finally operational, and is used for significant parts of each day. The principal delay to fulltime operation has been the lack of systems programmers available to introduce additional capabilities to the JANUS system which have not as yet been implemented (examples of unimplemented capabilities are the lack of teletype and magnetic tape handlers in the JBCM). The other difficulty is the necessity of recasting existing programs (especially realtime data acquisition) into a timesharing form. These problems are, however, being met, and JANUS will approach greater permanence as these implementations occur.

One of the most striking conclusions that can already be drawn from observed operation of JANUS concerns demand paging. I feel that JANUS has conclusively demonstrated the value of demand paging in a batch processing configuration as a memory expansion device. By the use of a relatively inexpensive map and RAD, one can simulate the existence of a much larger, and more expensive, core memory. Since in any installation the majority of problems will fit into core, the use of demand paging introduces no essential overhead. For the few problems which do not fit, there must be a mechanism provided for execution, either through overlays, job-chaining, or through some other means. Demand paging extends memory--no additional knowledge is required of a user in order to demand page a large job. The efficiency of the job execution is his problem, and it is relatively easy to explain how to optimize execution.

Any other method requires the user to introduce a large number of control cards, and to have a thorough understanding of the structure involved (requiring that more systems programmers be available to answer user questions). The other side of the problem, system programming, is of comparable difficulty under either method. In general, resident buffers are required under demand paging, but the necessary amount of space in the monitor may be provided by being able to delete control card processing relative to overlays, and by deleting core allocation functions. A loader capable of segmenting overlays is not necessary, and the effort required for its generation and maintenance could be directed elsewhere.

However, the demand paging currently used in JANUS is impractical for use in a timesharing environment where more than two or three tasks are active. The difficulty with the JANUS implementation is the lack of sufficient history to permit adequate judgements as to usage of demandable pages. As JANUS permits each task to perform its own demand paging, it is a relatively easy matter to test different demand paging algorithms by modifying some standard task. This has not been done as yet, because of a lack of available computer time, and the existence of higher priority problems. The problem, and a possible solution, may be simply stated.

Under JANUS, all memory or usage exists for only one timeslice. In demanding a page not currently in core, jobchange must be effected. Only those pages referenced the last time may be brought into core the next time. If, as is frequently the case, three successive instructions reference three different pages, in a hard-swapping environment only one instruction will be executed each timeslice. By the time the third

instruction is executed, the first is forgotten. Instead, what is necessary is to keep a record of each page of a task, with a memory of how recently the page was referenced. The demand paging routine, in addition to setting the "used this time" bit in the task control page, would also have to set the corresponding entry of this table to "referenced this time". At slice start, the task must reset each entry back by one. At task end, all entries must be compared, and the N most recently referenced pages (where N is to be empirically determined, but probably about 10) must be flagged in the TCP as "used this time". The significance of this now changes from "used this time" to "used recently enough to justify its presence". An alternative, and more suitable, method would be to flag up to N pages as above, but ignore all references which occurred more than M timeslices ago, where M is also to be empirically determined.

Similarly, the usage of core memory pages under JANUS does include a two-bit (four level) memory as to how recently the page was used. This is an inadequate memory, but unfortunately is so deeply imbedded in JANUS that it would be extremely difficult to change. The point would be well to remember, however, in future implementations.

(Crude measurements made since the bulk of this thesis was written indicate that this approach has definite merit. With the case of four identical tasks operating concurrently, total running time was within 10-20% of the time required to run the same tasks serially, using this method. Using the previous approach, times differed by 100-200%.)

The JANUS capability of timeshared monitors, each capable of dedicating realtime processes as necessary, has been successful. Many system functions may be greatly streamlined, to produce an efficient system

task. While suffering from the necessity of using two independent address spaces, one of which (unmapped) is unique, it is possible to overcome the problem by the construction of a relocatable task loader, provided that the multiple address spaces can be referenced in the process generating the relocatable task. This will doubtless be a limitation in the future, but is not yet a problem.

The use of a memory map is a great advance in computer design, making possible demand paging and therefore, more efficient use of core memory. In all probability, more and more computers will have a map available. As the demand paged memory provides one of the most flexible file systems available, I foresee that available address spaces will increase to a large value, on the order of 100,000,000 words and more, even though it would be impractical to have actual core memories of this size. This will be true especially of small, non-timeshared computers, such as are used for batch processing, research, and process control.

Finally, JANUS works as defined. There is room for improvement, but more study of specific inefficiencies is required to optimize the computer usage. It should be possible to make JANUS as efficient for many active tasks as it is now when there are only two or three active.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. On-line Computers for Research, Nucleonics, January-March 1967.

2. Dictionary of Classical Antiquities, Oskar Seyffert, Meridian Books, 1957.

3. Users Manual: Brookhaven Scheduler for Data Terminal Network, B.J. Shepard, April 10, 1968.

4. Timesharing Systems Manual, General Electric Corp., May 1966.

5. An Advanced Computer-Based Nuclear Physics Data Acquisition System, H.L. Gelernter et. al., Nuclear Instruments and Methods 54 (1967) 77-90.

6. Initial Operating Experience with the Yale-IBM Nuclear Data Acquisition System, M.W. Sachs et. al., Internal Report No. 32, Wright Nuclear Structure Laboratory, Yale University.

7. Time-sharing: A Computer for Everyone, Jeffrey N. Bairstow, Electronics Design, April 25, 1968.

8. System/360 Model 67 Timesharing Systems Preliminary Technical Survey, IBM form C20-1647-0.

9. IBM System/360 Model 67 Timesharing System Technical Summary, IBM, August 18, 1965.

10. IBM System/360 Operating System PL/1 Language Specifications, IBM form C28-6571-4.

11. A Scheduling Philosophy for Multiprocessing Systems, Butler W. Lampson, Communications of the ACM 11 (May, 1968), 347-360.

12. SDS Sigma 7 Computer Reference Manual, November 1967.

13. SDS Sigma 5/7 Batch Processing Monitor Operations Manual, January, 1968.

14. SDS Sigma 5/7 Basic Control Monitor Reference Manual, May 1968.

15. Basic Language Reference Manual, General Electric Corporation, May, 1967.

General References

A new remote-accessed man-machine system, General Electric Corporation, (from Proceedings, Fall Joint Computer Conference, 1965).

SDS Sigma 5/7 Batch Processing Monitor Reference Manual, July, 1968.

APPENDICES

APPENDIX A.

Glossary of Terms

Absolute--any datum (including instructions) the value of which is independent of its location in a storage medium.

Active--the word is used in two different senses, depending on context. A task is active if it is not on wait status, when discussing scheduling. Also, a task is active (current) if the current time-slice is assigned to it.

Address space--the full range of addresses which may be accessed.

Algorithm--the specific procedure used to implement a given process.

Associative addressing--a method of referencing a datum by content rather than by position. The datum consists of a key (the content referenced), and the associated information.

Background--a timesharing technique in which programs can be run concurrently with realtime processes in those periods when no realtime activity is required of the computer.

Byte--a unit of data, consisting of 8 bits. A byte is identical to one character.

Channel--a means of initiating a single I/O data transfer which then automatically runs to completion without needing further program intervention.

Clock interrupt--the Sigma 7 has two standard realtime clocks, one "ticking" at 500 Hz, the other at 2 KHz. These can be used to time realtime processes.

Data--a set of information, other than instructions, used in performing a process.

Dedicate--changing the usage of a resource from general availability to a specific usage. For example, under JANUS, a page of a task may be dedicated into a page of physical core memory, such that the physical page is used only for that task, rather than being available for all tasks.

Double precision--the use of two words of computer memory to maintain a single datum. The larger size permits a greater information content to be provided than under single precision (one word).

Foreground--a timesharing technique in which realtime processes can be run concurrently with other processes, interrupting the background as necessary.

Honest task--one which is careful to manipulate only those resources which belong to it, and which does not indiscriminantly affect those resources which belong to other tasks.

H/t disk--(head per track). A diskfile where a read-write head is positioned over each track, thereby requiring no head movement on an I/O operation.

Inactive task--a task which can temporarily perform no operation because it is waiting to be synchronized with a realtime event, such as the completion of an I/O operation.

Index register--a hardware feature permitting automatic arithmetic operations during a reference to an address, such as the addition of a displacement to a base address.

Interrupt--a hardware feature which permits the computer to change states in a rapid fashion--interrupting the execution of one process in order to execute a second process.

Intersection--an area of storage common to two or more address spaces, capable of being references by different names from each address space.

I/O--the abbreviation for Input/Output; the process of transferring data to and from the computer.

Location counter--a datum within an assembler to keep track of the address of each datum generated (including instructions) relative to some specific point such as the beginning of the assembly. Used to generate relocatable binary code for the loader, and to define addresses.

Map--a feature permitting the automatic translation of an effective address to the real address used to reference a storage medium. See also relocation.

Mask--a specific bit pattern used in performing logical operations under computer control.

Master--the mode of computer operation wherein all operations are permitted. Master mode is used normally for control operations.

Memory mapping--the process of using a map to translate addresses in the computer core memory.

Monitor--a program designed to supervise the usage of the computer in executing a problem, and which provides the control functions necessary and sufficient to that problem, or to a set of problems.

Overlay--a method of generating a program for execution in such a manner that independent subsets of the program may alternately be executed within the same address space, and be capable of referencing common areas.

Page--a natural unit of memory which is machine dependent. In the Sigma 7, one page contains 512 words.

PL/1--a relatively recently developed high-level programming language, containing many of the functions provided by FORTRAN, ALGOL, COBOL, and other special purpose languages in a fashion that permits the statement of a problem in a manner much more powerful and flexible than in any single special purpose language.

Pointer--a datum indicating the location of a set of data, referenced instead of the data set itself.

Realtime--a realtime process is one which is initiated asynchronously with respect to the normal flow of machine operation. A realtime process is normally associated with an interrupt.

Register--a piece of hardware, normally consisting of an ordered set of bi-stable elements, capable of operations in addition to a storage function, such as arithmetic or logical operations. The time required to access a register is much less than that required to reference core memory.

Relocation--the capability of a datum to have, in addition to a value, information as to some other quantity to which the value is related.

Resident--that portion of a monitor or supervisory system which is kept permanently in core memory.

Slave--that mode of computer operation capable of being completely controlled as to permitted operations. Computational functions are permitted, but control functions are not. A mechanism is provided for a slave-mode process to request of the monitor that a specific control process be performed.

Task--a set of processes capable of being timeshared as a unit, independent of any other usage of the computer, and containing those monitor functions necessary and sufficient to its operation.

Task control page (TCP)--a block of storage under JANUS which is always located in specific addresses in the address space of a task. This contains the status of the task, including trap and memory page usage. Also referred to as the state vector for the task, and is unique to the task.

Two-address computer--a computer where each instruction specifies both a source and a destination, as opposed to a single effective address, in addition to a process to be performed.

APPENDIX B.

JANUS Reference Manual

Some features of JANUS are of interest primarily to programmers who intend to build tasks to operate under JANUS. As has been noted previously, there are no aids to building a task currently available. While many of the computational functions desired of a task may be written in a higher language, such as FORTRAN, it is still necessary that all monitor and control functions be coded in assembly language. This requires an understanding of specific functions available in JANUS, and how they are used.

The following sections describe the properties of JANUS on a coding level, and the system functions available. They are ordered in terms of memory, disk, and address space usage, and then proceed into task communications and realtime operations.

The rather curious names which are sometimes used result from the necessity of compromising between the need for helpful mnemonic names and the SYMBOL defined constraint limiting a symbolic name to 8 characters or less.

B. 1 Resident Tables and Lists

JANUS concerns itself primarily with certain tables and lists kept for the purpose of bookkeeping. I now inteend to provide a list of these,

along with their use. Names may be mentioned which are as yet undefined in this thesis; however, because of the interrelated nature of JANUS, it is necessary to start somewhere. Table elements are almost always exactly one addressable base in size, such as word or byte. This is because, when an element may be referenced from more than one place, including an interrupt routine, it is necessary to reference it in a way that is not interruptable, either by setting a flag that it is not to be touched, by performing the operation in an instruction which can not be interrupted, or by inhibiting the interrupt. JANUS is written to take advantage of realtime, thus interrupts are inhibited as little as possible. As much as possible is done with single non-interruptable instructions.

However, there are certain abnormal conditions which may require abnormal action, including inhibiting all interrupts. These include actual hardware errors (e.g. memory parity), software errors (e.g. traps from unmapped code), and one additional special case. The latter results from having a number of lists partially resident, with the rest of the list existing on the disk within a task. Under normal circumstances, the non-resident task (the Housekeeper) is brought in to tidy up. However, in freak cases, it may be discovered that a list is full or empty, with no recovery procedure available for the requestor. In this case, a resident routine is invoked--the Troubleshooter. This routine suspends all functions while bringing into core enough of the Housekeeper to straighten out the difficulty. For the duration of this process, all interrupts are inhibited. However, this is definitely a last ditch effort on the part of JANUS to stay viable, and thus happens extremely infrequently, provided all tasks and interrupt routines are correct and honest. Any practice which is not completely honest, expecting certain

timing relationships, etc, may work 99% of the time..the 100th time an error will occur, frequently resulting in the destruction of the operation system. The method of getting around this problem is discussed below.

This chapter will consider those tables used in timesharing tasks. Let us consider first the one non-resident table, the Task Control Page (TCP). This page is always the first unique page of the task, and is of fixed format. It contains all information as to the current status of the task which is of interest to JANUS. This includes pointers to routines associated with traps, program status, Signals, and the task USAGE table. The task USAGE table consists of a word (MAXSIZE) specifying the size of the task under the map in pages, and the list (USERPAGE) of pages and their attributes. Each of the latter is in mapped sequence; that is, the N-th entry corresponds to the N-th page of the address space. An entry is null if diskpage 0 is specified, as this page is inaccessable to all tasks. The entries are designed to take advantage of the INTerpret instruction, such that the first four bits are usage information, the next twelve are general information, and the last sixteen are the diskpage address. The attribute bits have the following significance:

0. Absolute code (ABS) page. This page may be dedicated at any time, and bits 8-15 of the entry will specify the unmapped page to load this page into, if bit 0 is set. ABS pages will be loaded into core each time the task is active.

1. Virtually dedicated page. This page must be in core for the duration of any timeslice the task is active.

2. NEED-NEXT. Used primarily in a demand paging task which cannot proceed until that page becomes available.

3. USED-LAST. Again used in demand paging, this bit is set during a timeslice if the page is used. If not set, and no other bits are set, no effort will be made to bring in the page. Bits 2 and 3 are cleared at the start of each timeslice.

4. WRITEBACK. Indicates that this page is modified regularly, and must be unconditionally written back on the disk.

5. Not used.

6-7. The Access Protection Lower Limit (ACL, =0-3) which may be used for this page without error.

8-15. Used if bit 0 is set, as described above. Otherwise ignored, except at task generation and destruction. At generation, the page will be copied onto a new diskpage and the copy used if this byte is non-zero. At destruction, only if this byte is non-zero, will the page be freed. This allows multiple use of an absolute task, since all non-modifiable pages used will be the original copy of the task, and are shared by all task copies. Only the volatile storage will be different for each task, and efficiency may be greatly improved. The convention used is that, if the task allocates a page it did not start with, bit 14 is set, while if a copy of the page is used, bit 15 is set.

The table described above is the only one of which it is necessary to have knowledge in order to write a task. However, other associated tables are described in order to allow one to become more familiar with the operation.

Two of the resident tables are required only because the hardware registers are not readable. These are the access protect image (ACIMAGE) and the map image (MAPIMAGE). These are respectively 2-bit and 8-bit entry tables, each with 256 entries, and are in map sequence.

Another table is the image of unmapped core (TRUECORE). This is again set up to take advantage of INTerpret, as was the usage table.

0. This bit is used by the troubleshooter as a flag for pages it is using.

1. This page is in use by the currently active task.

2. This page is being subjected to swapping.

3. This page is part of the next task.

4. This page must be written back onto the disk before being used for anything else.

5-6. Unused.

7. This page is dedicated to swapping, and may not be used for ABS pages.

8. This page contains a TCP.

9. This page may become dedicated, and should be used only temporarily.

10-13. Dedication level for this page.

14-15. Reuse priority for page.

16-31. Diskpage currently residing in memory page.

The last table actively associated with swapping is the stack of Task Control Pages (TASKPAGE). Again a table of one word entries, this is the only reference to a task which is kept resident. Bits used are:

0. This task must proceed immediately, regardless of the time-sharing ring (RUSH bit).

1. This task must be started next in normal sequence; i.e., if a task is being brought into core or is ready to go, JANUS will proceed with it, but will cause this one to be the next task readied (HURRY bit).

2. This task is loaded and is ready to proceed.

3-14. Unused.

15. This task is on wait status. This bit is set at the request of the task, and is removed only on the receipt of a Signal or if bits 0 or 1 of this word get set. Bits 0, 1, 2, and 15 are cleared each time a task is started.

16-31. The diskpage address of the TCP of this task.

Associated with this table are an entry (TSKCNT) specifying the number of tasks which exist, and an entry (NEXTTCP) specifying which task the Swapper is currently manipulating as the next task. NEXTTCP has these attributes, set by the Jobchanger:

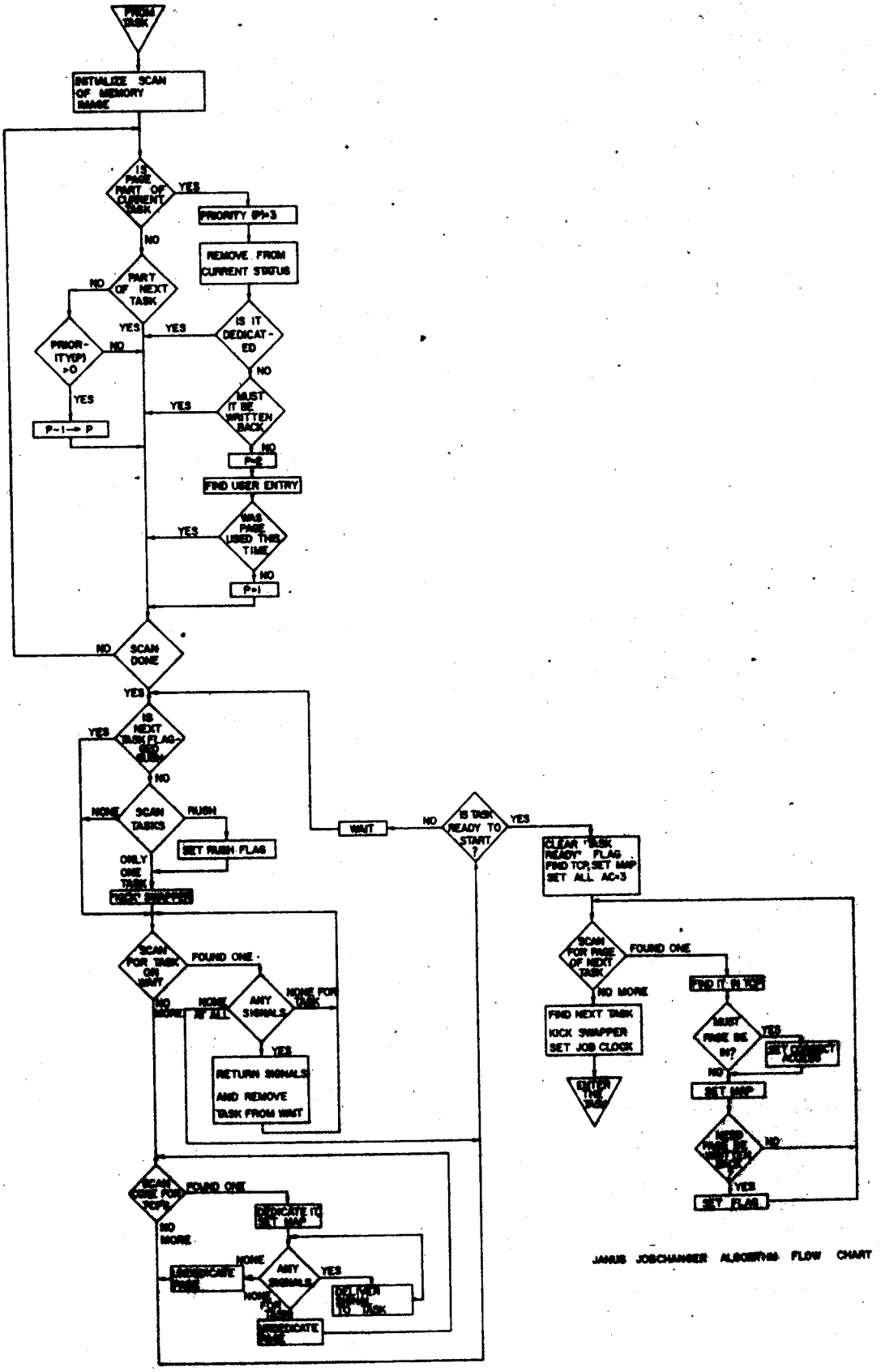
0. This is a new task to load. Flag cleared by Swapper.

1. This task is on rush priority.

2. All tasks are on wait status.

In order to understand the timesharing process, it is necessary to know that the lowest priority interrupt in the machine must be a clock, (the Jobchanging interrupt). Timesharing proceeds as follows (Figure 19):

1. At some point in time during the execution of a task, the Jobchanging interrupt fires, either because the time is up, or because the task has, for its own reasons, triggered the interrupt. As soon as there are no higher level interrupts active, and there is no inhibit on the clock, an Exchange Program Status Doubleword (XPSD) instruction is executed, which references the TCP of the task. As a result, the current status of the task is saved, and control is transferred to a part of the task (Slice-end routine) which performs all unique and necessary slice-end functions, before transferring control to the resident Jobchanger routine.



JANUS JOBCHANGER ALGORITHM FLOW CHART

Figure 19. The Job Changer - flow chart.

2. The Jobchanger performs common slice-end cleanup, examining each entry in table TRUECORE. The reuse priority (P) is a measure of how recently that page was in use, and to what degree it was used. The lower the priority, the less it is necessary for that page to remain in core. A non-zero priority is reduced by one if the page was not part of the current task. If the page was part of that task, the corresponding entry in USERPAGE is found. If not flagged as ABS, virtually dedicated, or USED-LAST, the priority is set to 1--otherwise it is set to 3 if it must be written back, 2 if not. The flag for being a part of the current task is also cleared.

3. If NEXTTCP does not contain its rush flag, TASKPAGE is scanned for the presence of a RUSH flag. If found, the rush flag in NEXTTCP is set, and the Swapper (RAD interrupt routine) is kicked. If only one task is active, the Swapper is also kicked. In kicking the Swapper, the RAD status is checked. If not operational, a comment is produced on the console teletype, and JANUS hangs the machine in an alarm loop until the RAD becomes operational.

4. All tasks on wait status are checked for the presence of RUSH or HURRY conditions, and for the presence of one or more Signals. If any of these conditions hold, the task is removed from wait status.

5. If any Signals exist, all TCP's which are in core are located in turn. The map is set to reflect the location of each one, and a search is made of Signals, to locate and transmit all Signals for that task, deleting each Signal found in the process.

6. The task specified by NEXTTCP is tested. If that task is not ready to proceed, a WAIT instruction is executed, and after the next interrupt, execution transfers back to step 3.

7. If the task is ready to proceed, the map is set for the TCP, and the access protects for all pages are set to 3. TRUECORE is scanned for entries flagged as part of the next task. For each such entry, that flag is cleared, and the associated page name is picked up. Each reference to that page is found in table USEPAGE and the entry is INTerpreted. If it is flagged as ABS or virtually dedicated, the access protect specified in USEPAGE is set. The map is set, according to the locations of the references in both USEPAGE AND TRUECORE. If the USEPAGE entry is flagged as having to be written back, the corresponding flag is set in TRUECORE. All "NEED-NEXT" and "USED-LAST" flags are deleted from USEPAGE.

8. The NEXT and HURRY bits of the TASKPAGE entry are cleared. Routine FINDNEXT is called to locate the next task to process, and this information is saved in NEXTTCP. If more than one task is active, the Swapper is kicked, the timeslice duration is computed, and the Slice-start routine of the new task is entered via an LPSD, resetting the Job-changing interrupt.

A typical example of the sort of timing problems which must be always considered is seen here, in that, while the Jobchanger is the lowest priority interrupt, the interrupt which ticks the clock is one of the highest. Setting the clock and transferring to the task requires two instructions. As the Jobchanging interrupt will fire only as the clock runs through zero, and not at all if the interrupt is active, it is conceivable that, as a result of heavy interrupt usage, the clock may run out between setting and transferring control. The result would be that a task would start with a timeslice, not of a nominal 100 milliseconds, but instead, of two months, the time required for the clock

to tick 2 billion times. As a result, the Jobchanger may not set the clock itself, but instead tells the task how much time to ask for. Because of this, the Slice-start routine must always operate with the Jobchanger inhibited. Also, the task can play tricks, such as setting the clock to a fixed fraction of the time, and at the end of this partial timeslice, start a different segment of itself for the remainder of the time. A task may thus timeshare itself within a timesharing environment.

Let us now turn our attention to the Swapper, the resident RAD interrupt routine (Figure 20). This routine may be entered in two ways: normally through an I/O operation, or abnormally by being "kicked", that is, by the execution of a specific and easily recognizable invalid I/O operation, instigated from outside the interrupt routine, and which can occur only if the RAD is not busy.

1. Test if entered via kick. If not, check the last operation performed. If an error was detected, go to POINT S. Otherwise determine diskpage used, the entry of TRUECORE referenced, and the operation performed. If write, clear all flags from the TRUECORE entry except 2 and 7-13. If read, set the diskpage into the TRUECORE entry, set bit 3, and clear all bits but 3, 7, and 9-13. Finally, delete that operation from the queue.

2. Test NEXTTCP. If flagged as a new task, clear that flag, clear FLAGS, clear bits 2 and 3 from all TRUECORE entries unconditionally, and clear out the queue. Exit if no tasks should proceed, or if the next task is ready.

3. Copy the diskpage specified in the TASKPAGE entry specified by NEXTTCP. Determine if it is in core. If not, proceed to POINT A. Otherwise set the flag in TRUECORE accordingly, and compute the

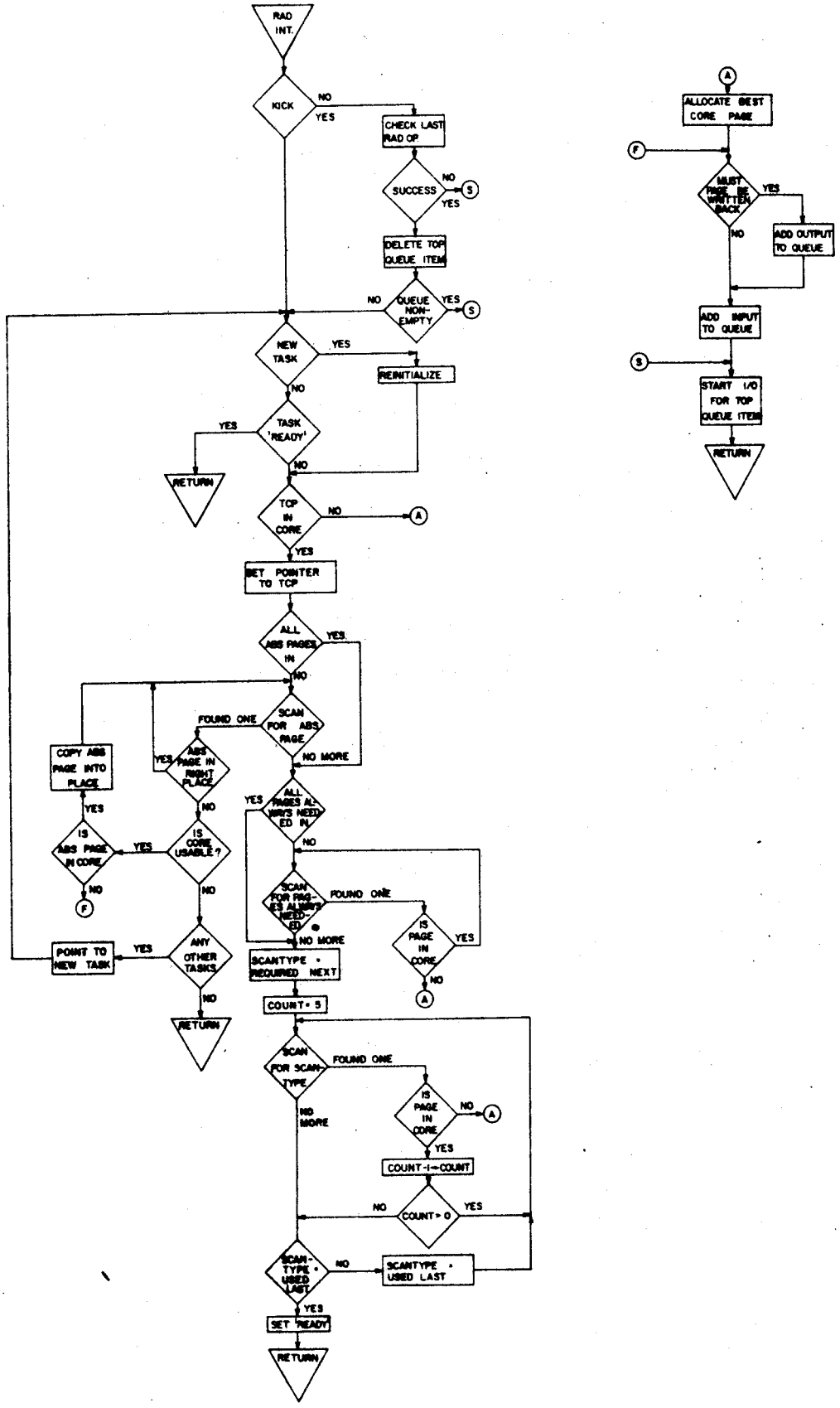


Figure 20. The Swapper - flow chart.

unmapped address of USEPAGE for future reference.

4. Test FLAGS. If not set to indicate all ABS pages have been found, scan USEPAGE for ABS entries. For each one found, determine if in the correct place in core. If so, set bit 3 of the TRUECORE entry and continue. If not, determine if the required page is free, and if not, find a new task and continue to 2. if possible. If the page is free, scan TRUECORE for the desired diskpage. If found, copy into the correct page, writing the original contents out to the RAD if necessary. Define the new contents, and free the page it was in. If the page was not found, go to POINT F. When all ABS pages have been located, set FLAGS to reflect the fact, so that 4. can be skipped in the future.

5. Check FLAGS to determine if all virtually dedicated pages have been found. If not, scan through USEPAGE to locate all such entries, ignoring all ABS entries. For each one, scan TRUECORE for the diskpage specified. If not found, proceed to POINT A. Otherwise set bit 3 in the TRUECORE entry and continue. When all virtually dedicated pages have been located, set FLAGS to reflect the fact, so that 5. can be skipped in the future.

6. Scan USEPAGE for all entries flagged "NEED-NEXT", ignoring all ABS and virtually dedicated pages. Scan TRUECORE for that page. If not found, go to A. Otherwise set bit 3 and continue, counting that entry.

7. If less than 5 pages were found in step 6, repeat the search, looking for "USED-LAST" pages. As soon as a total of 5 pages have been found in either of these latter categories, set the task ready to proceed (in TASKPAGE) and exit.

POINT A (Allocate). Two assignments are presented to this routine--the weights to give the swap dedication (W_S) and dedication (W_D)

attributes in TRUECORE. Each entry of TRUECORE is INTerpreted to determine if dedicated, or in use for the current task or the next task. If not, a value is computed on the basis of:

$$V = 2*(\text{WRITEBACK}) + W_S*(\text{swap dedication}) + W_D*(\text{dedication}) + 2*(\text{reuse priority}),$$

where quantities in parentheses are TRUECORE entry attributes. The page with the lowest value V is passed to POINT F.

POINT F (Fetch). Set bit 2 (page undergoing swapping) in TRUECORE entry specified. If that page must be written back, generate an output entry and put into the queue. Always generate an input entry for the queue.

POINT S (Start I/O). Set up and initiate the I/O operations for the first entry in the queue, and then exit.

As a result of the Swapper algorithm used, JANUS is a primitive learning program, in that it tends to keep in core those diskpages used most frequently. Given a set of tasks which may all fit into the machine core memory simultaneously, and a demand paging scheme, only a few time slices are required for JANUS to discover the pages required and bring them into core, where they will remain until freed or replaced. As a result, in the case where everything fits into core, the overhead due to Swapping, Jobchanging, and demand paging drops to an extremely low value compared with other timesharing systems.

B. 2 Resident Routines

Since it is required of most tasks that they be able to manipulate the resident tables, and since it should be unnecessary for a task to have to know all the details of the tables, it is desirable to have a

number of resident routines, callable from tasks, which will perform the manipulative functions required.

The calling sequence is common to all routines. It is:

BAL,R11 ROUTINE

with parameters transmitted in R6-R11 as necessary. Any information is normally transmitted back in the same fashion, and if necessary, CC4 is set to 1 if the request was satisfied successfully.

Consider in this section those routines which deal with the time-sharing tables already described. These will be subdivided for the purpose of description by the table they reference. Each descriptor will be of the form:

NAME(*)

R Parameters transmitted Parameters returned

Comments

Where (*) is a flag in the descriptor to specify that a success code is returned. Each parameter is described by contents and register thus:

R6 Value **00FFXX

where 8 hexidecimal digits are displayed, and the characters mean:

* Unpredictable garbage, to be ignored

0 Zero

F All bits set to 1. Any hexidecimal digit may be used.

X Field of interest.

B. 2 (1) Routines Which Deal with the Map

A. GETMAP

R6	Not used	********	Unmapped address	000000XX
R7	Mapped address	000000XX	Unchanged	000000XX

This routine permits referencing MAPIMAGE, in order to locate the

actual page a specified page maps into. The addresses are page addresses.

B. SETMAP
R6 Unmapped address 000000XX
R7 Mapped address 000000XX

This routine returns if the map is set as specified. Otherwise, MAPIMAGE is updated as requested, and the map is reloaded before return.

B. 2 (2) Routines which deal with the Access Protection.

A. GETAC
R6 Not used ******** Current access 0000000X
R7 Mapped Address 000000XX Not used ********

This routines permits referencing ACIMAGE, to located the currently used access for a page. Page addresses are used.

B. SETAC
R6 Access Protect 0000000X
R7 Mapped address 000000XX

This routine compares the access specified with that in ACIMAGE, returning if they are identical. Otherwise it updates the image, and reloads the access protect registers.

B. 2 (3) Routines Which Deal with Table TASKPAGE.

These routines are all of the same form, since in each case:

R6 TCP NAME ****XXXX

All routines are called by name.

A. WAIT

The specified task is located, and placed on wait status.

B. RUSH

The specified task is located and its RUSH flag is set. This routine is reentrant and may be called from an interrupt level.

C. HURRY

The specified task is located and its HURRY flag is set. This routine is reentrant and may be called from an interrupt level.

D. KILL

The specified task is located, and removed from the ring of active tasks to a stack of dead tasks, to be serviced by the system MORTICIAN task. Since this routine has to rearrange a table which is referenced from multiple interrupt levels, it is necessary to inhibit both I/O and external interrupts for a brief period (31.0 microseconds). However, this routine is called but once for each task--thus the condition will not occur often. Furthermore, this is the only place in all of JANUS where it is necessary to inhibit these interrupts as a normal condition.

E. START

Unnecessary bits are masked off the task name, and an attempt is made to add it to the ring of tasks. If successful, a "wake up" Signal is sent to the task.

B. 2 (4) Routines Associated with Table TRUECORE.

All references are associative, in that the diskpage contained in a page of memory is specified. In all but specific cases, the operation will not succeed unless the specified page is in core, and flagged as being part of the current task.

A. CURRENT*

R6 Disk address ****XXXX Unmapped page address 000000XX

If the page is in core, it is flagged as being part of the current task.

B. RITEBACK*

R6 Disk address ****XXXX Unmapped page address 000000XX

The flag is set that the page specified must be written back onto the RAD, because true copy no longer exists there.

C. REDEFINE*

R6 Old disk address ****XXXX Unmapped page address 000000XX
R7 New disk address ****XXXX Unused ********

If the page specified by R6 is in core, change the name to that specified by R7. This routine is used in disk copying operations, since a task may bring a page into core, change its name (which is equivalent to making a copy), then modify the copy independent of the original.

D. DROPFILE*

R6 Disk address ****XXXX

This routine is used to get rid of pages not currently in use, but which must be preserved. If the page is in core, and not dedicated, the page is removed from the range of the task. If, in addition, the page need not be written back, the page of memory is freed.

E. DEDICATE*

R6 Disk address ****XXXX

If in core, the dedication level of the page is increased by 1, locking it in place as a resident page.

F. UNDEDICT*

R6 Disk address ****XXXX

If in core, the dedication level of the page is decreased by 1. If the resultant dedication level is zero, the page is free to engage in swapping again.

G. ALLOCATE*

R6 Disk address ****XXXX Unmapped page address *00000XX

This routine evaluated the worth of each page of TRUECORE, ignoring all pages in use, dedicated, or which must be written back, and assigns a value according to:

V = 4*(reuse priority) + 2*(dedicable page) - (Swap dedicated page), where the quantities in parentheses are attributes of each TRUECORE entry.

If one or more pages are not ignored, the one of these with the lowest value is assigned the new diskpage specified, and flagged as part of the current task. This routine is used in attempts to acquire temporary storage without proceeding through a Jobchanging cycle.

H. FREE*

R6 Disk address ****XXXX

If the page is in core, and not dedicated, it is unconditionally freed. That is, it is undefined, and will never be written back onto the RAD.

B. 2 (5) Routines Associated with Disk Pages.

These routines deal with a stack of resources, which is only partially resident. If at any time the stack is endangered, the Jobchanging interrupt is triggered. Hence any task should permit Jobchange to occur between each request to these routines. A diskpage address is a 16 bit, non-zero, unsigned quantity specifying the location on the disk where it may be found.

A. ALOCDISK*

R6 Unused ******** Allocated disk page 0000XXXX

If a disk page is available, it is allocated to the requesting task. No effort is made to know to which task a specific diskpage is

assigned.

B. FREEDISK

R6 Diskpage ****XXXX

The diskpage specified is returned to the pool of free disk pages.

Since JANUS does no elaborate checking, it is the responsibility of the tasks to use these routines and resources properly. Typical examples follow, which illustrate the difficulty which may arise from thoughtless use of the functions.

1. Over-dedicating a page. A page may be dedicated up to 15 times without difficulty. This is sufficient if it is dedicated once for each interrupt routine which may reference it. However, if it is dedicated a sixteenth time, an arithmetic carry occurs, such that the page is no longer dedicated. In as much as such a page is normally flagged as dedicable, that bit is also cleared, and the carry may extend to defining the page as a TCP, or even dedicated for swapping. When undedicated, the page enters the swap swirl. The first time an unmapped mastermode transfer is made into the middle of data or mapped code, all hell breaks loose.

2. Over-undedicating a page. The same arguments apply as in 1, except that a borrow occurs, leaving the page totally dedicated.

3. Overdefining a page. Under certain circumstances, it is possible for a free diskpage to be in core. (For example, the last task which freed the page may have been interrupted by the Swapper after freeing the page, but before removing the reference from the TCP, such that the Swapper did cause the page to be brought into core again, where it might remain for a long time under low usage.) Also, a freshly freed diskpage is most likely to be allocated next. As a result, one should

never allocate a page directly, but should instead first check if the page is in core. If not, it may be allocated. If allocation is unsuccessful, then there is no recourse but to effect Jobchange, causing the page to be actually loaded off the disk.

Similarly, in freeing a page, a task is being polite to all users of the machine if it performs a sequence of:

A. Freeing the diskpage,

B. Deleting the TCP entry,

C. Checking if the diskpage was in core, and if so, freeing that page of memory, all without permitting Jobchange to occur.

4. Difficulty can also ensue from freeing a disk page twice, since the name will now appear in two places, and may be referenced by multiple tasks in the future.

5. Making requests with invalid diskpage addresses. Any reference to diskpage zero is ignored by the Swapper and Jobchanger, since diskpage zero specifies an unused (null) entry in various tables. However, if a task tries to look up page zero, and a null page exists in core, then that page will be found. Similarly, defining a page to have a disk page address outside the range of the RAD, or requesting that such a page be brought into core, will cause the Swapper to hang unconditionally. The only valid diskpage names are those a task starts with, or has allocated.

B. 3 Demand Paging

Under JANUS, it is possible for a task to operate without being entirely in real memory at all times. This scheme is called demand paging, in that a given page of the task is brought into the working

memory upon demand, whenever referenced. The routine involved is over half a page in length, and there is a point of diminishing returns, beyond which it is no longer economical to use a half page of virtually dedicated code for demand paging. Since demand paging applies only to slavemode code and references, this limit is reached when there are about five pages demandable. If there are more than five, demand paging becomes profitable.

The demand paging algorithm is described here both for its use, and to illustrate the use of previously described JANUS routines. This routine is full-blown, in that it takes care of all eventualities and idiosyncracies of the Sigma 7 in addition to demand definition (the automatic extension and definition of the task address space). Certain features may be eliminated with previous knowledge of the task usage--if it is known unconditionally that the trapping instruction will always do word addressing, and will always be present with the correct map and access, esoteric tests may be dropped.

The demand paging routine is connected to the X'40' trap (non-allowed operations), which includes violations of memory protection. There are two parts, shown in Figure 21--one of which deals with JANUS, and is called by the second, which interpretively decodes the trapping instruction. We consider first the JANUS oriented routine.

Function SCANPAGE(EWA)--S(A). EWA is the Effective Word Address. The routine always expects a word address as an argument. The routine determines the status of the page referenced, and returns Condition Codes (CC) as follows:

1. CC = 0, EWA in registers. 2. CC1 = 1, EWA is not in core.
3. CC2 = 1, situation improved--EWA is available, but usage was

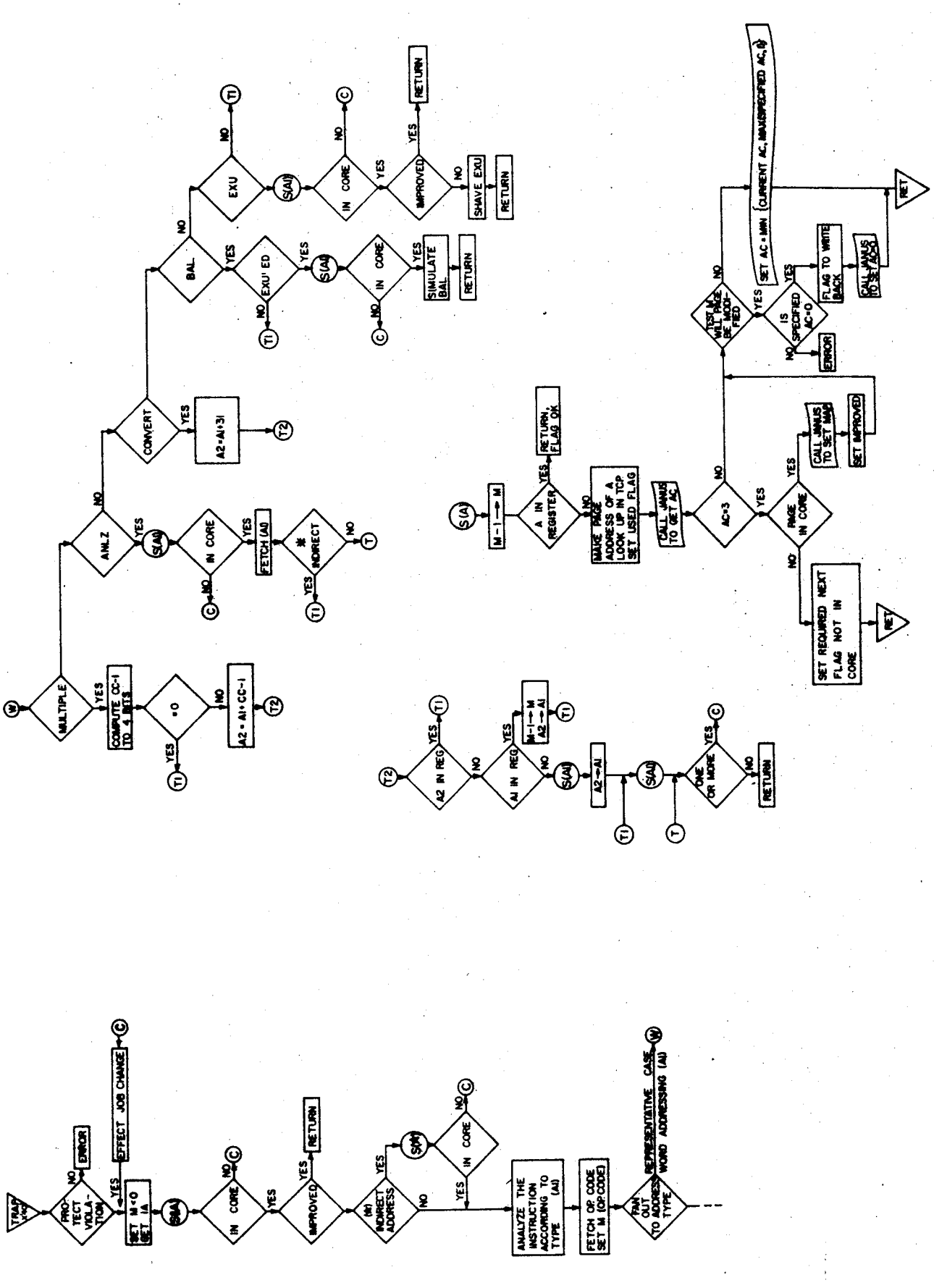


Figure 21. Demand paging - flow chart.

limited by access protection. 4. CC3 = 1, situation normal--EWA in core, AC set as specified in TCP. 5. Error detected--unconditional transfer to error routine.

Two quantities set by the main routine are referenced in addition to the previously defined tables. These are: 1. MODCOUNT: maximum number of pages left which may be modified by this instruction. 2. EXUFLAG: a flag which indicates that the EWA is an instruction address, and to be treated accordingly.

SCANPAGE IS OUTLINED:

1. If EWA is in registers, return with CC = 0.

2. Make EWA a page address (EPA). Look up corresponding entry in USAGE table, skipping to 9. if within range of task (EPA less than MAXSIZE), and if page address is non-null.

3. If outside range of task, increase range to MAXSIZE = EPA+1.

4. Call ALOCDISK to allocate a diskpage. If none available, set CC1 = 1 (page not in core) and return.

5. Use allocated page to make non-null entry in USAGE table, thereby defining it.

6. Call CURRENT to determine if diskpage is in core. If so, skip to 10.

7. Call ALLOCATE to get a core page. If successful, skip to 10.

8. Set NEED-NEXT flag in USAGE table, set CC1 = 1, and return.

9. Set USED-LAST flag in USAGE table. Call CURRENT to determine if page is in core. Go to 8. if not.

10. Here if page is in core. Look up Access Protect Limit (ACL) specified in USAGE table. If ACL = 3, error.

11. Call GETAC to discover the ACcess (AC) the page is operating

under. If AC = ACL, set CC3 = 1 (situation normal) and return.

12. Test and clear EXUFLAG. If not set, skip to 16.

13. Set ACcess to Set (ACS) = 1.

14. If ACS less than ACL, error. If ACS less than AC, call SETAC to set AC = ACS. Set CC2 = 1 (situation improved) and return.

15. Otherwise set CC3 = 1 and return.

16. Reduce MODCOUNT by 1 and test. If negative (page will not be modified), set ACS = 2 and go to 14.

The main routine:

1. Save trap conditions and registers. If the trap conditions do not include memory protect violation, error.

2. Set EXUFLAG to indicate instruction address reference, MODCOUNT to no memory being modified.

3. Get EWA of trapping location, and call SCANPAGE.

4. If CC1 is set, skip to 20--if CC2, return. If CC = 0, reload the registers.

5. Get the instruction and save it. If it is indirectly addressed, call SCANPAGE using the indirect address as an EWA. On return, skip to 20., if CC1 is set.

6. Restore the registers. ANaLyZe the instruction, saving conditions. Initialize various registers, and get the N-th entry from table OPCODE, using it to set MODCOUNT. OPCODE is a table, in instruction sequence, of the maximum number of pages any one instruction may modify. (It is to be noted that, while 108 instructions of a possible 128 are defined, only 23 of these modify one or more pages of core. Only one of these can modify as many as 3 pages with one instruction.) Restore the ANaLyZe conditions and fan-out to various special handling

(nos. 7.-11.) on the basis of the instruction type.

7. If byte addressing, determine if decimal. If not, make word address and skip to 18. Otherwise determine upper limit of memory addressed. Make both upper limit and lower limits word addresses, then skip to 17.

8. If halfword, make word address and skip to 18.

9. If byte immediate, determine destination and source pages. If not translate, go to 17.; otherwise, call SCANPAGE to test destination, save condition codes, then compute upper limit of source and skip to 17.

10. If doubleword, make word address. Determine if stacking instruction. If not, skip to 17.; otherwise call SCANPAGE to test Stack Pointer Doubleword. If CC1 = 1, skip to 20., otherwise determine upper and lower limits of core referenced, going to 18. if one word referenced, 17. if more than one.

11. If word addressing, test for special cases and fan out to: 12. if EXU, 13. if BAL, 14. if ANLZ, 15. if Multiple, and 16. if convert. If none of these, go to 18.

12. Here if EXU. Call SCANPAGE to test EWA. Go to 20. if CC1 = 1, return if CC2 = 1. If CC3 = 1, EXU can proceed, but the target instruction (which may be another EXU) cannot. (This case is one of the design faults of the Sigma 7, which of all instructions allows infinite levels of referencing only to the EXU instruction. This capability is not only unnecessary, but is indeed a handicap to use in demand paging, where accessible address space may be larger than the actual machine, to the end that a slave mode program could hang up the entire machine with a large ring of EXU's.) As a result, it becomes necessary to "shave" a

chain of EXU's, as follows. A special pair of locations are provided. The target instruction is copied into the first of these locations. If the address of the source EXU is not the special location, the second location of the pair is formed into an unconditional transfer to the instruction following the EXU. The trapping Program Status Doubleword is modified to point at the special location, rather than at the source EXU, and we return. This process, although slow, is safe in all cases but one, namely BAL.

13. Here if BAL. Determine if the instruction was at the special EXU location, and if not, skip to 18. If, however, the BAL was the target of a trapping EXU, call SCANPAGE to evaluate the EWA, and if CC1 = 1 is returned, skip to 20. Otherwise, look up the link register specified, and force a link (using the second special location), and branch, plugging the EWA into the PSD. Then return. This sequence, while not foolproof, does guarantee that one level of executing BAL will work correctly, with respect to FORTRAN-like parameter lists.

14. Here if ANLZ. Call SCANPAGE to locate EWA. Skip to 20. if CC1 = 1. Otherwise determine if instruction being ANLZ'ed is indirectly addressed, skipping to 18. if so, else to 19.

15. Here is LM or STM. Compute top address of sequence, and to to 17. if more than 1 word referenced, otherwise to 18.

16. Here if CVA or CVS. Compute top address, as EWA+32.

17. Here to check out two addresses. If the second is in the registers, skip to 18. If the first is in the registers, reduce MODCOUNT by 1 (if one or more pages may be modified, the first reference will be of this class), and skip to 18. If both are in the same page, skip to 18. Otherwise call SCANPAGE to evaluate the address, and save

the conditions.

18. Here to evaluate a single address. Call SCANPAGE to do so, merging conditions returned with previous conditions.

19. If all pages referenced are in core, return.

20. Otherwise effect Jobchange, and upon return, go to 2.

At this point it would be well to point out several anomalous cases. These are the cases where a sequence of instructions may work differently in a demand-paging environment than in a normal environment. In as much as these are primarily hardware limitations, they must be considered as definite design errors in the Sigma 7. (It must be remarked, however, that in general the Sigma 7 is a well-engineered machine, designed for the convenience of the programmer, rather than the engineer. These design errors are due to a lack of foresight, since the Sigma 7 was not planned with a demand paging capability; this device is so powerful that it is frequently used, however.)

One of these cases is the previously mentioned infinite chain of EXU's. A second is that of BAL, which the hardware treats as a link-and-branch. That is, the link register is set while the effective address is being computed and tested. If BAL traps because of an access protect violation, the link register will have been modified from its previous value. As a result, one cannot safely use the link register to hold either an index or indirect address for BAL, as there will be no recovery possible if a demand paging trap occurs.

A second nuisance is that of conditional branches (BCS, BCR). The hardware assumes that a branch will normally go, and thus in anticipation will access the effective address. If the branch doesn't go, the hardware must recycle and get the next instruction. (This is why a

branch that doesn't go takes 50 percent longer to not branch.) Each of these references will trap, however. As a result, more pages may be referenced than will be used, especially since the No-OPeration (NOP) instruction, a favorite for parameter lists, is normally an unconditional no-branch.

Yet another source of annoyance is that the access protection necessarily used to implement demand paging does not apply to mapped master-mode code. This has two results. First, all storage commonly referenced by master-mode must be virtually dedicated or ABS to insure that it will be in core at all times when it may be referenced. Second, there is no guarantee that any non-virtually dedicated area will be in core at any given time. On the execution of a CAL instruction (a specific set of instructions which allow the slave-mode to make up to 64 unique requests to master-mode by trapping) the only pieces of non-dedicated area guaranteed accessable are the instruction itself, any indirect address, a possible chain of EXU's leading to the CAL, and the registers. Specific address references are guaranteed accessible only until the first time Jobchange can occur. (The alternative is to use a SCANPAGE-like routine to check each possible reference.) Hence, all parameters must be transmitted through the registers.

The only real criterion for using master-mode is to perform operations which are to be denied to the slave-mode directly. Let us consider these operations briefly. They can be divided into two categories; the execution of privileged instructions, and the referencing of storage in a way not allowed to the slave-mode. We can eliminate many of the problems described by the extension of master-slave operation to include two more classes. Let us call these meta-master and

meta-slave operations. Master and slave would still have their present form of operation. Meta-slave, however, would permit the execution of privileged instructions, while applying all the addressing restrictions of the access protection. Meta-master would also be restricted by the access protection, except that write protection would not apply. With this change, the only pages which need be virtually dedicated under JANUS would be those containing the TCP and the X'40' routine. The X'40' routine would be the only one which need run in master-mode; all other monitor functions other than unmapped interrupt routines (to which demand paging cannot apply anyway) could run successfully in meta-modes.

In addition to meta-modes of operation, there are several other features, which, if implemented via hardware, would be a boon to demand paging and timesharing in general. These would be in addition to read- and writeable map access protection registers, single level executes, and recoverable instructions, as discussed above.

One of these is a change register; containing one bit for each mapped page of memory, the bit corresponding to a page would be unconditionally set each time that page was written into. The change register could be cleared before each timeslice, and at the end of the timeslice all modified pages could be located, looked up, and flagged as having to be written back.

A second would be a reference register; like the change register, a bit would be set whenever a page was referenced.

The access trap should be divorced from the non-allowed operation trap, permitting independent operation.

The most powerful tool would be the introduction of a privileged,

execution-analize instruction. One would initialize the registers, specify the pseudo-mode to operate under, and execute this instruction. It would interpretively execute the instruction at the effective address, under the identical conditions under which the trap occurred, stopping short of actually modifying core or registers. When a point is reached within the target instruction where a trap would have occurred, execution would be aborted, and the instruction would cause a register to contain three pieces of information--the offending address which would cause the trap, the protection currently set for the offending page, and the maximum ACcess Required (ACR) which would avoid the trap. Computing the page, one could compare ACR with ACL in the TCP; if less, an error condition would hold. If AC was not 3, one could immediately go set AC = ACL. Writeback and USED-LAST would be automatically controlled by the change and reference registers respectively, and would not have to be manipulated by software on each trapping reference.

B. 4 Program Optimization

While almost any program will run under a demand paging scheme, it is possible to write programs which are completely pathological, executing almost no instructions per second. Conversely, it is also possible to write programs which take advantage of timesharing and demand paging. Fortunately, programs written in this fashion are not penalized when operating in a normal environment. For example, because of demand paging, it becomes feasible to use multirecord I/O buffers resident in core for each device; manipulated by a task during a timeslice, and by an interrupt routine asynchronously. Then only very slow devices, or interactive devices where buffering is infeasible, will be a limit on the speed of

the program. The program will no longer be primarily I/O limited, as it would be when only a single record buffer was used, requiring waits.

The primary methods of optomization are those of limiting modifications and references. Limiting modifications means that one does not rewrite areas within the address space indiscriminantly, but instead causes all variables to be located in contiguous pages or blocks of pages, such that a minimum number of pages will have to be rewritten on the disk. A major item to avoid is rewriting sequences of code. The designers of the Sigma obviously had this concept in mind when they very carefully excluded all instructions which would allow one to easily build instructions in core. Any time it is necessary to build an instruction in line of code, the Sigma designers made it easier to use indirect addresses, indices, or, if absolutely necessary, to build the instruction in a register and execute it there. This holds true also for the insertion of in-line parameter lists. Further, to avoid variables imbedded in code, it is desirable to use common storage areas as much as possible.

The second main facet of optimization is organization. If N routines which are always referenced together are in the same page, the program will execute more than N times faster than if each routine was isolated in its own page, surrounded by little used routines. Remember that demand paging is only a scheme of automated overlays, which cause recently unused areas of core memory to be replaced with demanded pages; if in each instruction it is necessary to fetch another page, efficiency is decreased, since we are now executing useful (as opposed to overhead) instructions at a rate of one every 40-50 milliseconds, the delay being necessitated by waiting for a slow device, the RAD.

B. 5 Signals and the MESSAGE CENTER

One capability required of any system involving asynchronous operations is that of synchronization. Synchronization is effected by the occurrence in time of an event of unspecified form and meaning defined by common convention between the synchronizing and synchronized parts. This idea can essentially be reduced to one bit of information--the event has occurred. It is not normally necessary or desirable to be able to specify that the event has just occurred, since this necessitates defining just.

In a timesharing system, especially one with as much flexibility for asynchronous operations as JANUS has, this synchronizing capability is especially important. Synchronization is necessary between tasks, as when a subtask must inform its parent that it is done, and between interrupt routines and their controlling task, to inform the task of the occurrence of a condition. A task may be unable to proceed until the occurrence of a specific event, and may have put itself on wait status. The occurrence of the event should be capable of pulling the task out of wait status.

In JANUS, synchronization of this form is provided by means of Signals, of form:

00XXYYYY,

where XX is a unique Signal number (0-255), and YYYY is the name of the task it is directed to. The Signal number is eventually used as an index to set a bit in the TCP: if greater than 255, the bit will still be set, but not in the normal signal region. Two Signals have standard definitions; Signal 0 is a wake-up Signal, Signal 1 is a standard abort Signal to the task.

Signals may be sent from any level of JANUS--from the highest level active interrupt routine to the lowest level of a task. Signals are sent to the MESSAGE CENTER via the sequence:

BAL,R11 MESSCENT

R6 = Signal.

The MESSAGE CENTER will always accept a Signal, under all conditions of interrupts. The operation performed is to push the Signal onto a stack. In the event that the stack is full, the Troubleshooter is invoked to handle the condition, and upon return, the attempt is made again. The volume of the stack is checked, and if it is getting significantly full, the Housekeeper task is flagged to HURRY and process the stack; if very full, the Housekeeper is flagged RUSH priority, and Jobchange is effected. Thus only in extreme cases should the Troubleshooter be called upon to handle Signal difficulties. If called in, the Housekeeper will even out the resident stack, moving excess Signals to its own swappable stack, or returning them if the resident stack becomes empty. Thus it is possible that a time delay of up to seconds can occur between sending and receiving a Signal, under heavy Signal usage.

Great pains have been taken to insure that Signals are neither lost nor duplicated. This is done by the use of reentrant routines and multiple stacks, all to avoid the necessity of having an accessible copy of a Signal in more than one place. Thus the stack of Signals is not scanned for a Signal for a specific task in order to remove that Signal--it is instead unstacked, saved in a second stack, and at a later point, Signals are individually removed from the second stack, and if not desired, returned to the first stack via the MESSAGE CENTER. This degree of complexity is necessary because the stack of Signals

may be simultaneously referenced from many levels. For example, while referencing the Signal stack, a task may be interrupted to effect Jobchange, the Jobchanger interrupted by an interrupt routine while fetching a Signal, interrupt routines interrupting each other, and finally when the stack is full, the Troubleshooter interrupts the current active routine and locks out all usage while it unscrambles the difficulty.

However, there are limits beyond which nothing can save a Signal. These usually occur in the case of uncontrolled sending of Signals. These can be avoided by reasonable operation. Tasks should permit Jobchange to function between sending Signals. Interrupt routines should take care to send a Signal only once, even if the condition is recognized repeatedly, until the interrupt routine knows that the task has recognized and acted upon the Signal. A simple way to do this is to use the preformed Signal as a flag. The Signal is placed in the required location by the task when necessary. The interrupt routine would exchange a null entry for the Signal when necessary, and if the Signal fetched was null, would ignore it. Thus a Signal would be sent only once. There are other methods to accomplish the same ends. Null Signals should not be sent to the MESSAGE CENTER indiscriminantly, as a significant amount of time is required to delete them, for the duration of which valuable stack space is lost.

B. 6 Timekeeping

It is often desirable to perform temporal synchronization, either at the end of a specific delay, or at a specific time. As the computer normally keeps track of time by clock interrupts, which are limited in number, and as this function should be provided to all users in a time-sharing system, a resident routine is called for. In JANUS, the lowest

level priority clock interrupt is the Jobchanger; the next higher is the TIMEKEEPER. Time requests are performed by the calling sequence:

```
BIG_BEN
R6    XXXXXXXX  Delay requested (milliseconds)
R7    SSSSSSSS  Signal (Form 1)
      or
R7    8***XXXX  Unmapped address (form 2)
```

The time request is compared with the time delay remaining until the next interrupt--if less, it is set as a new delay. The entry is then pushed into a stack. At each interrupt, each entry is pulled and updated. If the specified time has elapsed, and the entry is of form 1, the Signal is sent on to the MESSAGE CENTER; otherwise the entry is saved on a second stack, while the first is emptied. Then the entries are removed from the second stack, one by one, and if the time is not up, send back to BIG_BEN. Otherwise the entry is of form 2, and the interrupt routine performs a BAL, R11 to that unmapped address. The routine there can perform required operations before returning, and can assume R6-R11 are volitile. This provides a capability of clocked interrupt routines, such as might be used to generate an unbuffered graphic display.

In using form 2, there are certain constraints which must apply to the external routine. First, the routine may not manipulate the clock inhibit bit in its PSD. Second, there is no automatic deletion of such a request upon task exit--it is necessary for a task to wait until the time actually runs out, and control is transferred to the external routine. While the external routine would normally request another delay of the TIMEKEEPER, when exiting it must stop itself. Third, the external routine must always be in core when such a request is pending. Fourth, any time delay requested by an external routine should be greater

than the time normally required from request to return, otherwise the computer will hang, spending all of its time in the interrupt routines. Fifth, external routines should refrain from making more than one request at a time.

Form 1 requests allow the task to be signaled at the end of a given time. Enough information is available to any task to allow it to compute a time delay required to be signaled at a given time. Thus it is possible for a task to be started, and thereafter perform some process every hour on the hour, if so desired. Likewise a task performing a low priority calculation could be brought awake only between midnight and 6AM, or at some other time when the computer is light loaded.

Since a centralized routine is called for by the nature of things, only a small increase in the code required enables the TIMEKEEPER to perform time of day calculations. The current values are available to any task which desires to reference them. These quantities are:

BCDTIME	HHMM	4 bytes of hours and minutes on word boundary.
BCDDATE	DDMMMYY	7 bytes of day, month, and year on a double-word boundary.
BCDDAY	DDDD	4 bytes of day of week on a word boundary.
TRUETIME	XXXXXXXX	1 word of half-milliseconds elapsed to last interrupt.
LASTTOCK	XXXXXXXX	1 word of half-milliseconds to elapse between interrupts.
TICK	XXXXXXXX	1 word of half-milliseconds required until next interrupt.

The TIMEKEEPER deals in actual time, independent of usage. As a result, the times are quite accurate. Each night at midnight the

Housekeeper task is call in to update BCD -TIME, -DAY, and -DATE. The calendar is good to 10,000,000 years, and includes leap year calculations.

The time stack is another of those stacks and lists which extend onto the disk via the Housekeeper. Whenever the stack threatens to overflow, the Housekeeper is called in to tidy up. All entries are unstacked, converted to an absolute time, and saved in the Housekeeper stack. The Housekeeper then reorders this stack, and returns enough of the imminent time requests to half fill the resident stack. As the resident stack is emptied, the same process takes place. Thus, long delays will drift onto the disk until they become imminent. Short delays on the disk drift back to being resident. The delay is thus a guaranteed minimum delay--it may actually be longer in duration than requested. If the resident stack does overflow while a request is being made, the Troubleshooter is invoked to unscramble things, just as it is for the MESSAGE CENTER.

B. 7 Unique Resources

There are a number of unique resources available to the computer which cannot be shared simultaneously, but must be sequentially allocated to one task at a time from the system pool. Unique resources are characterized by an almost universal association with interrupt routines, and thus with I/O operations.

There are four routines used to allocate and free unique resources. Each searches through the resident list of unique codes, returning failure if the device is not located (nonexistant). A "downed" device is nonexistant.

```
IOASK*
R6    ****XXXX  Unique device code
R7    ****XXXX  Name of requesting task
```

If the device is owned, the routine returns failure, otherwise it assigns the device to the requesting task.

```
IOWAIT*
R6    ****XXXX  Unique device code
R7    SSSSSSSS  Signal for requesting task
```

If the device is available, the routine operates exactly as does IOASK. In this case the signal will never be sent--instead the task name from the Signal will be used to assign the device. If the device is in use, the request is added to a stack, and the Housekeeper is signaled, in order to add the request to the non-resident queue for later assignment. In this case, CC3 = 1 is returned. When the task's turn for the device comes up, the device is assigned to the task, and the Signal is sent to inform the task that it now has the device.

```
IODOWN*
R6    ****XXXX  Unique device code
R7    ****XXXX  Name of requesting task
```

If the task discovers that the specific device is not operational (usually by means of an operator key-in), this routine may be called. The device is found, and checked to belong to the requesting task. If so, the request is passed to the Housekeeper, to allow flagging the device down. A downed device cannot be allocated, and all requests, both pending and future, will wait until the device is brought up.

```
IOFREE*
R6    ****XXXX  Unique device code
R7    ****XXXX  Name of requesting task
```

This routine operates exactly as does IODOWN, except that the request sent to the Housekeeper will not cause the device to be flagged down, but instead passed either to the next register, or back to the

system pool. The device will be available as soon as the Housekeeper has processed the request.

In the Sigma 7, unique resources are of two forms, treated differently by the hardware, and thus by programs.

One form is used through the I/O processor (IOP). Characterized by being sequential byte (character) oriented, the IOP is normally used to transmit a buffer in to or out of core memory, and is connected to I/O devices, such as the cardreader. These devices have a unique address, which specifies a device subcontroller. A subcontroller may have multiple devices attached (e.g., teletype with paper tape capability), but a task buys a subcontroller from JANUS. Interrupts generated by IOP devices all filter through a common port, necessitating a common resident routine.

Four routines are specifically associated with the IOP.

IOASSIGN
R6 ****0XXX Device address
R7 ***XXXXX Unmapped external interrupt routine address

The parameters are merged and inserted into the resident stack of active devices.

IORELEAS

R6 ****0XXX Device address

The resident stack of active devices is scanned to find the entry corresponding to the device specified. When found, the entry is replaced by the top entry in the stack, which is then deleted.

IOKICK

R6 XXXXXXX Pseudo-AIO status

It is frequently desireable to be able to generate a device interrupt without affecting the device. For example, the interrupt routine

must be able to determine the device status. It is easier to kick the interrupt routine than to have duplicate code outside the interrupt routine. IOKICK makes this possible by pushing the pseudo-AIO status into a "kick" stack (queue) and then triggering the I/O interrupt. This, in effect, provides a signal path from the task to the interrupt routine. The routine is reentrant, and may be called sparingly from any interrupt level.

The interrupt routine is not directly callable.

On interrupt, the interrupt is acknowledged (this AIO should be the only one ever executed in the machine) and the device address returned is then used to scan for the device in the active device stack. If not found, diagnostic information is saved and the Housekeeper is signaled. If found, the calling sequence BAL,R11 is performed to the address associated with the device, with R6 containing the AIO status. Upon return, the AIO is again executed. This occurs until the AIO indicates that no interrupt was recognized. At this point, a unique flag is added to the kick queue. The kick queue is not scanned, one entry at a time. Each entry is treated exactly as an AIO status word, and sent to the corresponding interrupt routine. When the flag pops off the queue, the interrupt routine exits.

The second form of resource is that associated with the Direct I/O (DIO) port. The DIO is characterized by word data transfers between the external world and the registers, under program control. It is used primarily in situations where a buffer cannot be used because the data must be manipulated before use. These resources include interrupts, register pages (used exclusively by some interrupt routines), and external devices (in the MSU configuration, these external devices

are General Purpose Interface (GPI) half registers, and will henceforth be referred to as such).

All devices are specified by an 11-bit address; a 3-bit prefix code is defined to distinguish between different devices with the same address. This code has the values:

0--IOP,

1--external interrupt,

2--register page, and

4--GPI.

The DIO addresses are numbered sequentially, from the lowest to the maximum number available. Thus, (MSU configuration) external interrupts, 0-7; register pages, 1-3 (0 is common to all users); GPI, 0-7.

The only resident routines geared specifically to the DIO are those dealing with the external interrupts.

```
DTCHINTR
R6    *****XXX  Interrupt address
R7    will is loaded with the standard system interrupt location
```

plug and then control is transferred to:

```
ATCHINTR
R6    *****XXX  Interrupt address
R7    XXXXXXXX  Instruction to plug into interrupt location.
```

The operation is performed.

One point must be stressed with respect to use of routine IOWAIT. A task requesting a device through IOWAIT normally enters the wait state if the device is not currently available. A situation can occur, whereby two tasks, each using a unique device, can request the other's device. They will then hang upon each other, both being out of service, and keeping their device out of service until one or the other is explicitly told to let go. Any task which can get into this situation

should have a capability built in to provide for this occurrance. It may be avoided by asking for the additional device through IOASK, by using only one device at a time, freeing it before getting another, or by the use of symbionts.

The allocatable resources do not include the console teletype or RAD, as these are permanently assigned to JANUS.

B. 8 Prefices and the Console Teletype

An operation common to almost all tasks is that of communication with the operator. As JANUS handles the RAD through the Swapper, so also does it handle the console teletype. The teletype handler is shared by all users. Input is character directed, in that a unique prefix directs the input to the correct task. Input may thus be scrambled, not necessarily in the order of request. Output is strictly ordered, such that each request is added to a queue, to be typed out in due time. If input is in progress when an output request is made, the input is interrupted and a recovery procedure is set up, such that when the output is done, the original input is echoed and input is then continued at the point the interruption occurred.

Each request is accompanied by a unique prefix. Certain prefixes are defined, such as & for JANUS, CR for the card reader, LP, CP, PL, TY5, MTO for other devices. A task which requires a unique prefix may get it from JANUS in a manner analogous to diskpages. That is, there are two routines which allow a task to get or return a prefix.

GETPREFX

R6 ******** Not used CCCCCCCC Unique prefix allocated

PUTPREFX

R6 CCCCCCCC Unique prefix freed.

A prefix is a word of TEXTC format; that is, the first byte is a count of the useful bytes which immediately trail the count. Extra unused bytes are blanks packed onto the end of the word.

Again like diskpages, prefix handling involves a small resident stack, and a larger stack in the Housekeeper. The Housekeeper may be called in to adjust the resident stack, and if the stack over-or underflows, the Troubleshooter is invoked to fix matters.

The teletype has a number of routines associated with it. I will first list those which are callable from tasks, and then consider the interrupt routine:

DISPLAY*

R6	XXYYYYYY	Count of bytes; mapped address of record
R7	CCCCCCCC	TEXTC prefix
R8	OOXXXXXX	Signal to send

This routine is used primarily for low priority output, where it is not desirable to dedicate a page containing a record. If the one-record buffer is not free, failure is specified. Similarly, if the output queue is full. Otherwise save the Signal, copy the record into the display buffer, flagging it busy, then make up a new set of type parameters to send a Signal to the Housekeeper on completion. (On receipt of this Signal, the Housekeeper frees the buffer, flags it not busy, and Sends the Signal originally specified.) Then proceed to routine:

TYPE*

R6	XXYYYYYY	Count of bytes; unmapped address of record
R7	CCCCCCCC	TEXTC Prefix
R8	OOXXXXXX	Signal to send

Registers are unchanged and failure is specified if unable to accept the request. Otherwise "kick" the interrupt routine, return success.

ACCEPT*

R6	XXXXXXXX	Count of bytes	000XXXXX	Byte address of RESBUF
R7	CCCCCCCC	TEXTC Prefix	000XXXXX	Byte address of input
R8	00XXXXXX	Signal to send		

Registers are unchanged and failure is specified if unable to accept the request. Otherwise, compute the addresses to return, then return success.

When a task has received a Signal that the input requested has occurred and is in RESBUF, the input should immediately be processed, either by copying RESBUF into the task's storage area or directly. As soon as possible, the buffer should be freed to allow other input to proceed.

TTYFREE

This routine clears the flag that the input buffer is in use, and "kicks" the interrupt routine.

Two routines are available to delete input requests. (This is a job of the task on exit, and at certain other times.)

DELETETY

R6	****XXXX	Name of task

DELTYSIG

R6	SSSSSSSS	Specific Signal

In either case, the entry is flagged to indicate the type of search, saved in a "delete" cell, and the interrupt routine is "kicked".

The interrupt routine has five modes of operation.

1. Entered via "kick". Test if to delete input request. If so, search and delete as necessary for all occurrances of the condition, then clear the "delete" flag and exit.

If not delete, test if any requests are in the output queue. Skip

if not, or if in override or type mode. Otherwise halt the current input operation, initialize a recovery, and go start output.

If not type, check if buffer freed. If so, and not typing, start new input--otherwise initialize recovery.

2. Character mode. Used while inputting a prefix. If after 3 characters have been input, no match is found with any requests, a question mark is output, and character mode is reinitialized. If the prefix is recognized, copy the prefix into RESBUF, then proceed to read the associated record. (RESBUF is the standard buffer for console teletype messages, and is of TEXTC format, including the prefix.)

3. Input mode. Used while inputting a record. If terminated by EOM character, discard the entire line and revert to character mode after repositioning the carriage.

4. Type mode. While active, enable the override. When one record is done, remove from queue. If the queue is empty, disable the override, clear the override flag, and proceed to recovery. If not empty, and override is not flagged, start the next operation. If override is flagged, proceed to override mode.

5. Override mode. For the duration of the type mode, the console interrupt is active as an override. If pressed at this time, override is flagged, and at the end of the current line instead of proceeding to a new record override mode is entered for the duration of one input request. The input is recovered, and any interrupted input is continued. This allows an input to be forced through a large number of successive outputs. Override mode is recognized by the console interrupt light being lit while the mode is active.

B. 9 Diskfiles

In a paged environment where contiguous pages are not necessarily available, there are two methods of combining a collection of pages into a file. One method is to provide an ordered list of the component pages of the file. This scheme is illustrated by JANUS tasks, which are nothing more than executable files. In this case, the list is the USAGE table on the TCP. This scheme allows random access of file pages, but requires the allocation of a fixed size block to hold the list, thus setting an upper limit to the size of the file.

The second method is that of "chained" files, where each page includes information as to the next and last pages in the chain. While necessarily less flexible than the first method with respect to access, the limitations on chained files are no stronger than would be imposed by the use of addressable magnetic tape, where a single record in the middle of the file can be rewritten. The rest of this section will concern itself with chained files exclusively.

JANUS has certain routines and tasks which deal with files. To make use of these, the JANUS file conventions must be followed.

Each page of a file is a true page (512 words) of fixed format. The format of each page is:

Word 0 0000XXXX Pointer to last file page in chain,

Word 1 0000XXXX Pointer to next file page in chain,

Word 2-511 Available file page data area.

A null (0) pointer indicates the last page of the file in that direction. The first page of a file is in the FILE NAME.

One resident routine is:

Unfile*

R6 ****XXXX FILE NAME.

The FILE NAME is pushed onto a resident stack. If successful, the MORTICIAN task is signaled, and will eventually come in and chain forward through the file, freeing the diskpages. In this case, only the forward pointer on each page is examined. Note that if a page in the middle of a file is presented, preceeding pages are not touched.

In order to avoid confusion which might arise, I wish to define certain terms.

A file consists of one or more blocks. END-FILE corresponds to the terminal condition of the last block, going forward.

A block consists of one or more records, terminated by a MARK or END-FILE. A MARK is completely equivalent to a tape mark, which however is sometimes referred to as an endfile. This usage is not followed here. Similarly, a block is sometimes referred to as a file, but this usage is again not followed.

A file is normally of constant format--RECORD or STREAM. A RECORD FILE has as part of its specification, a fixed record size parameter. In this case, all records are of identical size. These records are normally packed into a file page, with no spaces between them.

Conversely, a STREAM FILE does not have fixed sizes. Instead, each record has associated with it a record size discriptor (COUNT), normally the first item in the STREAM RECORD. To locate the next record, it is necessary to combine the record pointer with the COUNT to develop a new record pointer. STREAM RECORDS have the advantage of improved packing density. However, it is occasionally desirable to locate a previous record. In order to locate the beginning of a record, knowing the end, we must have the size. Hence we define a REVERSABLE STREAM FILE, characterized by SUPER-RECORDS, which consist of STREAM RECORDS bounded

at each end by a SUPERCOUNT.

COUNTS and SUPERCOUNTS are normally of the same resolution as the elements of the record: in all future discussion I will be referring to a record of bytes, COUNTS and SUPERCOUNTS which fit into a byte, and which specify the number of bytes.

Given a record of N bytes, a STREAM RECORD would consist of N+1 bytes, the first of which (COUNT) would contain the value N. (This is also referred to as TEXTC format, from the usage in the assembler.) A SUPER-RECORD would consist of N+3 bytes, where the first and last (SUPERCOUNT) would have the value N+2, and the N+1 bytes between would be a normal STREAM RECORD.

In addition to the records, it is desirable for the file to contain control information. Immediately apparent examples are END-FILE, ENDPAGE, and MARK. CONTROLS should be readily distinguished from normal records--as no normal record has a COUNT of zero, this is our identifier. A CONTROL is a one-byte record with a COUNT of zero, except in one special case. Thus there are 256 possible CONTROLS. A REVERSIBLE CONTROL has no COUNT--the control byte follows the initial zero SUPERCOUNT directly. Except in the case of the ENDPAGE condition, each REVERSIBLE CONTROL is three bytes of form 00-XX-02. In the special case of the ENDPAGE condition, the rest of the page is the ENDPAGE RECORD, and the last halfword on the page is the SUPERCOUNT, which however is still a byte count. This is necessary, as a single byte is too small to be able to specify a large ENDPAGE RECORD.

It is important that these conventions for reversible files be known, as one system task (SYSGEN) is capable of generating reversible library files.

Unique controls defined are as follows:

00 ENDPAGE. This is also the END-FILE condition, when there is no forward chain specified for the page.

01 PAUSE. Normally causes the receiver of the file to halt until some condition is satisfied.

02 MESSAGE follows. The record following is not a normal data record, but instead contains special information to the reader.

09 File is switching to BINARY, or to UNFORMATTED mode.

0D File is switching to BCD, or to FORMATTED mode.

10 MARK. Used to identify the end of a block of records.

FF. Null. This control is just filling space, and is to be ignored. It may be used, for example, to clear a PAUSE condition on a driven file. (A driven file is being read as it is being written, thus the necessity of halting the receiver if it catches up to the driver.)

B. 10 Symbionts

A symbiont is a task which performs a limited set of self-defined operations on a file, usually transcribing it to another file. This discussion will be limited to I/O symbionts only, where exactly two files are involved, one of which is associated with a physical device.

Consider first the range of applicability of such a symbiont task. Since it must be capable of transmitting a stream of records from successive files, it cannot be used for an interactive (bi-directional) device, such as a keyboard or graphic display. Furthermore, since it deals with STREAM FILES, positioning operations are not readily implemented, or are meaningless if included.

In this context, we see that the range of applicability covers only

those physical devices which are mono-directional. These would include card readers, line printers, card punches, plotters, and under certain circumstances, magnetic tapes, Specific idiosyncrasies of particular devices require specific conventions, which in general conflict with those of other devices.

A set of symbiont tasks has been constructed for several of the commonly used I/O devices. This set could be expanded, but currently includes the cardreader, lineprinter, cardpunch, and plotter. These symbiont tasks are system tasks, and special resident routines are provided to permit any task to communicate with the requested symbiont task.

The calling sequence is the same in all cases:

R6 XXXXYYYY,

R7 SSSSSSSS Signal to send on completion,

where YYYY is the name of the first page of a standard non-rewindable stream file. XXXX are attribute bits. Currently the only bit defined is the first (8000), which signifies that the file may not be discarded after the symbiont is through with it. Unless that bit is set, output symbionts will return each component file page to the system pool as the information thereon is used.

The resident entry points are:

CRA03SYM,

LPA02SYM,

CPA04SYM,

PLA06SYM.

The operation in each case is identical.

On entry, R6-R7 are pushed onto a small resident stack unique to the specific device. If not successful, CC4 = 0 is returned--otherwise

a Signal is sent to the task, which is always on the ring of tasks.

When the task cycles through, it empties out the stack, putting each request into a queue internal to the task, in the order requests were made. If the task is working on a file, it is busy.

A busy symbiont, on completion of a file, frees the I/O device, and deletes that file entry from the queue. If the queue is empty, the task goes into the wait state, and upon return, checks the queue again. If the queue is not empty, it requests the device anew, and when the device is assigned, proceeds with the next file in sequence. This mode of operation, freeing and reallocating the device after each file, permits any other task to sneak in and acquire the device to perform it's own I/O operations.

Once a symbiont has been given a file, it is controlled from the teletype, by means of the device prefix. Valid key-in controls are:

GO Continue with the current operation.

DOWN Flag the device down, save the file for further continuation when the device goes up.

ABORT Discard the rest of the file.

In using input symbionts, (currently the cardreader only), a page is given to the symbiont. The symbiont will allocate further pages as necessary, and perform all necessary chaining operations. The specified signal is returned to the controlling task when the END-FILE condition arises.

B. 11 Control Commands and the Amperscanner Task

All useful operating systems incorporate some means of communication with the outside world. Under JANUS, this function is provided by

a system task, the Amperscanner (the name is contrived from ampersand, which is the teletype prefix used to direct input to JANUS, and scanner, which function the task performs).

The Amperscanner always has a request pending for input on the console teletype, unless it is actively precessing a line of input. Input is formally free field--that is, keywords may have no imbedded blanks, and must be separated by one or more blanks or other delimiters.

The control commands are formed by a keyword, followed by possible modifiers. Unused modifiers are ignored, unless they actively garble the meaning of a command. In the case of a garbled or unknown command, no action is taken, and a question mark is output on the teletype. Keywords fall into three main categories.

One keyword category is that where they keyword is the name of a library task. In this case, the Amperscanner will locate the task in the library, and will proceed to make a working copy of the task, performing a copy operation on all those pages which will be modified, on the basis of information kept in the master copy of the TCP. The copy is then added to the ring of tasks, and signaled to start. If unable to start a task, either because of a lack of diskpages, or because JANUS can accept no new tasks, an appropriate comment is produced and the operation is aborted. The Amperscaner is the parent of all such tasks, and will take care of destroying them after they exit.

A second category permits changing various system parameters, such as time and date.

The third category permits one to query JANUS as to the status of various features, such as resources. One keyword in the category (MANUAL) causes a standard system file to be printed, providing a manual of

operation which lists all keywords implemented, and includes a brief description of each one's significance.

B. 12 The Housekeeper Task

As mentioned previously, all possible effort has been made to keep the JANUS resident as small as possible. To this end, all large data sets and little used code which need not be resident are kept in the Housekeeper task, and brought into core on demand on a timeshared basis. Thus, infrequently used functions, such as the calendar and date computations, are provided by the Housekeeper, without tying up core memory.

Since the Housekeeper is the major non-resident system task, it has certain non-standard features. For example, the housekeeper TCP is dedicated permanently in core, in the first page above JANUS. In this sense, it is the Task Control Page of JANUS itself--thus JANUS can be considered a task. In normal circumstances, the TCP of all other tasks map over the JANUS TCP, and it is not seen by other tasks. To all other tasks, there is just an additional page of memory which is inaccessible. However, it is the TCP of unmapped JANUS, and thus any unmapped trap which occurs funnels through the Housekeeper TCP. (Unmapped traps are discouraged, and are normally a sign of either programming or machine error.) In addition, this page contains a number of standard system console teletype messages used by the Housekeeper and Amperscanner. These are kept here so that valuable mapped task address space will not be cluttered up by storage not relevant to the tasks.

The Housekeeper is invoked whenever an unusual circumstance occurs. This includes the case where a resident stack is depleted or surfeited.

Whenever the Housekeeper is executed a standard function is to cause all resident stacks to be adjusted until exactly half full.

In certain freak cases (which experience has shown occur most rarely), it is possible that a request from a task to JANUS, which affects one of the resident stacks, cannot be fulfilled. In this situation, it is possible for resident routines at any level to call upon the Troubleshooter, a special resident routine. This routine has the power to override the whole system, in a last-ditch effort to stay sane. It can, if necessary, actually take the RAD away from the Swapper, checkpoint storage, and bring in enough of the Housekeeper to attempt to recover. If it was necessary to checkpoint core, that core will be restored after the recovery attempt.

Because of the interruptable nature of tasks, and because of the possibility that the Troubleshooter may be invoked by a high level interrupt, it is necessary that the Housekeeper be reentrant. As a result, resources are normally kept track of in three different areas, in such a way as to defeat the interruptable nature of the Sigma 7. For example, stack manipulation instructions are not interruptable. As a result, resources have a resident stack, an intermediate Housekeeper stack, and the Housekeeper data set. Data are transferred via stack operations between registers and stacks. The intermediate stacks are buffer stacks; kept half full, data may be transmitted between them and the resident stacks by both the Troubleshooter (which uses the intermediate Housekeeper routines) and the Housekeeper. The Housekeeper calls upon the intermediate routine to straighten out resident storage, and then transfers data between the intermediate stacks and the main data sets of the Housekeeper. While unwieldy, this process permits

JANUS to run without inhibiting external interrupts used for realtime applications, and still guarantees that no datum will be lost or duplicated.

By adequate arranging of routines in the Housekeeper, such that all reentrant storage is at the beginning, it is possible for the Troubleshooter to need only a part of the Housekeeper. As JANUS now stands, it is only necessary for the Troubleshooter to use one page in addition to the Housekeeper TCP.

In addition to the above functions, the Housekeeper may be signaled by the Amperscaner to perform certain operations, such as outputting standard messages to the console. It may be called upon by JANUS for similar functions, as when a machine error (such as watchdog times runout) occurs.

APPENDIX C.

The JANUS Basic/File Control Monitors

When JANUS was undertaken, it was realized that the execution of FORTRAN programs would be a major requirement. It was felt that, since SDS had supplied with the computer a Basic Control Monitor (BCM) [14] and various processors such as FORTRAN, SYMBOL assembler, LOADER, and DUMPING LOADER--some of which would be used in generating JANUS--it would be an excess waste of effort to generate our own version of the processors, especially since SDS would maintain theirs. As a result, it was decided to build a timesharing monitor which would interface the SDS-provided processors to JANUS.

A brief examination of the BCM showed that it could not be readily changed to our requirements, if for no other reason than that it was inadequately documented. (One of the requirements of acceptable JANUS coding arose from this experience--all code was to be adequately documented with comments in the source such that anyone familiar with the computer could easily understand any part of any JANUS code.) As a result, it was decided to start completely anew to write the JANUS Basic Control Monitor (JBCM) which would perform the necessary interfacing functions.

With certain exceptions, the JBCM is used in exactly the same way as the BCM. These exceptions are usually minor, and except for some

changes in certain control cards, a program which operates under the BCM will also operate under the JBCM.

The difference in control cards are as follows:

1. The JOB card must have the users name on it. This is frequently the only way to identify output from several tasks.

2. Assignment of files to TYA01 (the console teletype) are invalid and are sufficient reason to exit a job.

3. The SDS-defined BIN, BCD, and EOD cards are replaced by the standard JANUS BIN, BCD, and MARK cards, respectively.

4. In an effort to cut down the number of control cards used, the loader was modified so that the data card is no longer needed, or acceptable.

5. Many of the operator system keyins have been deleted, leaving only X, E, and F (Fin, which will abort the task).

Further changes are that, since the I/O is buffered in the JBCM, the check function is a null operation. Devices are also addressed differently than under BCM.

In compensation, additional features have been implemented. These are:

1. Additional system DCB's, namely M:SI, M:GO, M:CI, M:CO, M:MAP, and M:GR. Some of these are defined in the SDS Batch Processing Monitor (BPM), a higher level monitor than the BCM. The default device assignments are CRA03, DFAF0, CRA03, CPA04, DIAF0, and PLA06 respectively.

2. Default specifications are assigned to the FORTRAN DCB's, to agree with the FORTRAN usage for READ, PRINT, and PUNCH.

3. M:BI and M:BO are assigned to DFAF0 (as was M:GO).

4. M:GO is rewound at the beginning of a job, and when a LOAD

card is encountered. As a result of these default specifications, the control cards required for a load-and-go FORTRAN job are:

```
JOB NAME
FORTRAN
(
( FORTRAN DECKS
(
LOAD
RUN
(
( DATA DECKS
(
```

5. Certain monitor functions implemented in BPM but not in BCM have been implemented in the JBCM. These may be found in the BPM manual under the titles:

M:TIME,

M:KEYIN,

M:TFILE.

6. Reasonable English language error messages. (This was a major complaint with the BCM since the error message ERR 65 02 01 00 covered more than fifteen different errors, from referencing nonexistant memory to the floating point calculation -A+A .)

7. The inclusion of extra control commands, such as UNLOAD.

By allowing multirecord I/O buffers, the timesharing capability is greatly extended. As a result, there is one page of monitor alloted to each I/O device, used for buffer and interrupt routines. In addition, a demand paging algorithm is included, as are certain other functions.

The JBCM is divided into two areas: slave and master. The master-mode area is interfaced directly to JANUS. The slave-mode area contains those functions which provide the personality of the BCM. All control card, I/O, trap handling, and other functions specific for the BCM operate in slave-mode, and are demand paged. The slave-mode monitor area is five pages in extent. However, when a program is running, using the JBCM only for I/O, only one of those pages need be in. Furthermore, these pages are write protected and need never be written back to the RAD, improving swap efficiency.

A recent rewrite of the JBCM introduced a new feature. By rearranging the mastermode storage, and adding code which was assembled on the basis of an assembly parameter, it became possible to reassemble the JBCM in the JFCM mode. The JFCM (JANUS File Control Monitor) differs from the JBCM mainly in the mastermode area. Whereas the JBCM is device oriented, the JFCM is file oriented, and does all I/O operations using the system symbionts. The JBCM permits one to provide a program parameters in a conversational mode, since the JBCM is connected directly to the devices. The JFCM conversely causes an entire card file to be input before starting, and does not cause output until the job is done. As a result, it "swallows" jobs, freeing all I/O until the end of the job, and runs truly in the background. Since it uses no I/O devices directly, multiple copies of the JFCM may run simultaneously. Furthermore, since over half of the JFCM is never modified, the original copy of the page is used in each copy running, and is thus common to all the JFCM tasks. As a result, there is a much higher probability that part of the monitor is in core, necessitating no RAD operation, and making swapping more efficient. The JFCM has the obvious capability of being

fed a long job, and then feeding short jobs to another JFCM or to the JBCM, to be completed and output while the first job is still being digested, thus shortening turn-around time for short jobs, without significantly affecting the time required for long jobs.

APPENDIX D.

Notes on Cyclotron Control Implementation

One of the design goals for the use of the Sigma 7 is the control of the cyclotron by the computer on a timeshared basis. In the ultimate form, this would be powerful enough for an experimenter to invoke the task, type in a minimum set of parameters (such as particle, energy, beam intensity, energy resolution, and experimental station), and then wait the necessary time for the computer to inform him that the cyclotron is operating under those conditions. The computer would then control the cyclotron, informing the experimenter when the beam was outside the range specified, or if an abnormal condition occurred. This would continue until the experimenter signed off, at which point a diagnostic listing would be printed out, giving information of use to the cyclotron service personnel.

How much of this dream is possible? Since the computer is replacing people to make adjustments and measurements, the access time required is of human speed, on the order of seconds. It becomes perfectly feasible to control the cyclotron from a demand paged slave-mode task. This is good, since a program of the complexity described would have to be written in a higher level language, such as FORTRAN, which is notoriously untrustworthy for computer control operations. Such a program can be written. Where then do the constraints lie?

The first constraint is obviously the matter of hardware suitable for computer control. Since this is outside the range of this thesis, I will not comment further, and continue under the assumption that it is available.

The second constraint is the task monitor. This would be similar to the JBCM or JFCM in many respects, but not identical. (For control, a realtime capability is required, and if this were available in the JFCM, any user could accidentally or maliciously send false control signals.) On the other hand, many of the functions provided in the JBCM/JFCM could be eliminated. Under the mild constraint that the FORTRAN be compiled under the JBCM/JFCM, one can delete all processor and control card handling. Furthermore, the FORTRAN runtime and math library actually reference the monitor in only three or four places, primarily for I/O. By changing these routines to interface directly with the mastermode monitor, then all of the JBCM/JFCM slave-mode storage could be eliminated completely.

Again, I/O can be limited to files in addition to a teletype. As a result, the obvious candidate for the monitor would be the JFCM. From this, one could probably delete all I/O but plotter and lineprinter, since a JANUS Cyclotron Control Monitor (JCCM) would not need to read or punch.

Thus, the JCCM could be made fairly simply from the JFCM by adding a small realtime handler (probably under 40 words), changing the FORTRAN runtime, and deleting great pieces of JFCM code. When this action is finally desired, it can be done in a relatively short time.

The third constraint, and probably the greatest, is the cyclotron controling program written in FORTRAN. It will undoubtedly be based upon

or similar to the SETUP program, in order to compute the operating conditions. There will then be additional subroutines to read the cyclotron parameters, compute the correction required, and perform the necessary control operations. At this point, the additional complexity introduced, of recognizing components which are drifting excessively so that maintenance may be performed, will quite likely be a minor perturbation on the amount of work involved. The program will doubtless be written piecemeal, with each new capability checked first under computer simulation (which must also be written and debugged), and then in real life. Furthermore, a new control capability may well contradict an older capability, requiring reprograming and even redesign of hardware.

Before any great amount of work is done on cyclotron control, an effort should be made to define the problem for all concerned. Once a definition is agreed upon and is available, it is time to specify hardware, software, and scheduling configurations. A control program of this complexity cannot work well if built of independent modules--each function will interact with others, and must be thoroughly checked, first for correctness of operation, then for interactions, and finally for successful operation in a timesharing environment. As for any complex realtime JANUS task, it is desireable to first develop an operational stand-alone system, with the foreknowledge that it will eventually be timeshared, and only when it is working rationally in a stand-alone configuration should the timesharing features be added. This is simply because for basic testing, timesharing is superfluous, serving only to confuse results, and without any real gain, since any realtime program could not be trusted in a timesharing environment with any other user

until completely debugged. Simulations can be performed under JBCM/JFCM, but any actual control attempts should be completely debugged under the BCM as a stand-alone system.)

What sort of problems might one encounter in a cyclotron control program? There would, of course, be various codes for computing initial parameters for a specific operating configuration on the basis of theory. Associated with these would be a capability to set the cyclotron controls to the correct value, in the correct sequence and with the proper timing. This phase could be readily generated from programs which exist today, such as the SETUP code. There would also have to be procedures available to make actual measurements, from quantities as simple as voltage, current, and pressure, to patterns as complex as the cyclotron turn pattern. These would introduce the need for pattern recognition codes capable of analyzing the current state of the cyclotron, and determining how the machine should be returned for best operation. There might be "learning" features, such that the program could vary parameters used in calculation such that control operations would converge faster. (Under JANUS, this could be readily implemented by keeping these parameters in a set of pages in the task library, modifiable and of which the original copy would always be used by the task. Learned parameters would thereby be changed in the library source.) These features will probably take a great deal of time to develop.

Thus, cyclotron control programs contain three types of functions--initialization, sequencing and stabilization, and tuning. The first two have already been implemented to some extent without the use of an on-line computer. They could be implemented fully without the computer through standard feedback techniques at probably less cost than if the

computer were used. The real advantage of using an on-line computer would come about by implementing the third--and hardest--category.

As a result, while I feel that this goal is possible and probably feasible, an actual computer controlled cyclotron of any significant value is probably a minimum of two years away, and the system described in the opening paragraph is more likely five years away.

APPENDIX E.

Notes on Conventional Terminal Implementation Under JANUS

One of the most important uses of conventional timesharing systems has been to allow simultaneous access to the computer from several remote terminals. These terminals generally consist of low speed I/O devices, such as teletype keyboards and paper tape. High speed devices, while not technically impossible, are uncommon due to the cost of a wide-band data link, required for a high data rate. Let us consider terminals consisting of teletypes only, possibly with low speed paper tape facilities. How might multi-terminal operation be implemented in a tasking environment such as JANUS?

Consider first the manner in which a large number of terminals would be coupled to the computer. The easiest (and most expensive) method would be for each terminal to be connected to the IOP separately. Much more likely would be the use of a communications link. In a communications link, a large number of terminals are connected together as a single I/O device, which includes a computer controlled switchboard. A mechanism is provided to scan all terminals for the presence of an input signal, as are remote addressing mechanisms. If the link were not busy, the presence of an input signal would cause an interrupt. The interrupt routine would have to perform a scan to locate the specific terminal requesting service, perform the necessary operations

to switch the data link to that terminal, and initiate a data transfer (normally one character). After the data transfer was accomplished, the character input would need to be examined to determine if it were a specific "break" character, such as a carriage return, necessitating specific action. Once that action was initiated, the terminals would be rescanned for more input requests. If present, the process would be repeated. If not, any output pending would have to be precessed, again by setting the data switchboard to the specific terminal and initiating a character transfer.

The use of a communications link implies the use of a centralized I/O handler. The question of terminal implementation can thus be rephrased in terms of the broader issue--that of implementing centralized I/O in a highly decentralized task structure such as used in JANUS. To put the following discussion on a firmer basis, I will specifically discuss the simulation of an existing timesharing system, the BASIC system developed at Dartmouth College and expanded by General Electric [15].

Assume that each active terminal requires a 100-byte input buffer, a 100-byte output buffer, and approximately 50-bytes of status information, such as a list of current break characters. Assume further that the communications link controlled 64 terminals. A possible implementation might be as follows:

Under these conditions, it would be necessary to dedicate 4-k words (64 terminals x 64 words/terminal) for terminal data blocks. This immense I/O package would be a part of a LINK task. The LINK task would have one main function--when a terminal signed on, the LINK would start a subtask, unique to that terminal, by making a copy of a standard TERMINAL task, and informing it, through the medium of the TCP, of the

location of the specific buffer block allocated to that terminal. The LINK task would also have the job of destroying a TERMINAL task after sign-off, and of communicating specific items of information (such as reports of system changes) to the terminals when they were inactive.

The TERMINAL task would have many functions, including user recognition, accounting, and recognition of standard commands. For some standard commands and functions, such as OLD, it would have the capability of locating the user's entry in the system file, locating his specific file therefrom, and finding a specific program file therein. For other functions, such as RUN, it might actually start a specific subtask, providing it with the terminal block information and an input file. The extent of using subtasks for various functions depends greatly on their complexity. As only one function is in use at a time, the decision as to whether to demand page functions within a task, or to start subtasks, is somewhat academic, being limited primarily by the fact that the maximum size of a task is bounded by address space limitations. One main function of the TERMINAL task, however, would be the taking of appropriate action on each line of input (as signaled by the interrupt routine in the link task), performing the necessary operations if a command, and transferring the line to the appropriate file if not (in a duplex system, the interrupt routine would be charged with the echoing of input, and such operations as backspacing). By using large buffers, the TERMINAL task need not have a small access time, especially as the BASIC system does not check syntax until execution time, and therefore there is little problem with fast response with diagnostic messages.

The implementation described is not the only possible or best

implementation--since terminal usage of this sort is not forseen at our installation, little effort has been expended to work out all the details required, other than the broad outline, to determine if JANUS could be used in such an application. From the above discussion it should be clear that it can, and may even be capable of handling as many as several hundred terminals at once. How well JANUS would do would depend on many other factors, including the actual implementation used and the efficiency of code developed (how much code could be used in common to several terminals), and is therefore open to interpretation.